

MENG FINAL YEAR PROJECT

IMPERIAL COLLEGE LONDON  
DEPARTMENT OF COMPUTING

# Augmented code editor for a functional programming language

*Harry Lachenmayer*

Supervisor: *Prof. Susan Eisenbach*



## **Abstract**

Integrated development environments (IDEs) are very powerful and complex tools which help programmers to understand the programs they are developing. Such sophisticated programming tools leverage the unique properties of a single programming language and environment to provide specific features to aid programming. The only IDEs in wide use today are designed for imperative, object-oriented programming languages; practically none exist for pure functional programming languages such as Haskell. This is surprising, since such programming languages have a number of unique properties which make them particularly suitable for creating novel editing experiences.

The aim of this project is to create an editing environment for a pure functional programming language. This editor allows the programmer to instantly inspect any definitions in the program, and enables context-specific editing interactions for certain types of values. The editor exploits the abstract structure of the program to enable these interactions. In contrast to traditional “structural editors”, all changes made using the structural interface are also performed on a textual level. This allows the editor to integrate with existing programming systems and workflows, such as version control systems, which is a weakness of current structural editing tools.

In this project, I explore a number of innovative systems for programming which attempt to move programming away from editing plain text, and explore the design and implementation of a novel “augmented” code editor which combines the advantages of structural editing and plain text editing.



## Acknowledgements

I would like to thank the following people:

Michal Srb – for endless late-night discussions over the past four years about the terrible state of programming, and what we will do to fix it.

My supervisor, Susan Eisenbach – for giving me a place in the Haskell tribe.

My second supervisor, Sophia Drossopoulou – for terrifyingly incisive questions and feedback.

My friends, particularly Christina, Ciara, Emi and Sean – for letting me escape the world of programming when I needed to.

My parents, Karin & Bernd Lachenmayer – for always allowing me to tread my own path.

And most of all, my grandfather, Helmut Geese – for introducing me to the beautiful worlds of programming, mathematics and music.

Without you, none of these ideas would have existed. Thank you.



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
<b>2</b>	<b>Principles</b>	<b>15</b>
2.1	Creators Need an Immediate Connection to What They Create . . . . .	15
2.2	Environment & Language . . . . .	15
2.3	Human & Machine . . . . .	17
2.4	Integrate with Existing Systems . . . . .	17
<b>3</b>	<b>Prior Work</b>	<b>19</b>
3.1	Bret Victor – Inventing on Principle . . . . .	19
3.2	Elm’s Time-Travelling Debugger and Elm Reactor . . . . .	22
3.3	Light Table . . . . .	24
3.4	Lamdu . . . . .	27
3.5	Tangible Values . . . . .	31
3.6	“Blocks”-based graphical programming environments: Scratch and Hopscotch . . . . .	34
3.7	Touch-based programming environments: TouchDevelop . . . . .	35
3.8	Notebook environments: IPython Notebook and Gorilla REPL . . . . .	37
3.9	Conclusion . . . . .	40
<b>4</b>	<b>Functional Reactive Programming with Elm</b>	<b>43</b>
4.1	Definitions: Immutable Values, Functions and Types . . . . .	43
4.2	Evaluation: Purity and Referential Transparency . . . . .	45
4.3	Collections and Higher-Order Functions: <code>map</code> , <code>filter</code> , <code>fold</code> . . . . .	46
4.4	Structured Values: Tuples, Records and Sum Types . . . . .	49
4.5	Dealing with Time: Functional Reactive Programming . . . . .	51
4.6	Putting It All Together: The Elm Architecture . . . . .	52

<b>5 Design</b>	<b>56</b>
5.1 Types . . . . .	56
5.2 Definitions: Functions & Values . . . . .	57
5.3 Function Composition via Drag & Drop . . . . .	58
5.4 Evaluating Definitions . . . . .	61
5.5 Interactive Editing . . . . .	63
5.6 Errors . . . . .	65
5.7 Conclusion . . . . .	66
<b>6 Arrowsmith</b>	<b>68</b>
6.1 User interface . . . . .	69
6.1.1 Definitions . . . . .	69
6.1.2 Imports . . . . .	74
6.1.3 Type definitions . . . . .	74
6.1.4 Evaluating a Module . . . . .	76
6.1.5 Plain Text View . . . . .	76
6.1.6 Project View . . . . .	76
6.2 Architecture . . . . .	78
6.3 Front-end . . . . .	78
6.3.1 Editor . . . . .	79
6.3.2 Environment . . . . .	82
6.3.3 Value Views . . . . .	83
6.4 Back-end . . . . .	85
6.4.1 Web Server . . . . .	85
6.4.2 Compilation . . . . .	86
6.4.3 Module . . . . .	87
6.4.4 Editing . . . . .	88
6.5 Limitations . . . . .	89

<b>7 Evaluation</b>	<b>92</b>
7.1 <b>read the vocabulary</b> – what do these words mean? . . . . .	92
7.2 <b>see the state</b> – what is the computer thinking? . . . . .	93
7.3 <b>follow the flow</b> – what happens when? . . . . .	94
7.4 <b>create by reacting</b> – start somewhere, then sculpt . . . . .	95
7.5 <b>create by abstracting</b> – start concrete, then generalize . . . . .	96
7.6 Conclusion . . . . .	97
<b>8 Future Work</b>	<b>98</b>
<b>9 Conclusion</b>	<b>100</b>
<b>Bibliography</b>	<b>102</b>



# 1 Introduction

Programming editors have not fundamentally changed since the 1970s. In 2015, most of us still program by editing a plain text file using a text editor which display a small contiguous section of the file on the screen, navigating a cursor around the screen using our keyboards.

Compared to the average consumer-facing app, most programming tools in use today seem downright archaic. Yet, programmers defend these tools with religious zealotry. Indeed, after having invested nearly a decade of my life committing `vim`'s editing commands to my muscle memory, I can understand why we have not moved on: tools such as `vim` or `emacs` give the programmer text manipulation superpowers, and the ability to transform twenty lines of code in three keystrokes is still as compelling as it was forty years ago.

However, the image of the furiously typing hacker is a romantic one. Indeed, programmer folklore suggests that professional programmers write about ten lines of code per day.

In reality, the vast majority of a programmer's time is spent either looking at code, or looking at errors, in an attempt at trying to figure out what the code does. A programmer has to build a complex mental model of the program, essentially executing the code in her head.

A text editor, no matter how efficiently it may edit text, provides absolutely no help with this task. Of course, a programming system does not consist only of a text editor: there's the compiler, debugger, version control, bug tracker, documentation, etc.

Most of a programmer's day is spent switching between these various tools. We write some code, compile it, run it, it doesn't do what we want, add some log statements, compile it, run it, still doesn't do what we want, load it in a debugger, write some more code, ad nauseam.

To solve this problem, we created “integrated development environments” (IDEs). IDEs are very powerful tools which enable the programmer to edit, compile, run and debug programs in one single place. In particular, these tools incorporate some very helpful features for making sense of a program. A good IDE allows the programmer to navigate through large codebases efficiently, and to perform powerful restructuring of code.

In order to support these features, most IDEs tend to be tied to a single programming language. The only IDEs in wide use today are designed for imperative, object-oriented languages such as Java or variants of C, such as C++ or Objective-C.

Such powerful tools currently do not exist for functional programming languages such as Haskell – programming in a functional programming language nowadays tends to be an exercise in switching between various incompatible command-line

tools. I find this surprising, since pure functional programming languages have a number of interesting properties which enable some powerful editing features.

Programming in a functional programming language is a very different experience to programming in an imperative language. In an imperative language, the programmer writes lines of code which get executed one after another. These lines of code might declare new variables, or assign new values to variables. Since these lines of code depend on and manipulate state established by previous lines, single lines of code are rarely meaningful outside their specific context inside a procedure.

In contrast, in a pure functional program, the programmer creates definitions which establish “universal truths” within a program. These definitions can be either constant values, or “pure” functions. Functions can be seen as values which are dependent on other values. A pure function is a function whose output value is always the same when given the same inputs.

The concept of variables does not exist in functional programming. Once a value is defined, it can never be changed. The only way to “change” a value is to apply a function to it.

A pure functional program is simply a collection of definitions. For the computations specified in the definitions to be performed, they need to be *evaluated*. Since values never change, it is irrelevant *when* a value is evaluated. This is a key insight which has important implications for the design of programming tools for functional programming.

Due to the fundamentally different nature of functional programming, many programming tools designed for imperative programs are not very useful. One such example is the step-by-step debugger, which allows the programmer to pause the execution of her program and to step through instructions one step at a time. Since functional programs are not executed in a linear fashion, it is not very helpful to step through the code line by line. Instead, tools are required which allow the programmer to inspect the flow of values through the program as they are being evaluated. Various implementations of such tools already exist. These will be described in section 3.

Inspired by the insight that definitions can be *evaluated* independently, we will explore the idea that definitions can also be *edited* independently. Current programming editors allow the programmer to edit a program one file at a time. Instead, I propose exposing separate editing fields for each function. While this is a subtle change, it enables some powerful new features. In particular, this enables us to expose a simple interface for evaluating definitions in line with the code. Additionally this change allows us to explore some ideas for context-specific interactive editing tools which move beyond plain text editing.

Rather than editing code as plain text, it is possible to edit a program on a structural level, using a “structural editor”. Structural editing tools allow the programmer to manipulate an interactive visual representation of a program’s

abstract syntax tree (AST). A big advantage of this approach is that it entirely removes an entire class of programming errors: the syntax error.

In section 3, I will examine a number of tools which provide unique ways of editing a program. Some of these tools enable completely new editing interactions that are far superior to editing plain text. While structural editors are conceptually very appealing, none of these tools have ever had mainstream success. I argue that this lack of success is not mainly due to limitations in the editing interactions.

Instead, I argue that these tools have failed mostly because they tend to operate on a specialised environment and support limited interaction with currently established programming tools and workflows. Many such tools try to eliminate the textual representation of a program entirely, replacing it with novel representations such as “blocks” or “bubbles”. Such representations, while powerful in some respects, limit the programmer in others: some changes that are trivial to perform in plain text become extremely tedious in such editing interfaces.

Since these tools do away with text entirely, currently existing structural editors are completely incompatible with essential tools for programming, such as version control systems. Thus, choosing to use such a structural editing environment tends to be an “all or nothing” proposition. Convincing an entire team or organisation to switch over to some unproven editing environment which is incompatible with nearly every other programming tool is effectively impossible.

Plain text has the unique advantage that it can be manipulated and inspected with a huge number of specialised tools; text truly is the “universal interface” extolled by the UNIX philosophy. Programmers have the freedom to choose whichever editor they like best, and additionally, text can be manipulated extremely efficiently with a huge number of non-interactive processing tools.

I thus argue that *replacing* text editing with structural editing is a misguided goal. Rather, I propose that programming tools should *augment* the plain text editing experience by giving the programmer contextually relevant structural editing interfaces, while still operating on plain text. The programmer should be able to change freely between different editing interfaces relevant to the task at hand. Any changes made with such editing interfaces should manifest themselves in a change in the textual representation of the program.

A key insight is that the *representation* of data is independent of the data itself. In the functional programming paradigm, it can be said that the representation of data is a function of the data. This insight applies to data which is manipulated by a program, but also to the program itself: plain text is merely a certain representation of the abstract structure of a program, and can be replaced by any other representation which acts on this structure. We will explore several consequences of this insight, especially the unique editing interactions it enables.

The aim of this project is to create a new editing environment which exploits the unique properties of a functional programming language. The specific programming language I chose is a language called *Elm*. This is a functional reactive

programming language which is specifically designed for creating interactive programs with graphical interfaces. This language has a number of extremely interesting properties which enable some unique editing interactions. We will explore these in detail in section 4.

In the course of this project, I created a number of novel user interface prototypes which take advantage of Elm’s unique features. This design work is described in section 5. I then used some of the insights gained from this to develop a working system, called *Arrowsmith*. Its user interface and implementation are described in section 6.

This system vastly improves on many other programming environments in some aspects of the programming experience. Big improvements in some other aspects are enabled, but not implemented. This is critically assessed in section 7, and specific possibilities for future work are discussed in section 8.

## 2 Principles

One of the most influential figures in the development of programming tools in recent years has been Bret Victor. In his keynote talk at CUSEC 2012, entitled “Inventing on Principle”[3], he lays out some fundamental principles for programming, as well as demonstrating some live programming environments. The main message I have taken from this lecture is the importance of “finding a guiding principle for your work, something you believe is important and necessary and right – and using that to guide what you do.” [3], 00:36

This project is guided by a number of fundamental principles which I hold to be essential for the successful design and evaluation of any programming environment. These combinations of these principles provide a clear conceptual framework, which I will continuously refer to throughout this report.

### 2.1 Creators Need an Immediate Connection to What They Create

The first such principle is that “creators need an immediate connection to what they create”, [3], 02:00 Any time you (the programmer, the creator, the “thinker”) make a change or make a decision, you should see the effects of that change immediately.

This principle is violated by most current programming tools: “You type a bunch of code into a text editor, kind of imagining in your head what each line of code is going to do. Then you compile and run, and something comes out. . . . But if there’s anything wrong with the scene, or if I want to make further changes, or I have further ideas, I have to go back to the code. And I edit the code, compile and run, see what it looks like. Anything wrong, I go back to the code. Most of my time is spent working in the code, working in a text editor blindly, without an immediate connection to . . . what I’m actually trying to make.” [3], 02:35

In the talk, he explores some solutions to this problem, which I will explore in depth in section 3.1.

### 2.2 Environment & Language

Bret Victor’s essay “Learnable Programming”[4], written in response to an implementation of some of the ideas demonstrated in “Inventing on Principle”, outlines an excellent conceptual framework for the creation of programming environments. Throughout this report I will use this framework as a point of reference against which I will evaluate various existing systems (in section 3), as well as my own creations.

In this essay, he outlines two key goals of a programming system:

- to support and encourage powerful ways of thinking
- to enable programmers to see and understand the execution of their programs [4]

He also provides an important distinction, which he himself attributes to Will Wright, creator of The Sims:

A programming system has two parts. The programming “environment” is the part that’s installed on the computer. The programming “language” is the part that’s installed in the programmer’s head. [4]

Conceptually separating the *environment* (ie. editor, compiler, runtime...) and the *language* (ie. grammar, “vocabulary” – programming constructs, abstractions, libraries) is essential. Most programming systems in use today owe their weakness to a misunderstanding of this distinction. Too often, programming language design is guided by compiler implementation; editing is mostly seen as an afterthought. Bringing this distinction to the center stage enables us to focus our design task even further:

The environment should allow the learner to:

- **read the vocabulary** – what do these words mean?
- **follow the flow** – what happens when?
- **see the state** – what is the computer thinking?
- **create by reacting** – start somewhere, then sculpt
- **create by abstracting** – start concrete, then generalize

The language should provide:

- **identity and metaphor** – how can I relate the computer’s world to my own?
- **decomposition** – how do I break down my thoughts into mind-sized pieces?
- **recomposition** – how do I glue pieces together?
- **readability** – what do these words mean? [4]

Various systems have been more or less successful at performing these tasks. By asking these particular questions, we are able to concretely evaluate and compare systems. In particular, I will use this conceptual framework to evaluate the success of this project, in section 7.

## 2.3 Human & Machine

In his book *Interface*, Branden Hookway provides a key insight into human-computer interaction, namely that “...the interface is not only defined by but also actively defines what is human and what is machine.” [14], p. 12

This insight is particularly visible in the historical development of programming tools. Programming a computer was originally performed at a machine-instruction level – the development of assemblers, compilers for higher-level languages, as well as software verification tools, can be seen as a process of passing to the machine tasks which were previously performed by humans.

This leads to another (certainly not original) defining principle which guides my work, which is:

*The programmer should never have to do anything the machine can do for her.*

## 2.4 Integrate with Existing Systems

In e-mail correspondence with Matthias Mueller-Prove, Alan Kay wrote the following:

When Martin Luther was in jail and contemplating how to get the Bible directly to the “end-users” he first thought about what it would take to teach Latin to most Germans. Then he thought about the problems of translating the Bible to German. Both were difficult prospects: the latter because Germany was a collection of provinces with regional dialects, and the dialects were mostly set up for village transactions and court intrigues. Interestingly, Luther chose to “fix up” German by restructuring it to be able to handle philosophical and religious discourse. He reasoned that it would be easier to start with something that was somewhat familiar to Germans who could then be elevated, as opposed to starting with the very different and unfamiliar form of Latin. (Not the least consideration here is that Latin was seen as the language of those in power and with education, and would partly seem unattainable to many e.g. farmers, etc.)

I think Martin Luther was one of the earliest great User Interface designers – because he understood that you have to do much more than provide function to get large numbers of people to get fluent. You should always try to start with where the end-users are and then help them grow and change. [20]

This struck me as an important concept, as it partially explains why we are still stuck with what I consider inferior programming systems. Programming tools do

not exist in a vacuum; solutions to many aspects of programming already exists. Many attempts at creating better programming environments try to rewrite the programming experience “from the ground up”. This approach tends to discard years of effort expended into improving (perhaps marginal, yet) meaningful aspects of current programming systems.

Rather than trying to do everything at once, effective programming tools fit into current environments and workflows, provide a meaningful improvement in (at least) one area of use, and provide ways to “grow and change” with the end-user.

## 3 Prior Work

The development of better programming tools has been an endeavour which predates computers themselves. Countless systems with the aim of improving the programming experience exist in some form or another. In this section, my aim is not to give a “complete” view of the programming environment design space, but rather to highlight the projects which have had the most impact on the development of my own programming environment.

As the old programmer’s quip goes, everything in computing was invented before the seventies – since then, we have just been reinventing the wheel with better hardware. This rings particularly true for programming environments. Indeed, many of the ideas exhibited by the systems that follow trace their lineage to some of the pioneering works in human-computer interactions from the sixties and seventies.

Considerable progress has been made in the field in recent years, spurred on particularly by a series of demonstrations, talks and essays by Bret Victor. Most programming systems developed since then have taken a considerable amount of inspiration from his ideas, so I will outline these further before examining specific systems. I will then explore some rather “direct” implementations of these ideas, moving on towards projects which aim to move programming away from text.

### 3.1 Bret Victor – Inventing on Principle

In his talk “Inventing on Principle”[3], given at CUSEC 2012, Bret Victor demonstrates a programming environment which embodies the aforementioned principle of a creator having an “immediate connection” to what they are creating. This programming environment consists of a code editor shown side-by-side with the image output that the code produces.

Even this simple layout change is already a marked improvement. The programmer does not have to switch back and forth between different environments (editor, compiler, output).

In this environment, the code and its result are tightly linked: any change in the code immediately triggers a corresponding change in the outcome. Every change is immediate, and the code and the image are always in sync. The programmer is not aware of any “compilation” step, and does not have to “reload” or “run” the code. Indeed, once such tight coupling is achieved, these terms become meaningless within the context of the editor.

Reducing the “edit-compile-run” cycle to just a single “edit” action with immediate feedback enables new ways of interacting with the code. Instead of merely typing text, the programmer in this environment can use interactive components

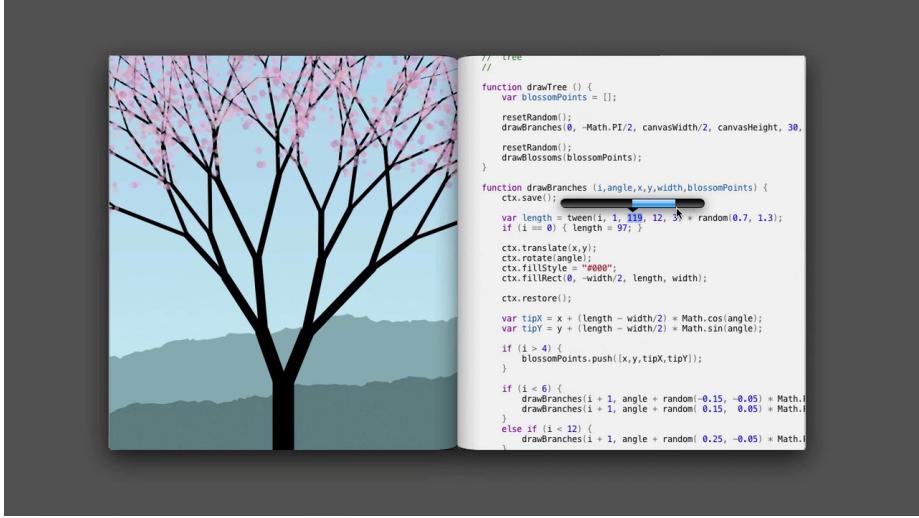


Figure 1: Changing a number using a slider. Bret Victor - Inventing on Principle 04:30

to change the values immediately. For example, numbers can be edited using a slider (see figure 1).

Most implementations of Victor’s ideas tend to focus on these specific interface elements. Much more important is the kind of interaction that these interfaces enable. Editing a number as text requires the programmer to choose a specific value, while a slider is fuzzy—the programmer can rely on intuition to choose a value that “looks right”. The programmer can now explore the effect of changing any value, instantly.

Indeed, the essence of these ideas is that “so much of art – so much of creation – is discovery. And you can’t discover anything if you can’t see what you’re doing.” [05:25] More precisely, it is crucial that “I can make these changes as quickly as I think of them. That is so important to the creative process: to be able to try ideas as you think of them. If there’s any delay in that feedback loop between thinking of something, and seeing it, and building on it, then there is this whole world of ideas which will just never be.” [08:49]

However, the environment as he shows it deals with a single static image, with no notion of state or time. At the time this talk was publicised, most of these interactions were already possible in certain ways, for example using a “live reload” plugin inside a browser, and laying out windows side-by-side. Since the talk, countless tools implementing these ideas have been created – I will explore some of these in detail. Notable implementations which I will not explore in further detail include Apple’s *Swift Playgrounds* which are integrated into Apple’s Xcode IDE[25], and the open-source *Tributary* environment[30].



Figure 2: Rewinding time. Note the slider in the top-left corner, which moves backwards and forwards through time. 12:54

In my eyes, the most important contribution of this talk is the demo of a 2D platform video game being created in the same editing environment. Again, the game is shown alongside the code. He is now also able to interact with the game itself, controlling the game character. He demonstrates changing some variables, which affect certain aspects of the game, such as how high the player character can jump, how fast the enemy character moves, etc. He is also able to pause the game. When he does, a slider appears (see figure 2): it allows him to move backwards and forwards through game time.<sup>1</sup>

Importantly, whenever he changes any of the values in the code, the resulting changes are applied on top of the current state of the game. For example, he can perform a “jump” in the game, rewind time using the time slider, increase the jump height (ie. acceleration in the  $y$  axis), and move forward through time using the slider: the game character will now jump higher.

This is already a vast improvement on the traditional edit-compile-run cycle: the programmer is able to identify bugs exactly where they appear, can apply necessary changes, and can test the changes immediately by stepping through the state of the program. This approach is not limited to video games, as will be explored in further sections.

However, this isn’t “immediate” enough: the programmer still has to manually move backwards and forwards through time, change some values, and again move

---

<sup>1</sup>Interestingly, the artwork for this demo was created by David Hellman, who also created the artwork for *Braid*[2], a 2D platform game which is based entirely on the concept of freely being able to move backwards and forwards through time.



Figure 3: Mapping time to space, by leaving a trail. 13:53

backwards and forwards through time. The key insight to solve this is that “if you have a process in time, and you want to see changes immediately, you have to map time to space.” [13:37] In the case of video games and animations, this can be done easily by leaving a trail caused by the movement of the character, as shown in figure 3. Any change in the values affecting the movement are now immediately visible to the programmer: the trail changes immediately as the values are changed.

The talk contains a number of other demos which embody the principle of giving creators a direct connection to their creations, but these are outside the scope of this report. Instead, I will now focus on working systems which have implemented these ideas.

### 3.2 Elm’s Time-Travelling Debugger and Elm Reactor

One of the most successful implementations of the ideas presented in “Inventing on Principle” has been Elm’s time-travelling debugger[10], implemented by Laszlo Pandi in February 2014. Elm’s programming model (as described in section 4) is perfectly suited to “time-travelling”. The entire state of the program is contained in one single data structure, which is only ever updated via pure functions. These functions are evaluated any time an external event occurs, such as when the user presses a key. The time-travelling debugger saves the values emitted by these events. When any of the functions in an Elm module are changed by the programmer, the debugger applies this sequence of events to the new module’s initial state. To go “back in time” then means only applying the events which happened prior to a certain point in time.

He demonstrates this debugger by showing a simple 2D platform game, similar to Bret Victor’s demo[5]. With this tool, the programmer is able to interact with the program being debugged, and can pause and rewind time. The programmer is also able to add “traces” to any moving elements in the program, such as the game character in the game. This can be done by applying the function `Debug.trace : String -> Element -> Element` to any value of type `Element` in the program being debugged. (The `String` argument is simply a label used to uniquely identify the trace.)

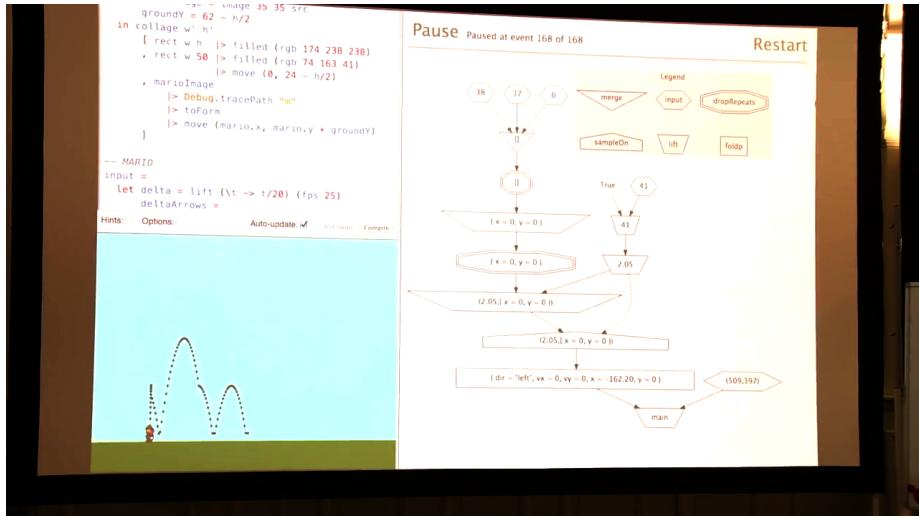


Figure 4: Elm’s time-travelling debugger. Note the traced path of the game character in the bottom-left-hand corner, as well as the visualisation of the program’s signal graph. *06:53*

An important feature of Elm is its treatment of time-varying values, expanded on in section 4.5. All values which change over time are contained in a single “signal graph”. A notable feature of this debugger is its visualisation of the program’s signal graph. As can be seen in figure 4, the data in the signal graph flows from “leaf” nodes, which represent external inputs to the program, to a single node labelled “main”. In this case, the leaf nodes represent the keys being pressed (the numbers in the “input” nodes are the keycodes for the arrow keys and space bar, respectively), as well as the dimensions of the browser window.

I feel that this feature perfectly embodies some of our design principles for programming environments, namely enabling the programmer to *see the state* and *follow the flow*.

The *Elm Reactor*[28], created by Michael James, is a slightly less ambitious development on top of Laszlo Pandý’s time-travelling debugger, featuring a much simpler UI.

The Elm Reactor integrates seamlessly with the existing Elm development

workflow: its functionality is exposed via a simple Elm library called `Debug`, which exposes functions to “watch” and “trace” time-varying values in a program. Importantly, these functions do not change the functionality of the code itself; when invoked anywhere but the Elm Reactor, these functions do not have any effect.

Elm Reactor renders the Elm program as usual, and additionally exposes a control panel which displays the values being “watched”. Additionally, it exposes a “pause” button, and a slider for moving forwards and backwards in time.

Elm Reactor demonstrates that a programming tool can be particularly effective if it contains simple, specific functionality and integrates seamlessly with current tools and workflows.

These tools have been some of the biggest influences in the development of my own tool *Arrowsmith*, along with Light Table, created by Chris Granger.

### 3.3 Light Table

Light Table is an IDE for the Clojure programming language. Clojure is a popular S-expression-based, ie. Lisp-like, functional programming language. S-expressions are an interesting construct, because they make explicit the structure of the program: they are a textual representation of the abstract syntax tree of the program.

My analysis of Light Table will be limited to its original demo video, created by Chris Granger[19]. Its actual implementation, which was open-sourced in January 2014, implements a limited number of the features demonstrated in this video. However, Light Table’s architecture has proven to make the editor easily extendable, and many missing features have been implemented by the open source community as plugins to the editor.

In the video, Granger states that “one of the guiding principles behind Light Table is that documentation should be available wherever and whenever you need it”. [19], 00:10 As such, it features a documentation view, shown beside the code, which automatically shows the documentation of any function or value underneath the cursor in the editing field (see figure 5). This view can also be used to search for documentation for functions outside of the current file, via a search box (figure 6).

However, one of the most interesting features of Light Table is the ability to edit code as a “live document”. This feature has actually been implemented – it is called *InstaREPL* in the released version, and is indeed very useful when writing “real” Clojure code.

Once again, the view is split in two. On the left-hand side, the programmer can edit code as usual. The right-hand side is more interesting: it also displays the code, but any function argument is replaced by the actual value it was invoked

```

(def pressed-keys (atom {}))

(defn on-keypress [ch]
  (swap! pressed-keys assoc (keyword (str ch)) true))

(defn on-keyrelease [ch]
  (swap! pressed-keys assoc (keyword (str ch)) nil))

(defn pressed? [k]
  (get @pressed-keys k))

(defn vel-to-dxy [speed direction]
  [(* speed (Math/cos direction))
   (* speed (Math/sin direction))])

(defn rad->deg [rad]
  (* rad (/ 180 Math/PI)))

(defn deg->rad [deg]
  (* deg (/ Math/PI 180)))

(defn move [[x :> y :> [speed direction] :> vel :as obj]]
  (let [[dx dy] (vel-to-dxy speed (deg->rad direction)))
    [x (+ dx x) y (+ y dy)]))

(defn hit-bounds? [[x :> y :> obj-width :> width obj-height :> height] {widt
  ahtur i=hit f= a=ta

```

keyword  
([name] [ns name])  
Returns a Keyword with the given  
namespace and name. Do not use : in the  
keyword strings, it will be added  
automatically.

Figure 5: Automatic documentation view. Note that the editing cursor is currently on the “keyword” name in the editing field on the left-hand side, and the documentation for that function is shown at the same point on the right-hand side. *00:13*

(let [sign (rand-int 2)]   cur (rand-int upper)   (if (zero? sign)       (* cur -1)       cur))  (defn reflect-with-wobble [cur axis]   (let [wobble (signed-rand 11)         new-angle (case axis                     :x (+ 90 cur)                     :y (- cur)                     :xy (- cur 180))         (+ new-angle wobble))  (defn collide ([{:speed direction} :> vel :as obj] axis)   (let [reflected (assoc obj :vel (speed (reflect-with-wobble direction     (merge reflected (move reflected)))))  (declare end-game)  (defprotocol updateable   (update [this state] "updates the current game object based on the state"))  (defprotocol drawable   (draw [this graphics] "draws the current game object on the given graphics"))  (defrecord ball [x y width height vel]   updateable   (update [this [board board-left :left-paddle right :right-paddle] :>     (if-left (axis (hit-bounds? moved board))       (collide this axis)       (if (or (intersects? moved left) (intersects? moved right))         (collide this :>)	noir.core noir.core/compojure-route noir.core/custom-handler noir.core/custom-handler* noir.core/defpage ([# args]) Add a route to the server whose content is the result of evaluating the body. The function created is passed the params of the request and the destruct param allows you to destructure that meaningfully for use in the body. There are several supported forms: (defpage "/foo/:id" {id :id}) an unnamed route (defpage {:post "/foo/:id"} {id :id}) a route that responds to POST (defpage foo "/foo/:id" {id :id}) a named route (defpage foo {:post "/foo/:id"} {id :id}) The default method is GET. noir.core/defpartial
--	---

Figure 6: Finding documentation with Light Table. *00:35*

with (see figure 7). This enables the programmer to *follow the flow through the code, as per one of our guiding principles.*

```
(defn x [a]
  (* 10 a))

(defn y [b]
  (+ b 5))

(y (x 3))
```

```
(defn x [3]
  (* 10 3))

(defn y [30]
  (+ 30 5))

(y (x 3)) => 35
```

Figure 7: Editing code as a “live document”. The function arguments (defined on the left-hand side) have been replaced by their respective values (on the right-hand side). 01:44

Another interesting concept demonstrated in the video is a way to “break away from the notion that the smallest unit of code is a file. Instead, the smallest unit of code is really a function.” 02.58

Instead of showing a single editor for an entire text file, Light Table shows a separate text editor for each function (see figure 8). Positioning the cursor on a function name within one of these editing fields opens up a new editor beside it, containing the definition of that function.

The screenshot shows a Light Table interface with several function definitions in separate windows:

- gravity**:  

```
(defn gravity [(:keys [vy y] :as me])
  (let [g 0.5
        vy (or vy 0)
        neuvy (- vy g)]
    (if (= vy neuvy 0) :up :down)
    moved (update-in me [:y] + neuvy)))
  (if-let [block (colliding? moved)]
    (let [block-edge (if (= (:x block) (:y block))
                         (+ (:x block) (:z block)) (:r me))
          (assoc me :y block-edge
                :jumping (= dir :up)
                :vy 0))
      (-> moved
        (assoc :vy neuvy))))
```
- jump**:  

```
(defn jump [me]
  (let [speed <= 10]
    (if (and (input? :space)
             (and (not (:jumping me))
                  (zero? (:vy me))))
           (assoc me :vy speed)
           (assoc :jumping true)
           me)))
```
- move**:  

```
(defn move [me]
  (let [speed <= 10]
    (cond
      (input? :left) (- speed)
      (input? :right) speed
      :else 0)
    moved (update-in me [:x] + vx)
    (if (zero? vx)
        me
        (if-let [block (colliding? moved)]
          (let [block-edge (if (< vx 0)
                             (- (:x block) (:z block)) (:r me))
              (assoc me :x block-edge)
              moved)])
          moved))))
```
- update-player**:  

```
(defn update-player [me]
  (let [me]
    (gravity)
    (move)
    (input)
    (reset)
    ))
```
- draw-player**:  

```
(defn draw-player [ctx me]
  (-> ctx
    (set-style (:x me))
    (stroke-style (:y me))
    (stroke-width 2)
    (circle me)
    (stroke)))
```

Figure 8: A separate editor for each function. 03:08

This is in my view an amazing way to discover the code in a large codebase. In order to make sense of a function, the programmer has to understand what each of the definitions in that function mean. Many IDEs feature a “show definition” function, but they tend to replace the entire editing view with the new file (this

is the default in Eclipse, IntelliJ, Xcode, etc.). By showing both the definition and the context in which it is used, the programmer is not forced to mentally switch between the two. Importantly, the definition is opened in a fully-fledged editor – the programmer can change it immediately, as well as continuing to explore “deeper in the call stack”. In the words of the “Learnable Programming” framework, it allows the programmer to immediately *read the vocabulary* of the program.

Programming environments for Lisp-like languages have the advantage that the structure of the program is always explicit, since S-expressions simply represent the tree structure of a program. This allows the programmer to easily and accurately evaluate specific expressions. Clojure, like many Lisps, is dynamically typed. The following project, Lamdu, is an exploration of the kinds of features that are enabled by a powerful type system such as Haskell’s.

### 3.4 Lamdu

Lamdu[18] is an abstract syntax tree editor by Eyal Lotem for a dialect of the Haskell programming language. It features many interesting ideas for editing code. The main idea behind this editor is that “the canonical representation of programs should not be text, but rich data structures: Abstract syntax trees”.

All editing in Lamdu is done on an abstract syntax tree level – Lamdu does not allow the programmer to edit text (other than names, of course). This gives the editor some impressive capabilities, but I argue that this has impeded its use as a “real” programming environment.

As the textual representation of a program can not be edited, Lamdu completely eliminates syntax errors. All changes produce a structurally valid program. This is achieved by use of Haskell’s powerful type system, and a number of clever user interface mechanisms.

I will attempt to explain the most important of these mechanisms by illustrating the process of writing a simple Haskell-style fibonacci number function. Such a function can be expressed neatly in Haskell:

```
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

The programmer can only enter text in a “hole”. A hole is defined by the abstract syntax tree of the program. Thus, entering text at the top level automatically results in a function definition:

```
Exported type: 
Inferred type: 
fib$ =
```

Pressing the = key results in the cursor jumping to the right hand side of the definition, rather than a literal = character being inserted in the code.

When the programmer enters 1, Lamdu automatically infers a type of Int.

```
→ Exported type: 
  Inferred type: Int
fib$ = 1
```

Continuing with the definition, the programmer enters :. The inferred type immediately changes to List Int, and the cursor jumps to the next “hole”.

```
→ Exported type: 
  Inferred type: List Int
fib$ = 1 : (1 : )
```

++ .. :

[]

case

nothing

just

case

empty

non-empty

concat

enumFrom

...

List Int

Since the editor can infer the type of the hole, it gives a number of useful suggestions, all of which result in an expression of type `List Int`, such as concatenation (`++`) or a range of numbers (`..`). Also notice that Lamdu has automatically added parentheseses. Typing a function name constrains the suggestions:

The screenshot shows the Lamdu code editor with the following code:

```
fibs = 1 : (1 : zipWith xs ys)
```

The editor highlights the `zipWith` function in red, indicating it does not fit the inferred type. A tooltip for `zipWith` shows its type: `func {x, y} → List`, where `x` and `y` are type variables. The inferred type is `List Int`.

Notice that the suggestion is highlighted in red. This indicates that the type of the suggested item does not fit the type of the hole. The type of `zipWith` illustrates some of the differences between the language Lamdu operates on and Haskell: All arguments are explicitly named, i.e. all functions take as argument a single record containing the parameters. Thus, the language does not support “currying”, ie. partial application (this concept is described in section 6.1.1). Also, type variables are represented simply as type “holes”.

The screenshot shows the Lamdu code editor with the following code:

```
fibs = 1 : (1 : zipWith λ x . xs fibs)
```

The editor now suggests an anonymous inner function `λ x .` with type `{x Int} → Int`. The inferred type is `List Int`. Below the code, the variables `xs` and `ys` are shown as `tail fibs`.

Entering `fibs` and `tail fibs`, as the two list arguments respectively, constrains the hole to the required type of `List Int`. Lamdu suggests an anonymous inner function (represented by the lambda) of the required type. The finished definition looks like this:

```

Exported type: List Int
fibs = 1 : (1 : zipWith ( $\lambda$  z . z.x + z.y))
           [x Int]
           [y Int]
xs fibs
ys tail fibs

```

Lamdu contains some excellent user interface ideas. The editor continuously offers suggestions, which due to the powerful type system are highly relevant. In most cases, very little text has to be entered – the top suggestions are in many cases exactly what is required.

Type errors are also very intuitive in Lamdu. If a definition contains a type mismatch, relevant nodes are highlighted in red, making it very easy to visually discern the source of an error.

Since the editor works at an AST level, variables can be renamed at any point, automatically updating all their occurrences.

It also features some interesting ideas regarding version control. Traditional version control systems (VCS), such as `git`, work on line-by-line text diffs. This makes it hard to track the actual evolution of a program: even a simple change like renaming a single variable results in a number of disparate one-line changes. It is very difficult to automatically infer the intent of a single change, thus requiring manual annotation in the form of commit messages.

Lamdu automatically stores revision in a custom VCS which records tree diffs, rather than text diffs. This highly reduces the risk of merge conflicts compared to a traditional VCS. Also, it would be much easier to automatically generate valuable commit messages, though I don't know if this has been explored in Lamdu.

The approach followed by Lamdu has some drawbacks however. It works with a custom programming language which differs from Haskell in significant ways, as outlined above. This immediately limits the usefulness of the editor, as existing code can not be edited in it. It is also not clear where, and in what format, code is saved. It appears that Lamdu programs have no simple textual representation, which makes the choice of using Lamdu an “all-or-nothing” proposition.

Editing in Lamdu suffers from the limitations which seem to afflict most AST-based editors: some changes in a program are extremely tedious to perform without breaking the structure of the code temporarily. I will expand on this in section 3.9.

That being said, the navigation through the AST is very well implemented. Lamdu currently only supports keyboard input, though I believe that its current UI would be well suited to a touch-based interface.

An even more ambitious approach for editing Haskell-like programs structurally is Conal Elliot's Tangible Values project.

### 3.5 Tangible Values

Conal Elliot's work on Tangible Values explores very interesting ideas on the representation of values in a pure functional environment. He defines Tangible Values (TVs) as “visual and interactive manifestations of pure values, including functions”[8].

A TV is a composition of a value and its GUI representation, i.e. `type TV a = (Output a, a)`. This value can also be a function.

(The following types are a slightly modified from Conal Elliot's original formulation of the system called “Eros”, and his documentation for the TV library on the Haskell wiki[26].)

I will attempt to illustrate this system by adapting a simple example stated by Conal Elliot. In order to convey the main concepts, I have simplified the datatypes in the following example. In the TV library, these datatypes contain additional information. The types stated here are closer to their original statement in the Eros system from 2007.

The `words :: String -> [String]` function in Haskell splits a sentence into a list of its constituent words, i.e.

```
words "The night Max wore his wolf suit" ==
      ["The", "night", "Max", "wore", "his", "wolf", "suit"]
```

The (nowadays) obvious user interface for a `String` value is a textbox. Functions in TV are represented as inputs and outputs laid out vertically:

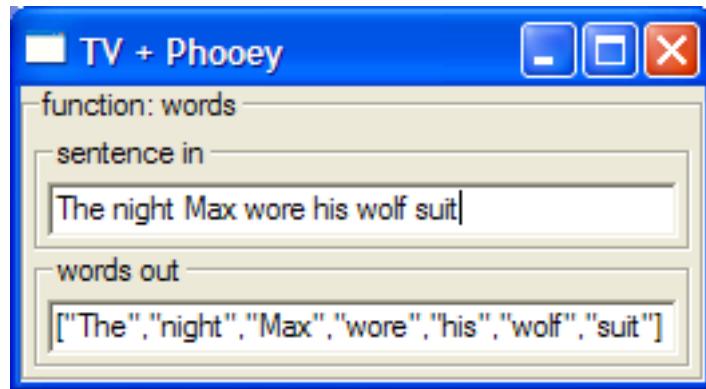


Figure 9: `words` function as TV

An interface of the shape shown above could be represented in the system as follows:

```
gui :: Output (a -> b)
gui = oTitle "function: words" (oLambda (iTitle "sentence in" defaultIn)
                                         (oTitle "words out" defaultOut))
```

The function `oTitle :: String -> Output a -> Output a` labels a given output. Similarly, `iTitle` labels an input. `oLambda :: Input a -> Output b -> Output (a -> b)` composes an input and an output, and arranges them vertically. The `defaultIn` and `defaultOut` functions are defined as the “default” input/output for a given type, viz. a textbox in the case of `String`.

Notice that the above definition only defines the *shape* of the UI. In particular, it describes the UI for any single-argument function (`a -> b`), with added labels. This is a key insight of the Tangible Values project: the type of a user interface corresponds to the type of the value that is being represented by the user interface.

This “generic” UI can then be composed with the actual function which implements the required behaviour on a value level:

```
tangibleWords :: TV (String -> [String])
tangibleWords = tv gui words
```

The `tv :: Output a -> a -> TV a` function turns the given `Output` (defined as `gui` above), and the given value-level function (`words` in this case), into a “Tangible Value”. This can then be run as a GUI program which splits any text entered in the top textbox into its constituent words, which are displayed in the bottom textbox.

Notice that this definition is independent of the actual UI mechanism being used. In Conal Elliot’s implementation, he defines a UI monad which results in the interface seen above. Additionally, he provides a command-line interface which implements the same behaviour (through the built-in Haskell `IO` monad).

Tangible Values are composed by a process he calls “fusion”. In the Eros system, a user can select compatible inputs and outputs, which are then “fused”: the selected input-output pair disappears, and the remains are fused into a single new `TV`.

Fusion is a remarkable interaction method: it is a very intuitive way to express function application and composition.

In figure 10, the program consists of:

- a function which scales an image of a circle by a given real-numbered value (represented by a “slider” `Input`)

- a constant value 1.0 (represented by a textbox Output)

The user selects both the input and the output, which are then fused into a single image with scale 1.0.

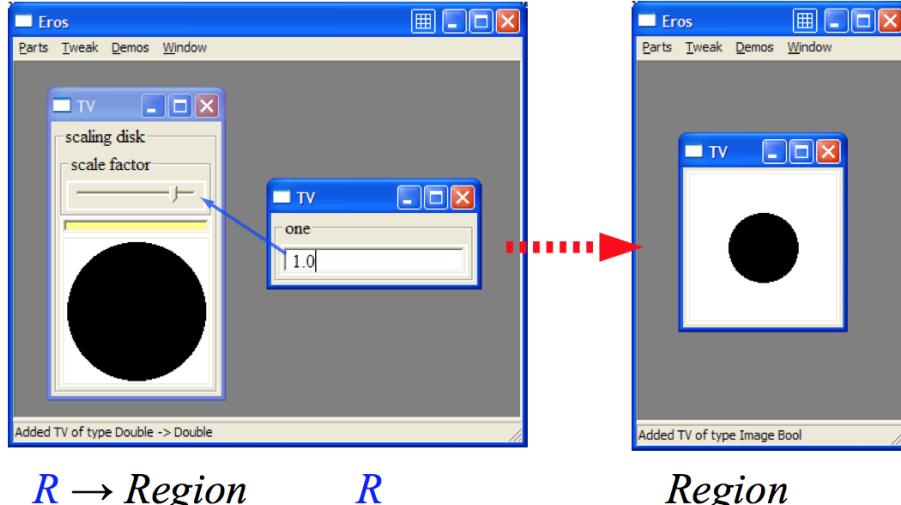


Figure 10: TV fusion as function application

In the original implementation (from 2007), the user had to click on the inputs & outputs separately. An obvious, improvement in today's touchscreen-heavy world would be a drag-and-drop touch interface.

Function composition works similarly: In figure 11, the constant “one” has been replaced by a function which returns the square root of a given number input (once again represented by a slider).

Fusing the output of the square-root function with the input of the scale function results in a TV in which the scale of the circle corresponds to the square root of the given input value.

Notice that the fused “square root” output and the “scale factor” input have disappeared in the resulting TV. This is analogous to how the composition of functions of type  $a \rightarrow b$  and  $b \rightarrow c$  is of type  $a \rightarrow c$ , hiding the “intermediary” type  $b$ .

Tangible Values are in my mind an excellent user interface paradigm for representing functions. However, the composability of the UI components comes at a price in complexity, particular at the type level. As all the values and functions are encapsulated in the TV type, function application, composition etc. have to be “lifted” to that type. Elliot achieves this by using a so-called “deep arrow” abstraction of his own creation, which provides “deep function application”.<sup>[7]</sup>

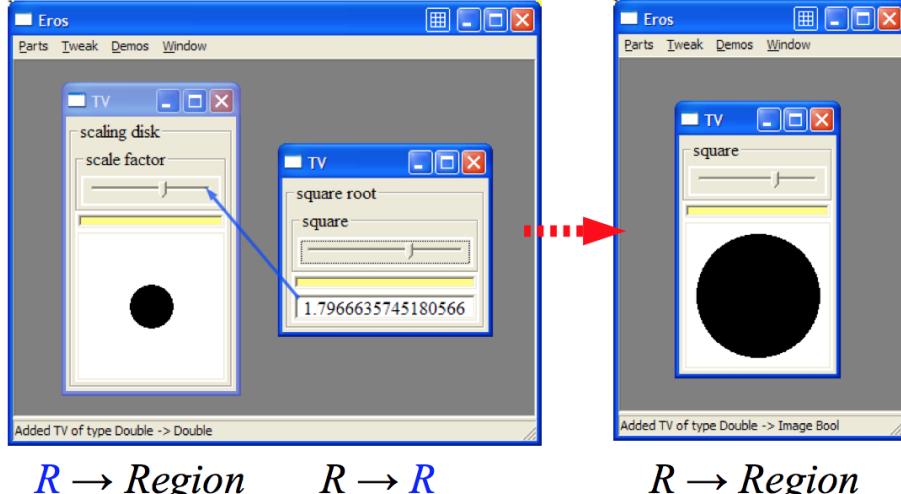


Figure 11: TV fusion as function composition

I am not convinced that this composability is worth this added complexity. In line with my principle of “the programmer should not have to do what the machine can do for her”, this “lifting” would have to be done completely automatically by the compiler in order for it to be acceptable for use. I’m of the opinion that composition at a value level is sufficient, and that the rendering of the UI should be kept as a completely separate stage.

### 3.6 “Blocks”-based graphical programming environments: Scratch and Hopscotch

The idea of representing lines of code as “blocks”, which can be dragged, dropped and rearranged interactively has been widely explored. This is conceptually a very neat approach, since it makes clear the structure of the program. In particular, this approach has been mainly used in educational systems designed for children. Indeed it has not been proven so far whether these ideas can work within the context of “professional” programming.

Scratch[21], developed at MIT, is arguably the best-known such system. The Scratch programming environment features an easy-to-use 2D graphics engine, in which image-based “sprites” can be manipulated using a “turtle graphics” approach. Since it allows for the creation of highly interactive and visual games and animations, it is often used as an introductory programming environment for schoolchildren.

The Scratch programming language is event-driven and imperative: “stacks” of blocks begin with an event (eg. when the green flag is clicked, as above), and

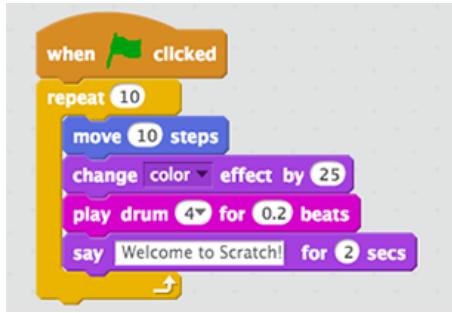


Figure 12: A simple Scratch program.

blocks that are “attached” below it are executed in sequence.

The programmer can choose from a pre-defined set of blocks, which have different colours based on their category (“type”). For example, all operations related to “motion” (of the sprite/turtle “pen”) are blue, control flow operations are yellow, blocks that change the “Looks” are purple.

Control flow is represented in an interesting way: loops (such as the “repeat” block in figure 12) add a second “connection point” at which blocks can be added. The blocks inside the loop are then surrounded by the loop block itself.

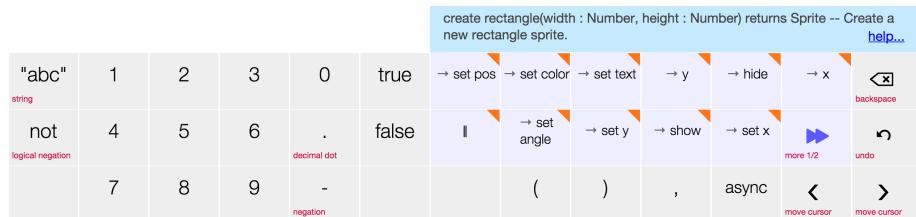
While Scratch definitely brought this style of programming to the (educational) masses, its core idea has been improved upon by some more modern projects, most notably Hopscotch[15]. Hopscotch is an iPad app which allows children to create games and animations similar to Scratch. It improves on Scratch by its use of an intuitive touch interface: blocks can be dragged into place, and complex scenes can be easily laid out.

Blocks are also coloured, and organised into categories. A complaint voiced by one of my friends, an experienced programmer, is indicative of the weakness of this kind of system for “real” programming: having to choose a block from a categorised list is very tedious, especially if the programmer knows what they are trying to achieve. Typing code is a much more direct form of editing a program.

### 3.7 Touch-based programming environments: TouchDevelop

An interesting attempt at creating a tool for “real” programming on touch-based devices is Microsoft’s TouchDevelop[29], a graphical editor for a typed “scripty” language which compiles to JavaScript. This is a web application which in theory can be used on any device, although the user interface is clearly optimised for tablets.

Editing is performed on a line-by-line basis: lines can be selected by tapping, revealing a wide range of editing commands:



The TouchDevelop language features a type system, so the methods and assignments featured in the highlighted buttons are always relevant to the current line of code.

TouchDevelop nicely shows structural errors inline: it also features typed holes.

**foo → create picture(picture)**

**⌚ insert a picture here [TD100]**

**foo ||**

**⌚ the operator needs something after it [TD154]**

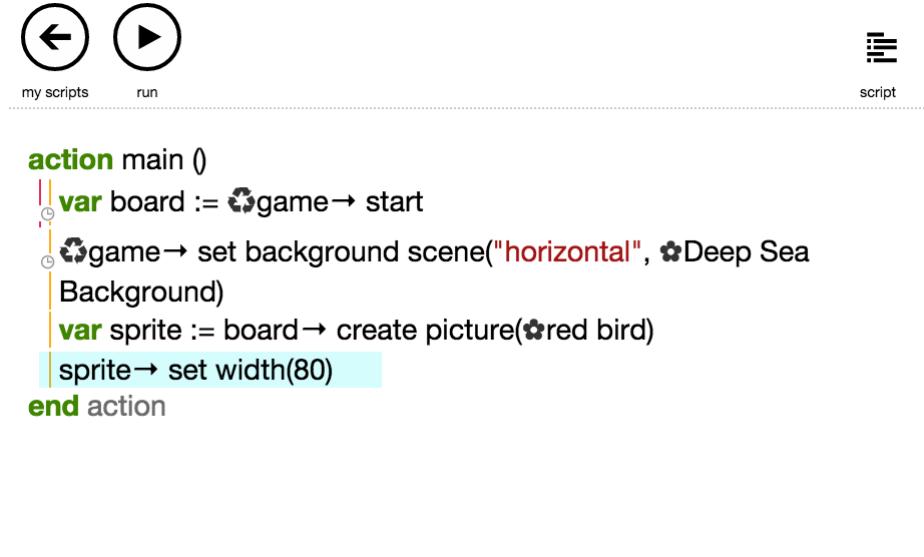
One very interesting feature of TouchDevelop is its different editing modes: The “Beginner” mode features a Scratch-like blocks representation:

```

function main ()
    var board := game → start
    game → set background scene("horizontal", Deep Sea
Background)
    var sprite := board → create picture(red bird)
    sprite → set width(80)
end function

```

The “expert” mode reveals more of the structure of the code:



```
action main ()  
    var board := game→ start  
    game→ set background scene("horizontal", Deep Sea  
    Background)  
    var sprite := board→ create picture(red bird)  
    sprite→ set width(80)  
end action
```

However, other than superficial differences, the modes do not change the editing behaviour.

TouchDevelop also suffers from the problem that none of the written code can be edited in other applications: the language is specific to the editor. However, the written code can be deployed on many platforms, such as mobile phones (using Apache’s Cordova APIs) and Microsoft’s Azure platform.

### 3.8 Notebook environments: IPython Notebook and Gorilla REPL

Notebook programming environments have had wide success in the scientific community due to their ability to execute code fragments within the context of a document which can contain explanatory text and images.

In notebook programming environments, fragments of code are embedded in a document (the “notebook”), which can be edited interactively. From a programmer’s perspective, such environments can thus be seen as an interactive tool for literate programming.

IPython Notebook[17] is probably the most widely used notebook environment. The project’s website describes it as “a web-based interactive computational environment where you can combine code execution, text, mathematics, plots and rich media into a single document”.

IPython Notebook features strong integration with Python’s scientific programming libraries SciPy and NumPy, and has impressive plotting and graphing

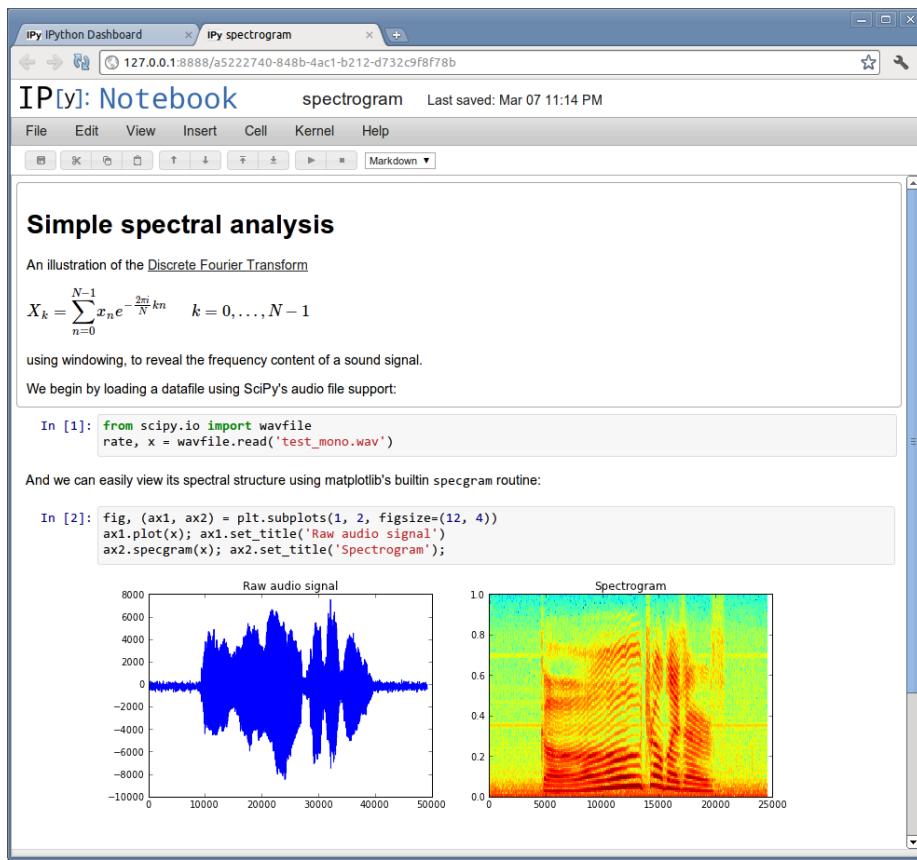


Figure 13: IPython Notebook

capabilities. As such, it is often used to illustrate scientific concepts with live executable code. Entire textbooks have been written in the environment, featuring plots and figures which derive from real code and data, and exercises which can be solved by writing Python code.

IPython Notebook also supports writing code in languages other than Python, by defining a “backend” protocol for other programming languages to interact with the notebook. Notably, the IHaskell[16] project is a Haskell backend for IPython which features some interesting details, such as a `IHaskellDisplay` typeclass, which allows the programmer to define custom rendering logic for Haskell datatypes.

Another interesting project is Gorilla REPL[13], which is a notebook environment for the Clojure programming language. Its user interface is similar to IPython Notebook, but it features some interesting implementation details, particularly its value renderer[12], which allows the programmer to specify custom rendering functions for various types of data.

One of the key design features of Gorilla REPL is that it is “fundamentally value-based”:

In Gorilla, plotting a graph, or showing a table isn’t a side-effect of your code – it’s just a nice way of looking at the value your code produces. You could say that Gorilla is fundamentally value-based. Being strictly value-based like this a limitation, but it’s an empowering limitation: it makes it possible to compose and aggregate rendered objects just like you compose and aggregate Clojure values, to save worksheets with their rendered output intact, and even to manipulate the output of worksheets that you haven’t/can’t run. [12]

This is done by wrapping the value in a data structure which specifies a rendering mechanism for that specific value. This echoes one of the key features of the Tangible Values project, namely that the user interface is simply a consequence of the specific values in a program. This has been a key inspiration for my implementation of “Value Views”, detailed in section 6.3.3.

One of the biggest weaknesses of current notebook environments is that none of the code written inside them can be reused easily. They are intended to replace REPLs (“Read-Eval-Print-Loop”), which traditionally have been command-line environments for writing exploratory “throwaway” code. (In fact, IPython Notebook’s name-sake, IPython, is a widely-used command-line Python REPL.) As such, they don’t allow the programmer to define modules and export definitions. This limits their usefulness to programs which can be meaningfully expressed within a single “module”. While these tools are well suited to exploratory coding, I believe a programming environment in which values can be easily visualised at any point can also be a useful tool for writing and debugging more complex systems and library code.

### 3.9 Conclusion

What unites the above systems is that each provides a novel way of interacting with a program that goes beyond simply editing text files. Some of these provide small, incremental improvements, such as *Elm Reactor*, while some try to entirely redefine the concept of programming, as displayed most forcefully in the *Tangible Values* project.

An essential component of most of these systems is also that they try to minimise the “feedback loop” between the programmer and the program being created. This manifests itself in features which allow the programmer to instantly see the effect of their changes, as well as having an insight into the state and data flow of a program. Arguably the most successful project in this space has been *Light Table*, whose “InstaREPL” is an extremely effective debugging tool. The flow of a value through the program is visible at the level of every single expression which “manipulates” a value.

Another effective insight demonstrated by Light Table (though hardly novel at the time of its creation) is the idea that functions should act as the “unit of editing”: every function should have its own editor, rather than simply being part of a text file with multiple functions. This has been an essential element in the design of Arrowsmith, which will be expanded upon in gory detail in the rest of the report.

A key feature which seems to be universally understood by now is that errors should be either instantly available to the programmer, or in some cases not even possible at all. For example, editors which operate on the abstract syntax tree of a program completely eliminate an entire class of programming errors: the syntax error.

Conceptually, AST editors are very appealing: having to “mentally parse” text into a (mental) abstract program structure, only to then have to “mentally serialise” the newly manipulated program structure is an obvious deficiency which must have crossed every serious programmer’s mind at some point in their career.

However, at the current state of the art plain text is still the programming interface with the highest “bandwidth”. We have at our disposal incredibly efficient tools for manipulating text, and have developed a vast number of solutions to make the structure of programs clear from their plain text representation, most notably syntax highlighting.

Attempts to move away from editing text have largely been unsatisfactory. I believe that this is largely due to the insistence on *replacing* text editing with structural editing – a clear case of “throwing out the baby with the bathwater”. A more successful approach, I argue, is instead to *hide* as much text as possible from the programmer, and to only selectively expose text when necessary. Importantly, the programmer should always be able to “fall back” to plain text editing if the structural tools fail her.

As the above-mentioned structural editing tools have proven, this is likely to happen: In terms of the transformations available to the programmer, any interactive interface (such as a *Scratch*-like drag & drop interface) will always be more restrictive than plain text. For example, say the programmer wants to change the expression  $(1 + 2) * 3$  to  $1 + (2 * 3)$ . This is a trivial change in text, but most structural editors struggle with this, since the change completely changes the structure of the expression. However, with these restrictions comes a certain freedom: when manipulating programs through such an interface (eg. by dragging blocks), the programmer can be certain that she has made no syntax errors in the process.

An important concept to note is that interactive structural interfaces and plain text code are merely different “views” of the same underlying structure: the abstract syntax tree of the program.

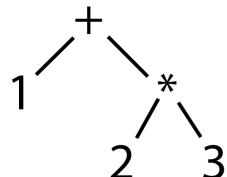
The following representations can all correspond to the same abstract syntax tree:

`1 + 2 * 3`

`(+ 1 (* 2 3))`

`1 2 3 * +`

Add 1 (Multiply 2 3)



Thus, in an ideal world we should be able to switch freely between any such representations. In reality, a single textual representation is enough, namely that prescribed by the syntax of our chosen programming language. In certain cases, several useful interfaces might exist for editing a certain piece of code. The programmer should be able to choose easily between any of these.

The most serious implication of replacing text with a purely structural editor is that these editors tend to interact badly with other existing environments and workflows. For example, none of the structural editing projects mentioned

exposes a human-readable textual form of the program. Scratch programs manifest themselves as complex JSON files<sup>[23]</sup>, Lamdu and Hopscotch provide no access at all to programs outside the environments<sup>2</sup>, and TouchDevelop files can at most be “reverse-engineered” from compiled JavaScript code – hardly a sustainable editing workflow.

Practically speaking, programmers working in a team on a project written in these environments is forced to use this exact editing environment. This is an unlikely prospect, as professional programmers tend to have invested serious effort into setting up their current editing environments, and tend to over-estimate the “hit in productivity” that changing to a new environment entails.

Lamdu goes so far as to implement its own version control system to handle its specific binary format. This approach disregards the many thousands of hours of work that have gone into “real world” version control systems. The reality is that the vast majority of programming work is eventually committed to either a company-internal instance of `git`/`mercurial`/`svn`/etc. or to open source code hosting sites such as *GitHub* or *BitBucket*. It is simply not possible to perform “normal” programming workflows with current structural editing tools.

This is to me the most likely explanation for why these tools have all failed as serious programming tools.

This project is an attempt at gathering together the “best of both worlds” of structural editing and text editing: a tool which exposes the meaningful structure of the code to the programmer, but which plays nicely with the rest of the programming world. In particular, it should be possible to collaborate on a project with programmers who use any other editor, whether it is `emacs`, `vim`, `ed` or butterflies. The design and implementation of this project is detailed in the following sections.

---

<sup>2</sup>Hopscotch at least has the excuse of being an iPad app, where users do not have access to the file system.

## 4 Functional Reactive Programming with Elm

As noted in section 2.2, a programming system consists on an *environment* and a *language*. Many programming editors exist which support multiple languages. This limits their power to the “lowest common denominator” of language features: apart from simple features such as syntax highlighting, most such editors do not have any powerful features for programming other than efficient text manipulation.

Limiting the editor to a single language allows the editor to fully leverage the unique features of that language. This is what is meant when we speak of *integrated* development environments, or IDEs. IDEs are able to provide editing features which rely on the specific features of a single language.

Some of the most popular IDEs in use today are Eclipse and IntelliJ for Java, Microsoft Visual Studio for C#, and Apple Xcode, for Objective-C. These IDEs have features that are tailored specifically towards imperative, object-oriented languages: they feature instant error detection, syntax-directed auto-completion, and step-by-step debuggers, among many other things.

No comparable tools currently exist for functional programming languages. Since such languages have an entirely different programming model to imperative languages, many of the advanced features of current IDEs do not make sense for such languages. In this section, I aim to outline some of the fundamental properties of functional programming languages, and the consequences this has for the development of programming tools. An understanding of these properties is essential for appreciating the reasoning behind the ideas presented in the following sections.

The programming language used for the code examples in this section is *Elm*[9]. Elm is a “functional reactive” programming language with a special emphasis on creating user-interactive programs. Its basic features and type system resemble *Haskell*, but its treatment of effectful code is very different from Haskell’s. Readers familiar with Haskell may want to skip to section 4.5. The unique properties of Elm’s programming model result in a way of structuring interactive programs that will be unfamiliar to most.

### 4.1 Definitions: Immutable Values, Functions and Types

The most fundamental feature of a pure functional programming language is that all values are *immutable*. This means that, once defined, values can never be changed. This is in contrast to imperative programming languages, where variables can be “destructively” updated: that is, variables can take on different values at different points in time.

Take the following example program:

```

answer : Int
answer = 42

otherAnswer : Int
otherAnswer = 1337

funky : Bool
funky = True

greeting : String
greeting = "Hello!"

betterAnswer : Int
betterAnswer = max answer otherAnswer

max : Int -> Int -> Int
max first second =
    if first > second then first else second

```

This program contains five definitions. For each definition, the first line is a *type declaration*. The next line specifies the value.

The first two definitions are simple: we have simply given names to the integers 42 and 1337.

The next two definitions demonstrate further types: booleans and strings.

The last two definition of are more interesting:

`betterAnswer` is defined as the result of applying the `max` function to `answer` and `otherAnswer`. (Note that unlike most programming languages, Elm does not require parentheses around the arguments – `max answer otherAnswer` would be written as `max(answer, otherAnswer)` in many other languages.)

The `max` function is defined below: it simply returns the maximum of two numbers. Notice the type of `max`: `Int -> Int -> Int`. In function types, each of the “argument types” are separated by an arrow. The last type is the “return type”. Thus, this type can be interpreted as meaning “`max` takes two integers and returns an integer”. An alternative, more accurate interpretation would be: “`max` takes an integer, and returns a function, which takes an integer and returns an integer”.

This is illustrated by the following definition:

```

overNineThousand : Int -> Int
overNineThousand = max 9000

```

This is an example of *partial application*: applying a single `Int` to a function of type `Int -> Int -> Int` results in the type `Int -> Int`. This is equivalent to writing `overNineThousand x = max 9000 x`, which makes it clear that `overNineThousand` is indeed a function taking one argument. Since this is already specified in the type, the argument name can be omitted.<sup>3</sup>

This already highlights an important consequence for the design of a useful programming environment: types are essential to the understanding of a functional program – indeed, in some cases the types reveal more about the intent of the program than the code itself.

## 4.2 Evaluation: Purity and Referential Transparency

Notice that the above program “does” nothing – it is simply a list of definitions. In order to get the program to perform the computations that we have specified, we need to *evaluate* them.

The easiest way to do this is to use a tool called a *REPL* (“read-eval-print loop”). This is simply an interactive command-line console which *reads* an expression typed by the programmer, *evaluates* it, *prints* out the result to the screen, and then lets the programmer type another expression (*loop*).

```
> answer  
42
```

In this example, the programmer has typed `answer` and got the result `42`, which is what we defined in our program.

A more interesting example is `betterAnswer`. (Recall that we have defined this as `max answer otherAnswer`). Typing this into the REPL gives us:

```
> betterAnswer  
1337
```

In order to understand this result, let us evaluate this definition “by hand”. This is a simple algebraic task: If we encounter a name we have defined, we replace it by its right-hand side (*RHS*). We continue replacing names by their definition until we have an expression composed of *functions* and *values* (steps 1-3). We then *apply* the functions to the values until we are left with the final result (steps 4-6).<sup>4</sup>

---

<sup>3</sup>Omitting the names for arguments is sometimes called “point-free style”.

<sup>4</sup>Notice that the arguments to the function are evaluated before the function itself. This is called “strict” evaluation. A language with “lazy” evaluation would evaluate the function before evaluating the arguments. Elm features strict evaluation, while Haskell (for example) features lazy evaluation.

```

betterAnswer
-- 1. RHS of `betterAnswer` -->
== max answer otherAnswer
-- 2. RHS of `answer`, RHS of `otherAnswer` -->
== max 42 1337
-- 3. RHS of `max` -->
== (\first second -> if first > second then first else second) 42 1337
-- 4. function application -->
== (\second -> if 42 > second then 42 else second) 1337
-- 5. function application -->
== if 42 > 1337 then 42 else 1337
-- 6. if-expression -->
== 1337

```

An important point to note is that each of these expressions are equivalent: the programmer could type the expressions on any of these lines into a REPL, and would receive the answer 1337.

The programmer can thus give a name to any expression, and replace all occurrences of that expression by its name, without changing the meaning of the program. This is called *referential transparency*.

Referential transparency is enabled by the fact that functions are *pure* – a function is guaranteed to always evaluate to the same value given the same inputs – and that values are *immutable* – once defined, a value will never change.

The biggest implication of this for a programming environment is that it does not matter *when* a definition is evaluated – since its value is guaranteed to always be the same, the programmer can choose to evaluate any part of a program at any time. This opens up some interesting possibilities for programming tools, as described from section 5 onwards.

### 4.3 Collections and Higher-Order Functions: `map`, `filter`, `fold`

In the previous section, we defined values and functions that operated on simple types only. Any non-trivial program is likely to deal with collections of values. The simplest such collection is a `List`. A list can be defined as follows:

```

excellentNumbers : List Int
excellentNumbers =
    [42, 1337, 365, 24, 1993, 2015]

```

An important feature of `Lists`, and indeed any collection in Elm, is that each element in the list has to be of the same type (in this case, `Int`).

In an imperative language, performing operations on the elements of a collection is done with constructs such as the `for`-loop. This construct does not exist in functional programming – instead, processing of collections is usually performed with *higher-order functions*. A higher-order function is any function which takes another function as an argument. I will detail the most common higher-order functions used for collections, as knowledge of these is essential for understanding Elm’s treatment of time-varying values.

The easiest higher-order function is called `map`. It takes any single-argument function and a list, and returns a list with the function applied to each of the elements individually.

```
map : (a -> b) -> List a -> List b
```

All the previous types we have introduced were concrete types – they are written Capitalised (`Int`, `String`, etc.). This type definition contains *type variables*, written lower-case (`a` and `b`). A type variable can be filled with any concrete type. A single type variable can however only be filled with a single type. We will illustrate this with an example.

Our definition `excellentNumbers` has the type `List Int`. Let’s say we want to transform each of these numbers to a string, for example for displaying the number on screen. This can be done using the function `toString : a -> String`. This converts any value (`a`) to a `String`.

Typing the following into a REPL gives us a list, with each of the values now converted to strings:

```
> map toString excellentNumbers
["42", "1337", "365", "24", "1993", "2015"]
```

The type of this expression is `List String` – the type variable `a` is now constrained to `Int`, and the type variable `b` is constrained to `String`.

Partial application is very useful in conjunction with higher-order functions. Take for example the function `always : a -> b -> a`, which given two arguments always returns the first argument<sup>5</sup>.

In the following example we partially apply this function, giving us a list of strings, all containing “hello”. We can also partially apply arithmetic operations: the expression `(+ 2)` has the type `Int -> Int`, so mapping this over a `List Int` gives us a `List Int`.

```
> map (always "hello") excellentNumbers
["hello", "hello", "hello", "hello", "hello", "hello"]
> map (+ 2) excellentNumbers
[44, 1339, 367, 26, 1995, 2017]
```

---

<sup>5</sup>You may recognise this as `const` from Haskell, or `K` if you’re a skier.

Notice that the resulting list always has the same number of elements as the input list.

Another very common higher-order function for collections is `filter`. This allows us to select certain elements from a collection.

```
filter : (a -> Bool) -> List a -> List a
```

`filter` takes a function returning a `Bool` (a “predicate”), and a list, and returns a list of the same type.

For example, if we define the function `even : Int -> Bool`, we get a list of just the even numbers:

```
> filter even excellentNumbers
[42, 24]
```

A slightly more complex higher-order function is `fold` (also called `reduce` in some languages)<sup>6</sup>.

```
foldl : (a -> b -> b) -> b -> List a -> b
```

This allows us to combine the values from a collection into a single value. It takes a function with two arguments (a “reducer”), an *initial value*, and a list. For each element of the list, the result is built up by evaluating the “reducer” function with the value emitted by folding the previous elements of the list.

For example, to sum all the elements of a list, we fold `+` : `Int -> Int -> Int` over the list:

```
> foldl (+) 0 excellentNumbers
5776
```

In the following example, we turn each number in the list into a string, and then concatenate the resulting strings with the string “numbers:”. (`++ : String -> String -> String` is the concatenation operator.)

```
> foldl (++) "numbers: " (map toString excellentNumbers)
"numbers: 4213373652419932015"
```

These three higher-order functions are immensely powerful – nearly all operations on collections can be expressed using these. We will explore some editing interactions that are enabled by these functions in section 5.

---

<sup>6</sup>The function shown in the example, `foldl` specifies a *left fold*, which consumes a list from beginning to end. The *right fold* function is called `foldr`, and has the same type in Elm.

## 4.4 Structured Values: Tuples, Records and Sum Types

In addition to simple values and collections, Elm also features structured datatypes. These allow grouping related values into a single combined value.

Tuples are the simplest structured value: they are simply a grouping of related values. Examples of tuples:

```
(2, 4) : (Int, Int)
("hello!", True, 42) : (String, Bool, Int)
```

Notice that, as opposed to a list, the element types can be different, and that a tuple always has a fixed size.

Records are essentially tuples with named fields, and are similar to objects in object-oriented programming.

Records can be defined as follows:

```
dublin = {
    name = "Dublin",
    country = "Ireland",
    population = 1110627,
    capital = True
}
```

Fields in a record can be updated as follows:

```
asGaeilge =
    { dublin | name <- "Baile Átha Cliath", country <- "Éire" }
```

Note that since all values are immutable, the values in the original definition remain unchanged.

Record fields can be accessed using the convenient notation familiar from object-oriented programming:

```
> dublin.country
"Ireland"
```

Combining this notation with higher-order functions is very expressive:

```
> map .name [dublin, asGaeilge]
["Dublin", "Baile Átha Cliath"]
```

The type of the record looks very similar to the record itself:

```
city : { name : String, country : String, population : Int, capital : Bool }
```

For big records, this can quickly get unwieldy – we can thus define a *type alias*:

```
type alias City = {
    name : String,
    country : String,
    population : Int,
    capital : Bool
}
```

Another very important datatype is a *sum type*. This is used to represent a value that can be one of several types. Sum types are similar to *unions* in C/C++ and *enums* in Java.<sup>7</sup>

For example, we can define the following types:

```
type ReportState = Writing | Procrastinating
type Bool = True | False
```

Just as a boolean value can only be `True` or `False`, but never both at the same time, the `ReportState` can only either be `Writing` or `Procrastinating`.

Sum types can also hold data:

```
type Contact = Email String | PhoneNumber Int
type Maybe a = Nothing | Just a
```

Notice the type variable `a`: `Maybe` can hold any type. For example, the expression `Just "Harry"` has type `Maybe String`.

Sum types can also be *recursive*. Thus we can represent more complex data-structures such as trees:

```
type Tree a
  = Leaf a
  | Branch (Tree a) (Tree a)
```

Sum types are usually dealt with in code by using the *case* statement:

---

<sup>7</sup>Strictly speaking, sum types are more general than Java enums, since every choice can hold arbitrary data – in Java, every enum field holds the same type of data.

```

encouragement : ReportState -> String
encouragement state =
  case state of
    Writing -> "Good job!"
    Procrastinating -> "Keep writing!"

```

Records and sum types are widely used in any non-trivial Elm program, particularly when creating interactive applications: in most Elm applications, they are used to represent the possible actions that the user can perform.

## 4.5 Dealing with Time: Functional Reactive Programming

Elm belongs to a class of functional programming languages called *functional reactive programming languages*. Functional reactive programming languages can be seen as the subset of functional programming languages which contain an abstraction for time-varying values.

In Elm, values that change over time are referred to as `Signals`. For example, the position of the mouse on the screen is given the following name in Elm:

```
Mouse.position : Signal (Int, Int)
```

According to its type, `Mouse.position` is thus a pair of coordinates ((`Int`, `Int`)) that change over time (`Signal`). Rather than referring to the mouse position at a specific point in time, this value refers to *all* mouse positions that the program encounters during its runtime.

A really useful way to think about time-varying values in a functional reactive language is that time is just another collection: one can `map`, `filter` and `fold` time-varying values in the same way that one can perform these operations on a list.

The type signature of `map` for signals looks like this:

```
map : (a -> b) -> Signal a -> Signal b
```

Recall that `map` transforms every value in a collection. Similarly, you can transform every value over time: the signal `screenSides` defined below updates every time the mouse position updates, and tells us whether the mouse is on the left side or the right side of the screen at any point in time.

```

type Side = Left | Right

side : (Int, Int) -> Side

```

```

side (x, y) =
  if x < 500 then Left else Right

screenSides : Signal Side
screenSides =
  map side Mouse.position

```

Notice that the function `side` is a pure function: it takes a single screen position and returns a `Side`, which we defined just above.

We can also create *past-dependent* signals using `foldp`. Its type is similar to `foldl` for lists.

```
foldp : (a -> state -> state) -> state -> Signal a -> Signal state
```

The name given to the type variable `state` in the official Elm documentation hints at the purpose of `foldp`: In fact, `foldp` is the only way to maintain state in an Elm program.

In the following example, the signal `clickCount` updates every time the mouse is clicked. The value of the signal at any given time is the total number of clicks that have occurred since the start of the program.

```

clickCount : Signal Int
clickCount =
  foldp (\click total -> total + 1) 0 Mouse.clicks

```

## 4.6 Putting It All Together: The Elm Architecture

All of the previously mentioned features combine into a neat structure for creating interactive applications in Elm. The architecture documented here is taken from a document called “The Elm Architecture”[27], written by the creator of Elm, Evan Czaplicki.

I will illustrate the architecture by developing a simple “counter” application. This application displays a “count”, and lets the user increment or decrement the count using buttons.

An Elm program can be cleanly divided into three separate parts: *model*, *update* and *view*.

A value of type `Model` specifies the state of the program at one specific point in time. It is usually defined as a simple record.

For our simple “counter” application, we define a record which contains one field: the integer `count`. We also specify the initial model: this is the program state at the start of the program.

```

type alias Model =
    { count : Int }

initialModel : Model
initialModel =
    { count = 0 }

```

The *update* part of the architecture is composed of three things:

- a sum type `Action` which specifies the possible actions available,
- an `update` function, which applies a certain update to the model, and
- a “mailbox” called `actions`.

In our application, we have two possible actions: incrementing or decrementing the counter. We also add an action type which represents “doing nothing”, `NoOp`. In the `update` function, we perform a case analysis on a single action: if the action is `Increment`, we add 1 to the `count` field of the model, and similarly for `Decrement`. In case of a `NoOp`, the model is returned unchanged.

A “mailbox” consists of of an *address* and a *signal*. Actions can be “sent to” a mailbox’s *address*. The mailbox’s *signal* contains all the actions that have been received by the mailbox. This will used in the `view` function to handle user input: clicking on a button results in a *message* with a specific action being sent to the mailbox’s address.

```

type Action =
    NoOp | Increment | Decrement

update : Action -> Model -> Model
update action model =
    case action of
        NoOp ->
            model
        Increment ->
            { model | count <- model.count + 1 }
        Decrement ->
            { model | count <- model.count - 1 }

actions : Mailbox Action
actions =
    Signal.mailbox NoOp

```

The `view` is a function which transforms a specific model into an `Element`.

`Element` is Elm’s type for creating graphics. Elm features a simple API for laying out user interface components on the screen: here we simply state that

we want to display the count, a “+” button, and a “-” button right beside one another.

```
view : Model -> Element
view model =
    flow right [
        show model.count,
        button (Signal.message actions.address Increment) "+",
        button (Signal.message actions.address Decrement) "-"
    ]
```

Notice the first arguments to both of the `button` functions: these specify the `Action` being sent to the `actions` mailbox. Clicking the button labelled “+” results in an `Increment` action, and similarly for the button labelled “-”.

The crucial step for putting these parts together lies in the `model` definition:

```
model : Signal Model
model =
    Signal.foldp update initialModel actions.signal
```

This definition specifies the model at all possible points in the duration of the program. Recall the definition of `foldp`:

- the initial value of the model is `initialModel`.
- Any time an action is sent to the mailbox (by clicking a button), the `actions` signal updates.
- When the signal updates, the `update` function is called with the previous state, and the current action.

The final part of the puzzle is our application’s entry point, `main`: Any time the model changes, we call the `view` function which renders an `Element` based on the current model. We can achieve this by mapping the `view` function over the `model` signal:

```
main : Signal Element
main =
    Signal.map view model
```

The Elm runtime renders the resulting `Element` on screen.

The end result is not beautiful, but it works: the user can click each of the buttons, and the counter changes correspondingly, see figure 14.

This way of structuring interactive programs is quite extraordinary. The entire application state is expressed as a stream of pure functions acting on immutable



Figure 14: The “counter” application created using the Elm Architecture

data. The `view` is simply a pure function which renders a single static view. The only “impure” definition in the entire program is the “mailbox”: all possible user actions are contained in this one single definition, and the rest of the program updates in response to its single signal of actions.

It is this architecture which enables the “time travelling” demonstrated by Laszlo Pandy in section 3.2. In his “time-travelling debugger”, he saves the contents of the `actions` signal. When any of the code changes, he simply “replays” these saved actions. Since all other definitions are pure functions acting on immutable state, he is guaranteed not to overwrite or corrupt any previous states of the program.

This programming model enables some truly novel ways of developing and debugging programs. In the following section, I aim to outline some user interface ideas which leverage some of Elm’s unique features.

## 5 Design

Using Elm's unique programming model as inspiration, I created several user interface prototypes which aim to embody some of the principles outlined in section 2. Most importantly, my aim is to create interactions which give the programmer an *immediate connection* to the program being developed, as well as ensuring that the programmer is not forced to do anything the machine could do for her.

I will first detail individual components, and will then go on to show how these components working together can enable new ways of programming in a functional programming language such as Elm. For each component, I will map out what can be gained by moving away from plain text, as well as taking a look at the drawbacks of these interfaces.

### 5.1 Types

One of the main features of Elm (and Haskell) which has not been explored to its full potential from a user interface perspective is its powerful type system. Types are one of the most powerful tools for understanding an unknown program. Indeed, alongside names and documentation, they are the only form of specification which is guaranteed to be up to date with the implementation. Documentation is notoriously either non-existent or outdated, and this is a problem that only a rigorous software development process can solve. With types, this rigour is enforced at compile-time.

Syntax highlighting has proved to be one of the most useful tools for reducing the cognitive load of editing text on the programmer. The programmer can instantly see whether keywords are spelt correctly, or that she is using the correct syntax.

By colour-coding types, we can help the programmer further: at an immediate glance, the programmer knows the type of each definition, and is able to visually discern which definitions and values might be related.



Figure 15: Colour-coded simple value types. (Note that in Elm, `String` is an opaque type as opposed to a list of `Chars`, as in Haskell.)

Concrete value types can be represented quite simply: each concrete type is assigned a colour (figure 15). I will refer to these colour-coded components as a *type tags*, or simply *tags*.



Figure 16: Parameterised types.

This is quite obvious; more interesting are parameterised types. These are used to create “generic” datatypes. The most common usecase for these in Elm is for collections, as well as for time-varying values (`Signal`). In the case of collections, the type parameter specifies the element type. Multiple type parameters are possible: `Dict`, a key-value map, has two parameter types – a type for the key, and one for the value. The way these are represented is by colour-coding the collection type with one colour, and showing the colour-coded parameter type as a “cell” inside the type tag, as can be seen in figure 16.

Function types are textually represented as “arrows” between two types, for example `String -> Bool`. This can be represented quite naturally by placing the tags beside one another, as seen in figure 17.

Notice that this representation is also suitable for displaying nested function types, for higher-order functions. For example, the last type in figure 17 corresponds to `(a -> Bool) -> a -> Signal a -> Signal a` (the type for `Signal.filter`). The first argument in this type is a function, and is represented as a “cell” inside a grey tag.

## 5.2 Definitions: Functions & Values

In the functional programming paradigm, values can be seen as functions with no arguments. It is thus useful to speak of “definitions”: these can be either function definitions, or constant values, or values which are the result of certain function applications.

One of the most compelling aspects of Chris Granger’s *Light Table* demo is the idea that we should be editing programs at the level of *functions*, rather than *files*. In the demo, this is achieved by allowing the programmer to navigate directly to a function in a given file, and opening an “editing bubble” containing only that function. This “editing bubble” contains a traditional code editor – the programmer is able to edit text as expected. I like this approach because it focuses the editing task on a “mind-sized” piece of the code: a single function.

Taking advantage of Elm’s powerful type system, we can enhance this idea (figure 18). Every function has its own editing field, as in Light Table. The type of each definition is displayed in the editing field. This editing field can also be hidden, by tapping (or clicking) on the name of the definition. This allows the

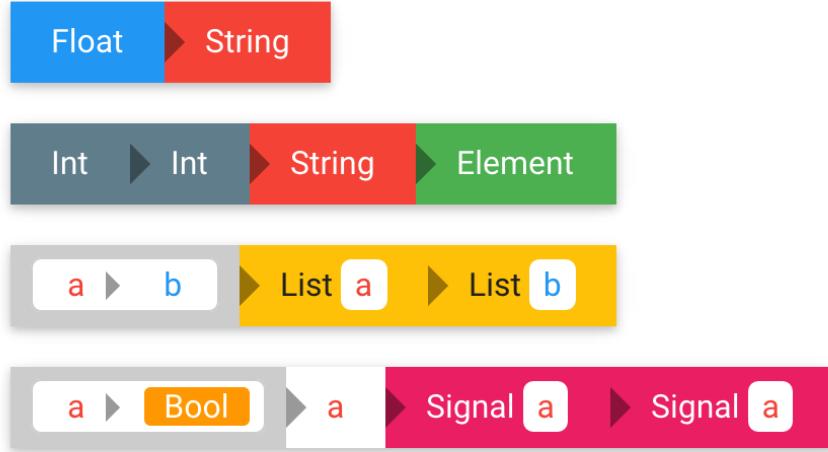


Figure 17: Function types.

programmer to hide code which is irrelevant to the current task at hand. An empty editing field is shown below the definitions, which allows the programmer to enter new code.

### 5.3 Function Composition via Drag & Drop

Imagine the programmer now wants to test her `clock` function in the example shown in figure 18. Note that `clock` has a type of `Float -> Element`, and she has previously defined a value of type `Float`, namely `goodTime`. Thus she could simply apply the value to the function to receive a value of type `Element`. Instead of typing this expression, she should be able to just drag the value onto the function, or vice-versa, drag the function onto the value (figure 19).

When she starts dragging the value, the editor gives her visual feedback as to where she can drop the value by changing the colours of the types: All incompatible types are shown in grey, while compatible types maintain their colour.

The editor infers a relevant name for the resulting expression, which the programmer can obviously change immediately.

This interaction by itself is not very interesting; applying simple values to functions is a trivial task, which is just as easily performed by typing the relevant expression. However, we can extend this interaction to also support collections.

In figure 20, the programmer has a value of type `List Float`, and drags it onto

The image shows a code editor interface with two separate sections for different definitions.

**Top Definition:**

- Label: `main`
- Type: `Signal Element`
- Code:

```
main =  
    Signal.map clock (every second)
```

**Bottom Definition:**

- Label: `goodTime`
- Type: `Float`
- Label: `clock`
- Type: `Float`
- Result Type: `Element`
- Code:

```
clock t =  
    collage 400 400  
        [ filled lightGrey (ngon 12 110)  
        , outlined (solid grey) (ngon 12 110)  
        , hand orange 100 t  
        , hand charcoal 100 (t/60)  
        , hand charcoal 60 (t/720)  
    ]
```

Figure 18: A separate editor per definition. The type of each definition is shown alongside the code. The `goodTime` definition is collapsed. An empty editing field allows the programmer to enter more code.

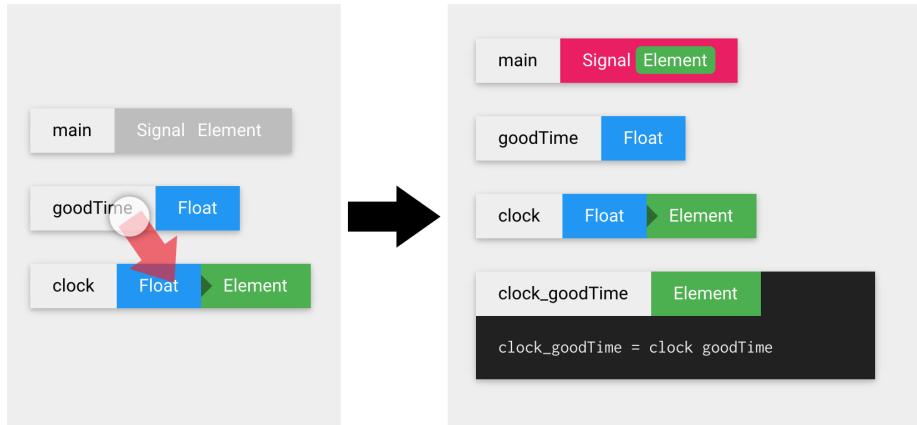


Figure 19: Function composition via drag & drop. The position of the finger/pointer is shown as a white dot. The programmer is dragging along the path outlined by the red arrow. The resulting UI change, with a newly-created definition, is shown on the right.

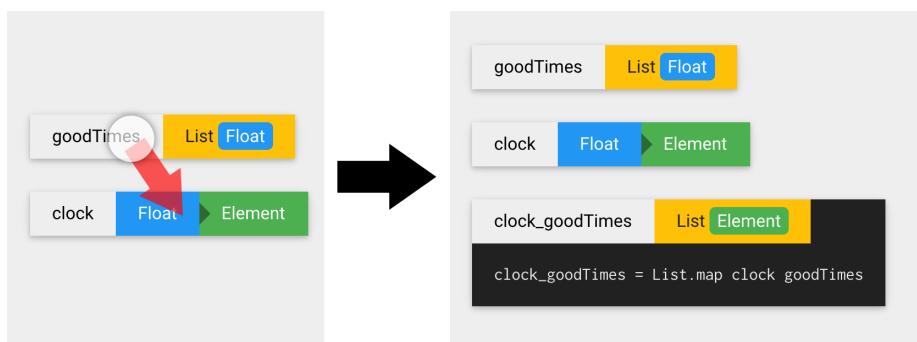


Figure 20: Mapping over a list via drag & drop.

the function `clock : Float -> Element`. The obvious interpretation of this is that she wants to apply the function `clock` to each of the elements in the list. This can be achieved with the `map` function. The resulting code is thus `List.map clock goodTimes`.

This is quite powerful, as this can also be used on `Signals`, ie. time-varying values, as well as any other collection which implements `map`. A common programming workflow in Elm is to create a view for a single constant state, and to then `map` the `view` function over a dynamic state which changes over time based on user input.

## 5.4 Evaluating Definitions

In the previous section, we only focused on editing code. However, in order to have a truly “immediate connection” to the program, the programmer should be able to evaluate any parts of the program instantly. This is where Elm’s programming model shines: because all definitions are pure, any part of the program can be evaluated at any point without affecting the rest of the program.

This is exposed very simply to the programmer: we add a “play” button to any definition that can be evaluated, as in figure 21. Tapping/clicking on this button shows the evaluated value directly underneath the definition.

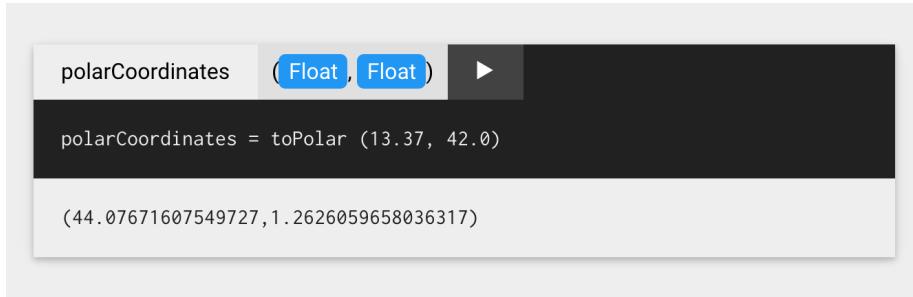


Figure 21: Evaluating definitions using a “play” button. The evaluated value is displayed underneath the definition.

For most types, this value will be displayed using a simple string representation. Some types have more useful representations, as shown in figure 22: `Elements`, which are Elm’s basic type for rendering graphics, can simply be rendered in the value field. `Colors` can be displayed as a coloured rectangle. `Signals` can simply display their current value.

As presented so far, these components are non-interactive: they are simply a way to make values more immediate to the programmer. Our approach can be extended to allow interaction with these values.

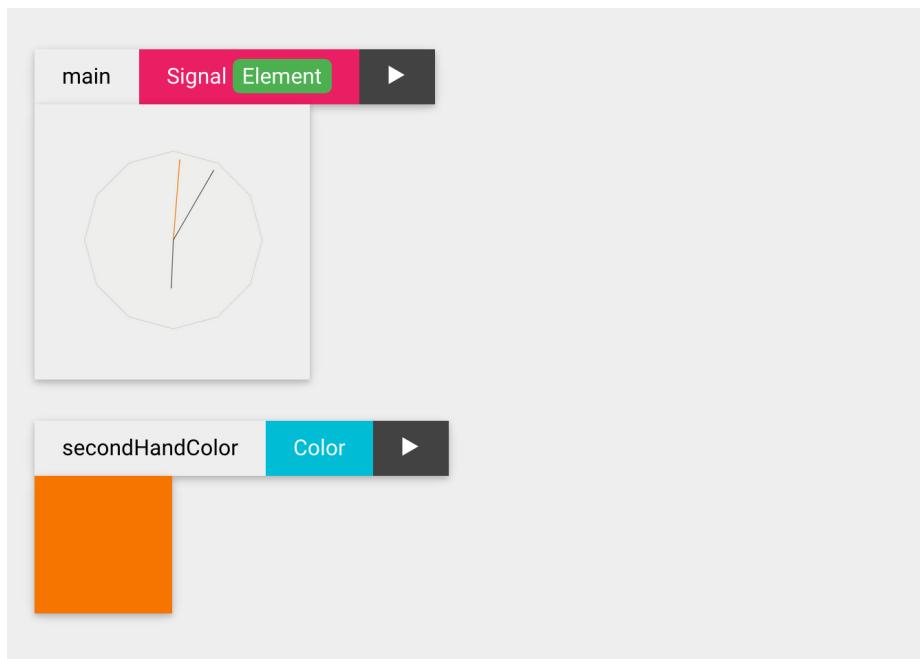


Figure 22: Rendering values of type `Element` and `Color` inline. Both definitions are collapsed. The element displayed for `main` changes every second.

## 5.5 Interactive Editing

Since we now have a separate editor for each definition, we can now selectively change the editing experience for definitions of a certain type.

type alias	City	⋮
name	String	
country	String	
population	Int	
capital	Bool	
location	(Float, Float)	
+		

cities	List City	⋮	▶	
name	London	Stuttgart	Dublin	+
country	United Kingdom	Germany	Ireland	
population	8416535	597939	1804156	
capital	True	False	True	
location	(51.507222, -0.1275)	(48.783333, 9.183333)	(53.347778, -6.259722)	

Figure 23: A record type declaration, as well as an interactive interface for editing a list of records.

Figure 23 features two examples of interactive interfaces: the first is a view for record type definitions, the second is a view for lists of records. The list of records is laid out like a spreadsheet: each element in the list is in a separate column, and every field within each element is on a separate row.

Each of these views is interactive: tapping/clicking on any of the fields allows the programmer to edit the value inside the field directly. The programmer can add new fields or list elements using the buttons labelled “+”, shown for each definition.

Notice that each of these definitions now contains an extra button beside the “play” button: this enables the programmer to edit the value as text, if she feels that the interactive interface is obstructing her, rather than helping her.

The “spreadsheet” view enables further drag & drop interactions: The programmer can now project a field from a list of records by simply dragging the row

label into “empty space”. In the example in figure 24, the programmer drags the `population` field from the `cities` list. The resulting code is `List.map .population cities`.<sup>8</sup>

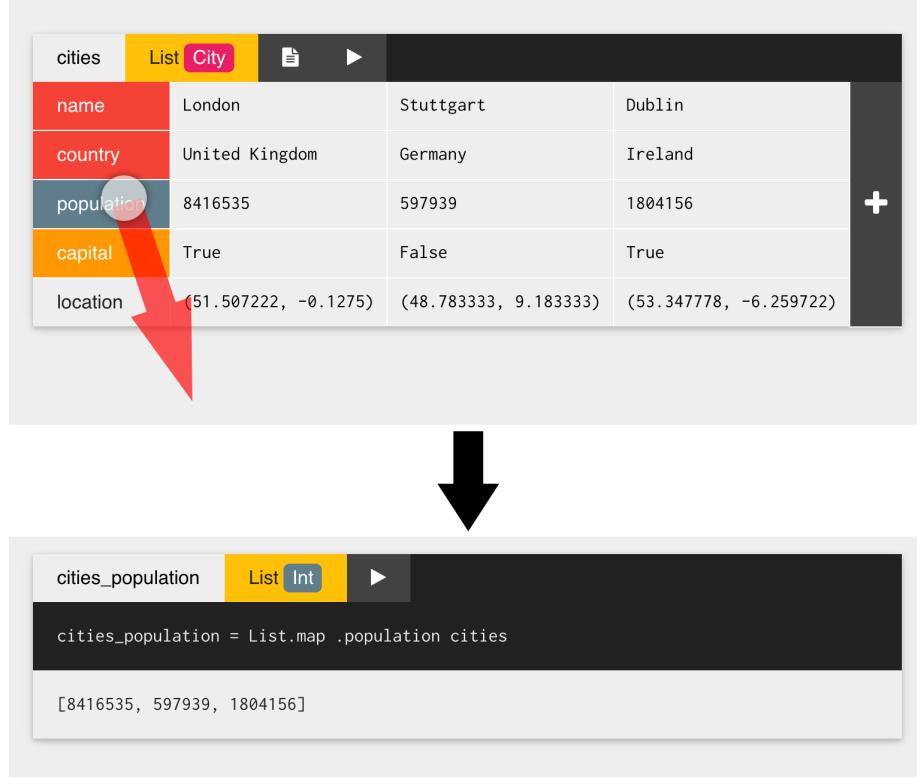


Figure 24: Projecting a field from a list of records by dragging the row label.

Such a component is likely to be only useful in particular circumstances. Notice that combining a strong type-system with the idea that every value has an entirely separate editor enables us to create highly context-specific editing interfaces. Since the programmer can always fall back to editing plain text, these components do not restrict the power of the editor. The editor should feature a mechanism for programmers to create their own components for interacting with values – many types of values have highly specific representations, and most will only make sense within a particular project.

<sup>8</sup>Recall that in Elm, record fields can be accessed using the notation `recordValue.field`. `.population` in this example is merely syntax sugar for the expression `\x -> x.population`.

## 5.6 Errors

In the process of writing a program, the programmer will certainly encounter many errors. Elm is designed so that the vast majority of programming errors can be detected at compile-time, rather than at runtime. Errors should be immediately visible to the programmer as soon as the error is made. In particular, errors should be visible as close to the source of the error as possible.

From a UI perspective, the simplest errors to show the programmer are misspellings. When editing text, these are frequent, but easy-to-fix errors. These kinds of errors occur at a single, well-defined point in the source code. The error message can thus be attached to the incorrect definition, and the misspelling itself can be highlighted in the code, as in figure 25. The error message shown in this example is an error message generated by the Elm compiler. The Elm compiler also outputs line and column numbers with such an error. Since the programmer is not editing an entire file, line numbers are meaningless inside this editing environment, and are thus not shown to the programmer.



Figure 25: A misspelt variable name in the code can be easily displayed to the user. The error is attached to the definition, and the location of the error is highlighted in the code. Additional visual feedback is given by replacing the “play” button by an “error” button.

Related to misspellings are syntax errors, which result in parsing failures. These are more difficult to recover from, since a parse error is likely to only manifest itself at the point of the next token. The reported location of the error might thus be far from the source of the actual error. An extreme yet effective way to let the programmer fix these kinds of errors is to simply display a single editor for the entire file, such as in a traditional code editor. The error can then be highlighted inside the text editor, as is common in most IDEs.

A more interesting class of errors are *type errors*. For an experienced programmer who is familiar with the syntax of the programming language, this is by far the most common source of errors. In fact, type errors are an essential tool for efficiently refactoring a large codebase. A common editing workflow in Elm or

Haskell is to change a datatype or function type, and then fix every point in the program at which a type error occurs as a result.

An important insight is that type errors always have at least *two* sources: A type error is a mismatch between the *expected* type of an expression and its *actual* type. However, the type that the compiler *expects* might not always be the type that the programmer *wants*.

Indeed, we can discern between two methods of implementing a program: When programming in a “top-down”, “wishful”[33], or “type-driven” manner, the programmer writes high-level functions and datatypes first, and then writes the underlying implementations of these functions, guided by the type system. One could say that the programmer defines *expected* types, and proceeds by fitting the *actual* types to the expected types. Programming in a “bottom-up”, or “implementation-driven” manner is the opposite: the programmer implements specific functions and then composes them afterwards – composing *actual* types until the *expected* type matches the “correct” type. In the process of writing any non-trivial program, the programmer is likely to move in both “directions” at some point. Thus, the editing environment should make no assumptions as to which type is “correct” – the programmer should be able to choose how to resolve a type error in either way.

## 5.7 Conclusion

Instead of detailing an entire programming system, I have chosen to focus on selected components and interactions. The most important idea is that programming should happen on a per-definition basis rather than a per-file basis. Considering each definition independently enables a number of powerful interactions: we can now interactively compose functions and values, for example via drag & drop; we can instantly evaluate any definition in place; we can choose more powerful representations for certain values; and we can create context-dependent interactive interfaces for certain values, while still being able to edit them as text.

Most of these interactions also rely on Elm’s powerful type system. By colour-coding types, we are able to give instant visual feedback as to which functions and values are related, and can be composed. We can let the programmer hide the implementations of definitions, since the name and type are in many cases sufficient for understanding a definition.

I explored some type-specific interfaces, such as the “spreadsheet” view detailed in section 5.5. While these are certainly great visual aids for understanding certain kinds of data, it remains to be seen whether these kinds of interfaces are useful in many cases. I believe that it is important for programmers to be able to develop their own interactive views in such a system. The “killer app” for such an interface is much more likely to be found by an exploring community than by a concerted design effort.

The interactions shown in this section are only mockups – they were implemented in a web browser using static data. These went through many iterations before converging on the interactions shown. Easily-modifiable, high-fidelity mockups proved to be an essential tool for developing these interactions. Since the main goal of this project is to develop a working system, I will now detail the development process of this system, which I have dubbed *Arrowsmith*.

## 6 Arrowsmith

Arrowsmith is an attempt to create a programming system which tries to embody the following principles:

- the programmer should have an immediate connection to the program she is developing
- the programmer should never have to do anything the machine can do for her
- the system should integrate with currently existing tools and environments

These are the principles I first stated in section 2.

As previous projects have shown, the design space for programming environments is large. The temptation is to start from a “clean slate”: to develop an entirely new programming language which perfectly supports the features being aimed at. I wanted to avoid creating a new language, and instead tried to find a language with an already existing environment most suited to creating this new editing experience.

Many of the interactions I ended up exploring as part of the design process are quite specific to Elm. However, before settling on Elm, I explored some other languages & environments.

As mentioned previously, the two key language features which allow us to create the editing interactions explored in section 5 are *purity* and a strong *type system*. The obvious choice of language with these features is *Haskell*.

Haskell was the first functional programming language I had any experience with, and the initial drive for many of the ideas shown here has come from my experience of using Haskell in the introductory programming course at Imperial College London. A big issue with Haskell as a “learnable system” is that performing any non-trivial task with it requires understanding of the `IO` monad, an incredibly powerful abstraction which allows writing effectful code in Haskell.

As well as being an impediment to beginners, use of `IO` also renders a Haskell program impure: our assumption that we can evaluate anything at any time would no longer be true. One solution would be to limit the programmer to a subset of Haskell without `IO`. This is a serious limitation: for example, it is impossible to render graphics in Haskell without the use of `IO`. For a system which aims to “move beyond text”, this is quite a damning limitation. Haskell is thus not an ideal choice as an editing language.

I briefly explored some other Haskell derivatives, notably PureScript[22]. PureScript is a language which compiles to JavaScript and thus runs inside a web browser. PureScript improves on Haskell’s treatment of `IO` with its “effect monad”: this allows the programmer to specify the particular effects that a

monadic action will have. However, again, this requires an understanding of monads to do anything “useful”.

Elm instead manages to contain effectful computation in a neat abstraction of time-varying *signals*. This is a much more limited abstraction, but one that is far easier to understand once the programmer is familiar with basic functional programming concepts.

Apart from its much simpler model for programming, Elm also has an active and enthusiastic community. The efforts from this community have already resulted in groundbreaking new tools for debugging, such as the “time-travelling debugger” by Laszlo Pandy, detailed in section 3.2.

In the following section I will detail Arrowsmith’s user interface, and will then go onto describing the project’s implementation. I found that Elm’s compiler did not easily support some of the features I required. I will detail the necessary changes I had to make to Elm’s compiler and build system, as well as some of the limitations of the approach I have chosen.

## 6.1 User interface

Arrowsmith is a web application which runs in any modern browser.

Integrating with current development systems is one of the key goals stated in section 2. One of the systems that developers nowadays interact most with is *Github*, the code hosting site based on the *git* version control system. Arrowsmith loads Elm projects directly from GitHub repositories – the user simply has to type in their project’s `username/projectname` combination into their browser’s URL field.

The most important user interface element in the editor is the *module view*, which is responsible for displaying a single Elm module. It consists of different components for definitions, imports and type definitions.

The module view is structurally similar to the textual representation of the module. Since we are not constrained to plain text, we are able to display useful interactive components for each part of the module. I will describe each of these components in detail.

### 6.1.1 Definitions

By far the most important component of the module view is the *definitions* view. The majority of the programmer’s interaction with the editor is performed here.

Each definition is given a separate editing field, as was demonstrated in section 5. An example of such an editing field is shown in figure 27. Each editing field shows the definition’s *name* and *type*, as well as the code corresponding to the definition.

The screenshot shows the Arrowsmith interface with a red header bar containing the title "Clock". Below the header is a toolbar with icons for information, file, and navigation. The main area contains three code snippets:

- main** `Signal.Signal Graphics.Element.Element`

```
main =
  Signal.map clock (every second)
```
- clock** `Float -> Graphics.Element.Element`

```
clock t =
  collage 400 400
    [ filled lightGrey (ngon 12 110)
    , outlined (solid grey) (ngon 12 110)
    , hand orange 100 t
    , hand charcoal 100 (t/60)
    , hand charcoal 60 (t/720)
    ]
```
- hand** `Color.Color -> Float -> Float -> Graphics.Collage.Form`

```
hand clr len time =
  let angle = degrees (90 - 6 * inSeconds time)
  in
    segment (0,0) (fromPolar (len,angle))
      |> traced (solid clr)
```

Figure 26: An example “Clock” module being edited in Arrowsmith.

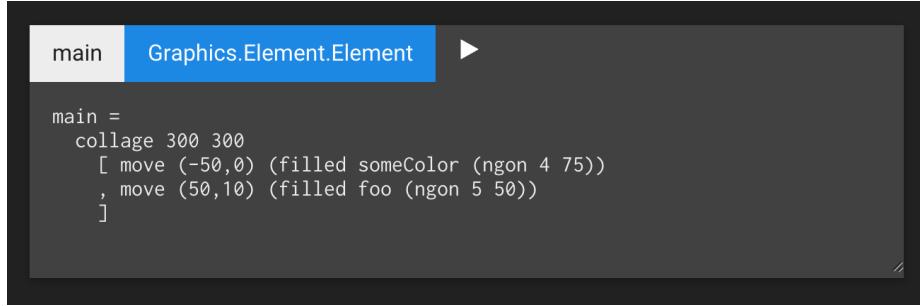


Figure 27: A single definition in arrowsmith.

The type shown in figure 27 is shown in blue: this means that the type was specified in the code. If the type was not specified, the definition's *inferred type* is shown instead. This is already an improvement over plain text: since each function has a separate editor, we can show function-specific information to the user without having to manipulate the underlying code. In section 5.1, I outlined several ideas for colour-coding types. I unfortunately failed to get this to work satisfactorily, for reasons I will outline in section 6.5.

The biggest innovation here is that each definition has a “play” button, which allows the programmer to evaluate the definition in question. After tapping/clicking on this button, the value is shown below the definition.

If the value in question is of type `Element`, it is simply rendered, as shown in figure 28.

Values which have a meaningful representation other than text are displayed using custom renderers. Currently, `Colors`, `Lists` and `Dicts` have simple special representations, as shown in figure 29. Values of type `Signal Float`, ie. time-varying numbers are plotted over time, as shown in figure 30. The plot updates automatically as the value changes. All other values are simply rendered using the built-in `show` function.

Tapping on a definition allows the programmer to change the code by simply editing text. When the programmer taps anywhere else, the resulting code is compiled and parsed, and the programmer instantly sees the new type of the definition, if it changed. If the programmer previously evaluated the definition, it is evaluated again, and the changed value is instantly shown to the programmer. If any other value depends on the definition that was changed, its value is also re-evaluated, and updated without programmer intervention. This enables an *immediate connection* to the code: the programmer instantly sees the effect of any changes to the code.

If the programmer makes an error when editing a definition, she gets immediate visual feedback: the background of the definitions view turns red, and the error is displayed at the top of the definitions view, as shown in 31.

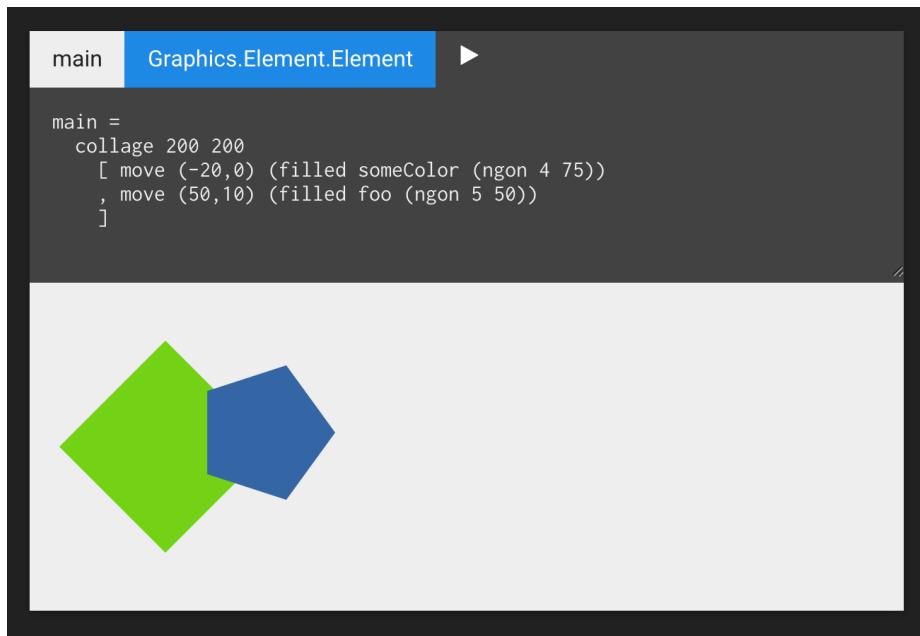


Figure 28: Pressing the “play” button evaluates a definition and displays the result below the definition. In this case, the definition is of type `Element`, so the element is simply rendered inline.

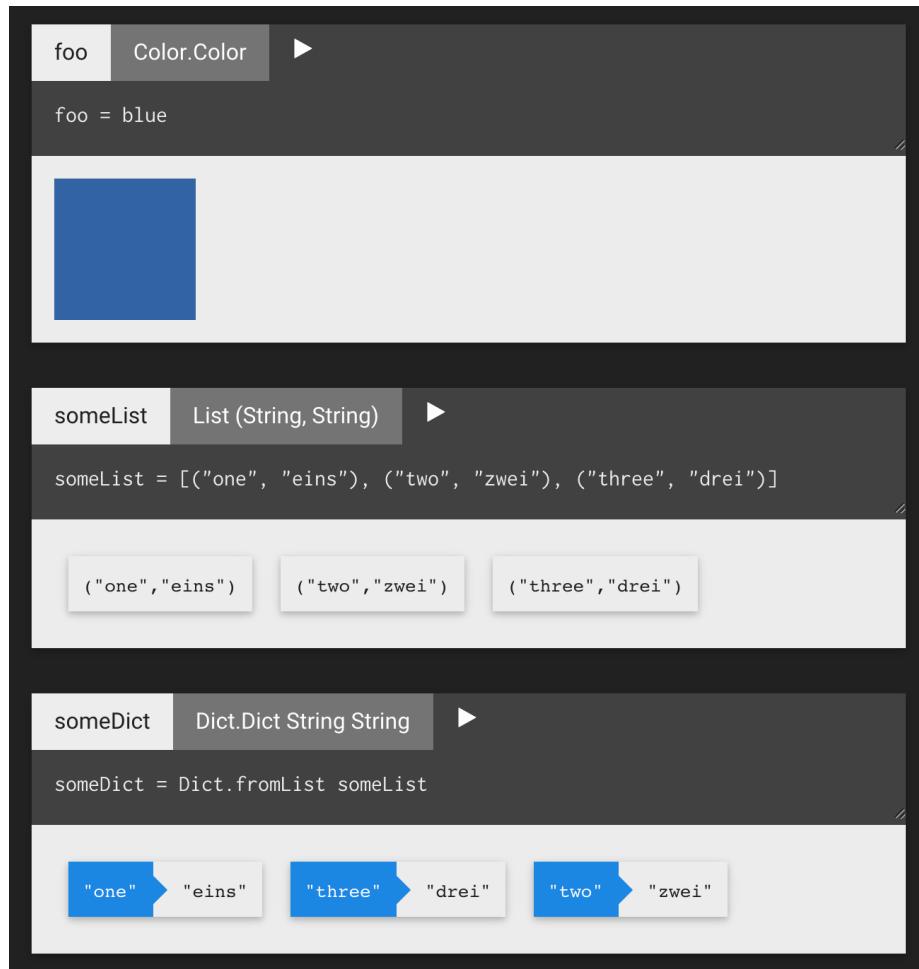


Figure 29: Special representations for certain kinds of values: Values of type `Color` are shown as a coloured rectangle, `Lists` and `Dicts` display separate “tags” for each element in the collection.

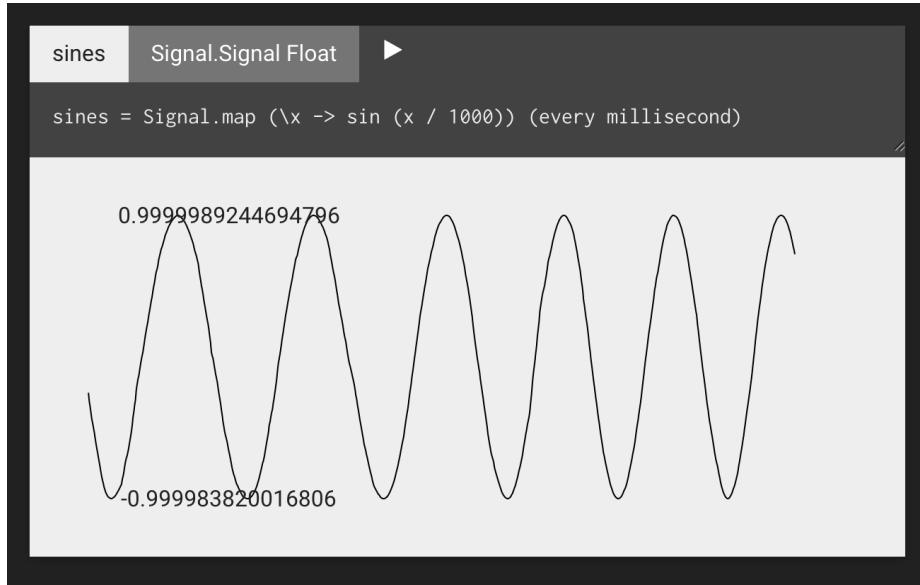


Figure 30: Values of type `Signal Float` can be plotted over time. The plot updates automatically with the latest values.

### 6.1.2 Imports

Any non-trivial module will contain a list of imports. The imports view is hidden by default from the programmer, since the list of imports is rarely useful while programming. It can be shown by clicking the button labelled “i” in the bar below the module header, as shown in figure 32. This is currently quite simple, showing the imports as they are written in the program. Structurally, an import is composed of three things: a *module name*, an optional *alias*, and an optional list of *exposed definitions*. Tapping/clicking on an import reveals separate textfields for each of these components. This approach eliminates the possibility of syntax errors in the imports.

### 6.1.3 Type definitions

Type definitions in Elm come in two flavours: *sum type definitions* (via the `type` keyword) and *type aliases* (via the `type alias` keyword).

Sum types are displayed as a table. Each row represents one constructor, which can contain argument types.

Type aliases are simply names for existing types. One of the most important use-cases for this is to give names to record types. Type aliases are simply shown

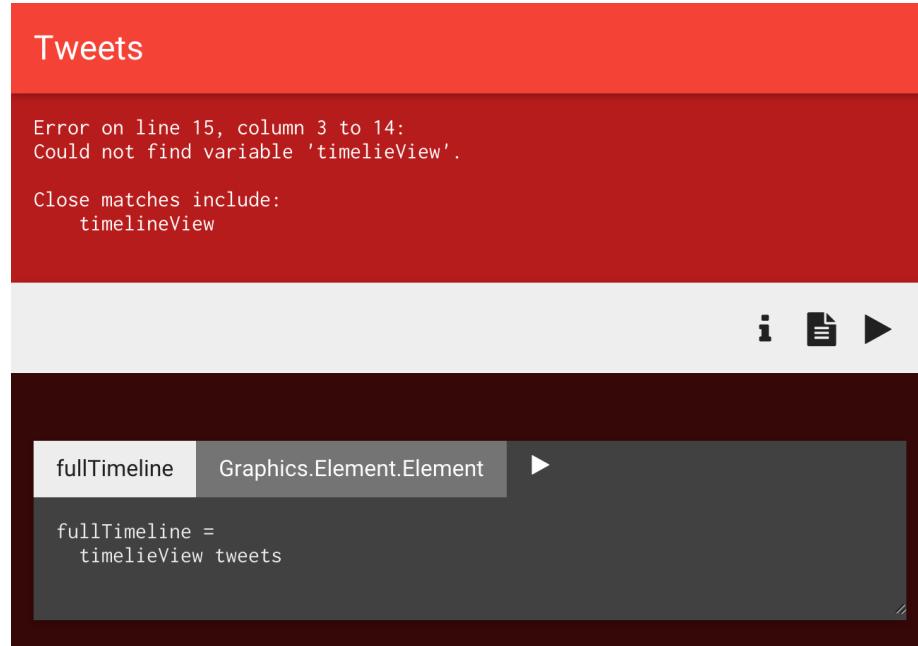


Figure 31: A definition in which the programmer has made an error. In order to provide instant visual feedback, the background of the definitions view is shaded red.

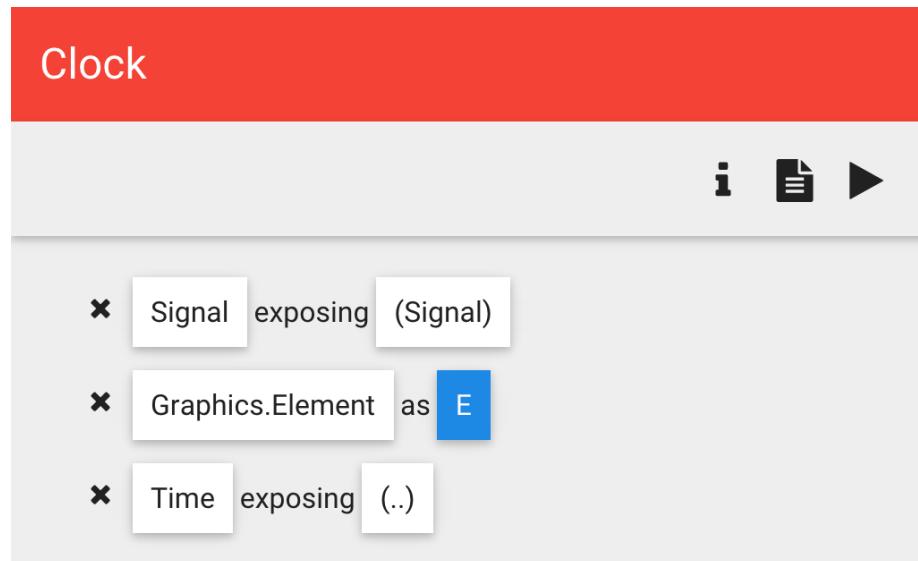


Figure 32: The imports view, shown after clicking the button labelled “i”.

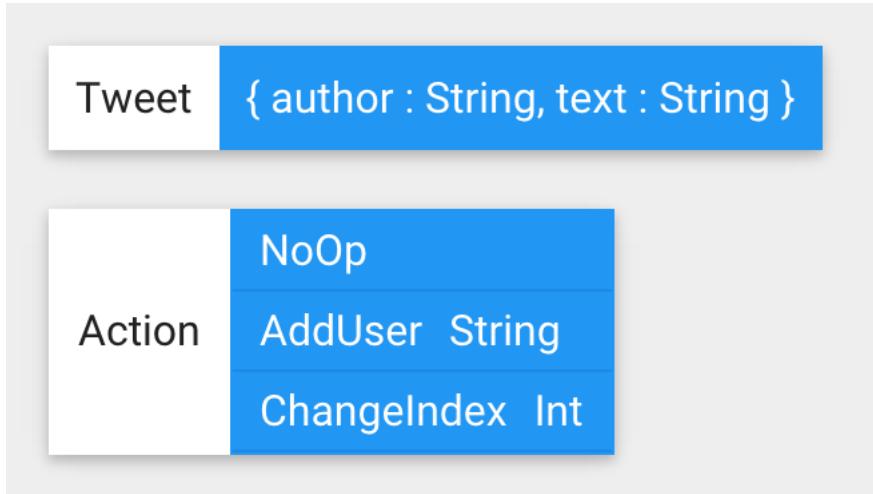


Figure 33: Top: a type alias for a record type. Bottom: A sum type displayed as a table. Each row represents one type constructor.

as “tags” containing the alias name and its corresponding type. These are shown in figure 33.

#### 6.1.4 Evaluating a Module

If the module being edited contains a “main” definition, then a “play” button is shown in the module header bar. When clicked, the definitions view is hidden, and the resulting application is displayed. The “play” button turns into a “pause” button, which allows the programmer to return to the definitions view.

#### 6.1.5 Plain Text View

In section 5, I argued that the programmer should always be able to fall back to plain text if the structural tools fail her. In Arrowsmith, the programmer can do this by clicking on the “text file” icon in the module header bar. The module being edited is then displayed as a plain text file, as shown in figure 35. When the programmer clicks on the “tick” icon, Arrowsmith compiles the code, and returns the programmer to the structured definitions view if no errors occurred.

#### 6.1.6 Project View

The project view, shown in figure 36, is a simple list of all the Elm files in the given git repository. Clicking on any of the module names takes the programmer to the module view.

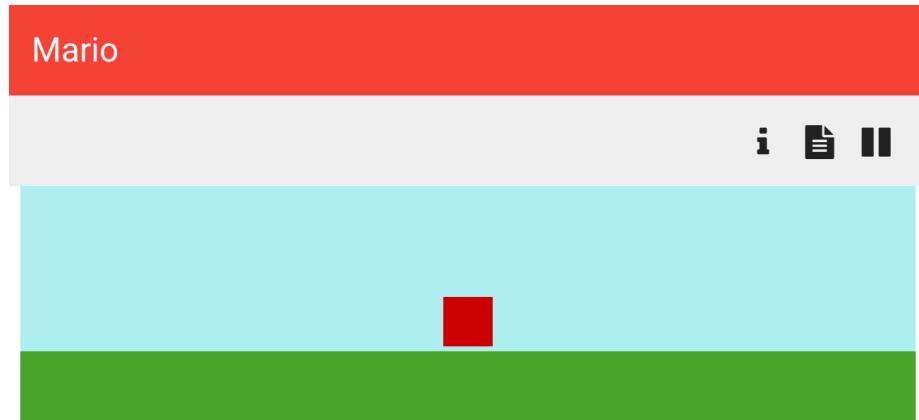


Figure 34: Pressing the “play” button evaluates a module’s “main” definition and displays the resulting application. The code is hidden. Pressing the corresponding “pause” button returns the programmer to the programming interface.



Figure 35: The fallback plain text view. This allows the programmer to edit code as usual.

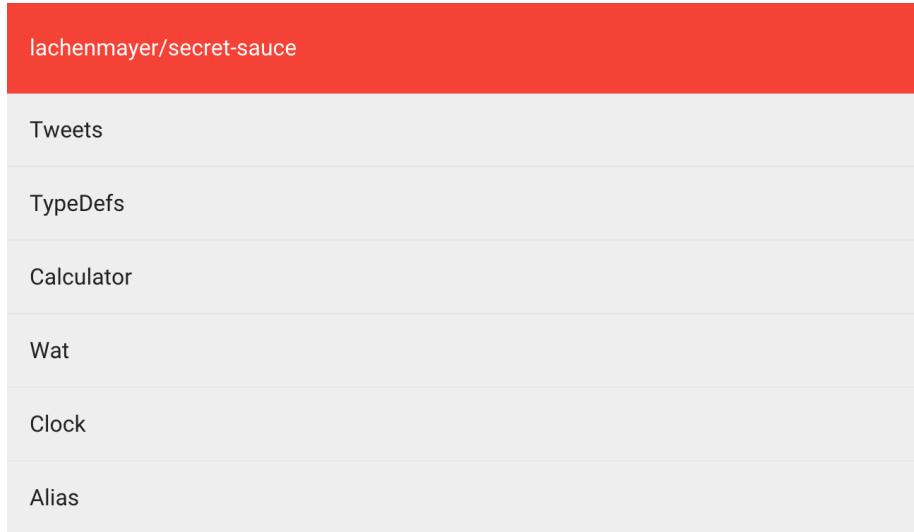


Figure 36: The project view simply lists all the Elm files in the repository.

## 6.2 Architecture

Arrowsmith consists of the following components:

- Browser front-end
  - Editor (Elm)
  - Environment (CoffeeScript)
- Back-end (Haskell)
  - HTTP server (Snap)
  - Compiler interface
- Modified elm-compiler
- Modified elm-make

These components will be explained in detail in the following sections. In these sections, I will try to focus in particular on aspects of the system which are unique consequences of the final design, rather than giving a comprehensive explanation of every component.

## 6.3 Front-end

Since Arrowsmith is a tool for programming in Elm, it was a natural choice to also implement the editor frontend itself in Elm. The hope for this was to

eventually *bootstrap* the editor, ie. develop the editor in itself. While I did not use Arrowsmith to develop any of its features, it did prove to be a useful prototyping tool in some cases.

While Elm is a great programming language for creating interactive user interfaces, it is quite limited when it comes to many other tasks. This is partially by design; most of Elm’s limitations simply stem from its immaturity.

Anything that Elm can’t do can be done in JavaScript. Elm features a great abstraction for communicating with JavaScript code in a type-safe way, *ports*. However, I found that the current implementation of ports is quite limiting. I will describe my use of ports in the following section, and detail some of these limitations in section 6.5.

Because of this, I separated the front-end code into the *editor* and the *environment*. The *editor* is an Elm program which is responsible for rendering the user interface and reacting to user input. The *environment* is a collection of JavaScript modules responsible for communicating with the editor backend and evaluating the Elm modules being edited.

### 6.3.1 Editor

The *Editor* component is an Elm program responsible for displaying the user interface. The code for this lives in `frontend/editor` in the Arrowsmith repository.<sup>9</sup>

Its main entry point is in the file `Editor.elm`. The code in this file is responsible for:

- interacting with the JavaScript *environment* through *ports*
- rendering either a `StructuredEditView` or a `PlainTextView`, and delegating actions to those
- rendering the module header
- displaying programmer errors

One of the most beautiful features of Elm’s programming model is that the entire application state is contained in one single place. Recall the “Elm Architecture” described in section 4.6. In the example described there, we contained the application state in one single `model` definition, a `Signal` which updates any time the user performs an action.

This architecture extends neatly to more complex applications with several sub-components, like Arrowsmith: The states of sub-components are simply contained as fields in the top-level `model` (in `Editor.elm`), and every single user action is sent to the top-level `actions` “mailbox”. The top-level `update`

---

<sup>9</sup>The code for Arrowsmith is publicly accessible at <https://www.github.com/lachenmayer/arrowsmith>.

function handles *all* updates, even those generated by sub-components: updates are delegated to sub-components by calling their respective `update` functions. This has a very useful implication for debugging: by logging the action with which the top-level `update` function was invoked, we have an immediate insight into all actions that are being generated in the entire application. This lets us easily verify that the application state is being updated as expected.

**6.3.1.1 Interacting with the Environment** In addition to actions generated by user interaction, the user interface needs to respond to actions generated by the back-end, and to the results of evaluating definitions in the Elm program being edited. To support this, Elm has a feature called *ports*. These allow passing values to and from the JavaScript runtime environment from which the Elm program was invoked.

*Outgoing* ports allow passing values to JavaScript: in an Elm file, an outgoing port is declared as a `Signal`. Any time this signal updates, its current value will be sent “through the port”. In JavaScript, the port is exposed as a named field on the Elm module object, and a callback can be registered which will be called with the value that was emitted by the signal. In practice, an outgoing port’s signal is usually defined by `filtering` and pattern-matching on the application’s `model` or `actions` signals.

*Incoming* ports are declared by simply stating their type in Elm. In JavaScript, these are exposed on the Elm module object as objects containing a `send` method. Invoking this method with a value updates the port’s signal with the specified value. Elm performs a runtime type check on any value sent through the port. This is a wonderful feature which ensures the type safety of Elm programs, but in practice, some of this project’s limitations stem from this feature. These will be expanded on in section 6.5.

As a concrete example, clicking a “play” button results in an `Evaluate` action being emitted in the application’s `actions` signal. In `Editor.elm`, a port called `evaluate` is declared. It filters all the `Evaluate` actions from the `actions` signal, and emits the names of the definitions which should be evaluated in the environment. These are then evaluated by running some JavaScript code (details in the following section), and the resulting value is sent back to the Elm program.<sup>10</sup>

**6.3.1.2 Model** The most important part of the Editor’s state is the elm file being edited. This is modelled using a type called `ElmFile`:

```
type alias ElmFile =
    { filePath : FilePath -- relative to project root
```

---

<sup>10</sup>This example as stated is not entirely true – the actual definition of `evaluate` is more complicated. For example, values are also re-evaluated when the module has just been updated with newly-compiled code.

```

, fileName : ModuleName
, source : ElmCode
, compiledCode : Maybe String
, modul : Maybe Module
, inRepo : Repo
, errors : List ElmError
}

```

This type contains everything that is required for displaying the editor frontend and for evaluating definitions in the file. Whenever a module is changed, the backend compiles the Elm file and sends the resulting value of type `ElmFile` to the frontend. This is exposed in the Editor as a port: `port compiledElmFiles : Signal ElmFile`.

This signal is *merged* with the user's actions in the Editor's top-level `model` signal. Since the view is simply a pure function which is mapped over the `model` signal, we don't need to perform any synchronisation of the view with the model. Most importantly, since the entire application state is in one well-defined place, all sub-components are immediately updated with the latest state. Synchronising sub-components has traditionally been a major pain point in developing web applications in JavaScript. This has proved to have been solved very neatly by the Elm approach.

**6.3.1.3 View** The two major sub-components in the Editor are the `StructuredEditView` and the `PlainTextView`. Of these, the `StructuredEditView` is much more interesting – the `PlainTextView` is simply a text field, along with an action to finish editing (exposed to the programmer via the “tick” button shown in figure 35).

Notice that the `modul`<sup>11</sup> field in the `ElmFile` type has a type of `Maybe Module`. If the Elm file can not be parsed, this value will be `Nothing`, and as a result the `PlainTextView` will be displayed, containing the Elm file's `source` code.

The `Module` type is a simplified model of the Elm abstract syntax tree. It contains fields for imports, type definitions, type aliases, inferred types, and definitions.

The main task of the `StructuredEditView` is to create definition views for each of the definitions in the `Module` type, as well as managing further sub-views: `ImportsView`, `DatatypesView` and `AliasesView`.

The `StructuredEditView` also handles the choice of *Value View* for values of a specific type. This is described in detail in section 6.3.3.

---

<sup>11</sup>This misspelling is intentional – `module` is a reserved keyword in both Elm and Haskell, and can thus not be used as a field name.

### 6.3.2 Environment

The Environment is a set of modules which are responsible for:

- launching the Editor,
- sending edit updates to the backend and sending updated ElmFiles to the Editor, and
- instantiating and evaluating the Elm module being edited.

The Environment code is in `frontend/environment`. The modules are written in CoffeeScript, a JavaScript preprocessor which adds some syntactic conveniences such as significant whitespace, value destructuring and a lack of superfluous punctuation. They are transformed into a single JavaScript file using the *webpack* module bundler[32]. This JavaScript code is loaded every time the user navigates to an Arrowsmith URL.

**6.3.2.1 Launching the Editor** The main entry point is in `main.coffee`. This file sets up two *routes*, which specify what needs shown to the user based on the URL the user navigated to:

- `/username/projectname`: This loads the *Project View* (figure 36), a simple Elm program which displays a list of all the modules in the project.
- `/username/projectname/ModuleName`: This shows the *Editor* for the specified module.

The *user name* and *project name* refer to the name of a GitHub repository. For example, the user can navigate to `/lachenmayer/arrowsmith-example`, which loads the project located at <https://github.com/lachenmayer/arrowsmith-example>.

When the user navigates to a URL of this shape, the relevant project or module are retrieved from the REST API exposed by the backend via an HTTP GET request. These calls to the backend are defined in the file `get.coffee`.

When a *project* is requested, the `Arrowsmith.Project` Elm program is initialised, and the received project information is passed to this Elm program through a port.

When a *module* is requested, the `Arrowsmith.Editor` Elm program is initialised with the received `ElmFile`. Additionally, callbacks for the *editing* and *evaluation* ports are declared.

**6.3.2.2 Edit Updates** The callbacks for editing are defined in `edit.coffee`.

In the Editor, an `editDefinition` port is defined which updates every time the programmer finishes editing a definition. When this port updates, the

`editDefinition` callback in `edit.coffee` retrieves the current value of the specified definition’s text field, and creates an *Edit Action* specifying the change. Edit Actions are described in detail in section 6.4.4.

The constructed Edit Action is then sent using an HTTP POST request to the backend, in the file `update.coffee`. In response, the backend returns the updated and recompiled Elm file. This is passed to the Editor frontend through the `compiledElmFiles` port described in the previous section.

Other Edit Actions for adding and removing definitions, and editing plain text are defined in a similar fashion.

**6.3.2.3 Evaluating Elm Code** Any time an updated Elm file is received from the backend, it is immediately “attached to the environment” (using the function exposed in the file `environment.coffee`).

This is done by placing the Elm file’s compiled JavaScript code in an `iframe`, the “execution frame”, and attaching this to the page. An `iframe` is an HTML construct for creating an “inline frame” – its normal usecase is to embed into a web site contents from another web site. Instead, we are using it as a sandbox for the compiled JavaScript code: When a `<script>` tag containing JavaScript is inserted into an HTML document, it is immediately executed. By containing this `<script>` tag in an `iframe`, we ensure that none of the enclosing page’s state is overwritten. From the enclosing page, we are able to access any of the JavaScript values defined inside the `iframe`.

Any update in the `evaluate` port in the Editor results in a call to the `evaluate` function in `evaluate.coffee`.

Values of type `(ModuleName, List (Name, ModuleName))` are passed through the port. The first field, of type `ModuleName`, contains the name of the module which contains the definitions to be evaluated. The module name is used to retrieve the Elm module from the “execution frame”.

The second field contains a list of the names of the definitions which are to be evaluated, as well as the name of a Value View for each definition. These are described in detail in the following section.

We evaluate each definition in the list by calling into a modified version of the Elm runtime (in `runtime.js`). This results in the definition’s value being rendered below the definition in the user interface.

### 6.3.3 Value Views

The standard Elm runtime only exposes functions to “evaluate a module” – ie. to evaluate its `main` definition.

The `main` definition is enforced by the compiler to be of type `Signal Element` (or `Element`<sup>12</sup>). This is because the runtime performs the actual rendering of the resulting graphics, as well as scheduling signals.

I extended the runtime to allow passing in a specific definition name to be evaluated instead of `main`. Since a definition can be of arbitrary type, the runtime had to be extended to render arbitrary types.

I solved this by creating *Value Views*. These are Elm modules which are able to transform a value of some given type into an `Element`, for rendering by the runtime.

As mentioned previously, each `evaluate` action contains the name of a Value View for each definition. This is used in the `evaluate` JavaScript callback to instantiate the correct Value View for each value.

Since we instantiate and call Value View modules “manually” from Javascript (which does not have a static type system), these modules need to conform to a specific interface. Every Value View contains an `info` field of type `ViewInfo`, and a `view` function, which is a function from any arbitrary type to `Signal Element`. Note that this is not the same as specifying the type to be `a -> Signal Element`, since most Value Views operate on values of a specific type. Run-time type checking is not possible in Elm, so we have to make sure that the given Value View is actually able to process a given value.

The `ViewInfo` type is defined as follows:

```
type alias ViewInfo =
    { matches : Type -> Bool
    , name : ModuleName
    }
```

The function `matches` is a rudimentary run-time type checking solution. The current implementations match types using regular expressions. The `name` of the module needs to be provided since Elm provides no way of accessing a module’s name at run-time.

The current Value View implementations can be found in the namespace `Arrowsmith.Views`. Simple special representations currently exist for `Colors`, `Lists` and `Dicts`, shown in figure 29. The `GraphView` plots the value of a `Signal Float` over time (figure 30). If no Value View matches the type of a given definition, the `SimpleView` renders the value, which results in a simple string representation.

This approach is easily extendable: a new Value View can be added to the system by simply providing a module which conforms to this interface. The

---

<sup>12</sup>Recall that this means that the `Element` does not change for the duration of the program. In the runtime, it is simply “lifted” to the `Signal Element` type using the function `Signal.constant : a -> Signal a`.

provided Value Views are quite simple, but they demonstrate the power of this approach. In particular, being able to plot **Signals** over time is an invaluable debugging tool.

A drawback of the current implementation is that these Value Views are read-only, ie. values can not be edited inside a Value View. This is mostly due to the fact that there is currently no satisfactory way to communicate between Elm programs running in the same environment. A further drawback is the fact that arbitrary records can not be rendered. This is because Elm provides no meta-programming tools such as type introspection or macros. Thus, it currently is not possible in Elm to fully achieve the vision outlined in section 5.5. However, even this partial solution is a significant improvement over current development tools, in particular for debugging time-varying values.

## 6.4 Back-end

The Arrowsmith back-end consists of a web server, which serves the frontend web application as well as exposing a REST API for retrieving Elm files and projects, as well as an interface to the Elm compiler.

The back-end is written in Haskell, mostly because the Elm compiler is also written in Haskell. The backend code lives in the `backend/` directory.

### 6.4.1 Web Server

The frontend code and API are served using the *Snap* web framework. This framework features a fast HTTP server as well as a “sensible and clean monad for web programming”[24].

Our usage of the Snap framework is quite simple: to serve the frontend, we define a “catch-all” route which simply loads the JavaScript code containing the Editor and the Environment. This is defined in `Main.hs`. This is done because the routing is performed in the client-side JavaScript code, as described in section 6.3.2.1.

The bulk of the serving code is in the API, in `Api.hs`. This file defines three routes:

- GET `api/:backend/:user/:project`: This route returns a `Project`, which contains a list of all the Elm files in the given repository.
- GET `api/:backend/:user/:project/:module`: This route returns an `ElmFile` (as described in section 6.3.1.2).
- POST `api/:backend/:user/:project/:module`: This route is used to perform edits on the given Elm file.

Retrieving projects and modules relies on `git`.<sup>13</sup> Projects are retrieved from a `git` repository using `git clone`, and stored within the `repos/` directory. A valid project is any `git` repository which contains an `elm-package.json` file. This file is used by the Elm build tools to determine the location of source files, as well as any package dependencies of a project.

Once a repository is cloned, Arrowsmith reads the `elm-package.json` file to find the locations of all the Elm source files in the repository. These are transformed into `ElmFile` objects which contain the Elm source code and all associated metadata. When a specific module is retrieved, the corresponding Elm file is compiled, populating the `compiledCode` field of the `ElmFile` object if compilation was successful, or the `errors` field if compilation errors occurred.

“Backend” in these routes refers to the host of the given `git` repository. Currently, the only working project backend is GitHub: in this case, the `user` and `project` fields correspond to GitHub user names and project names. This can be extended easily to any service which provides a `git` endpoint, by specifying a backend name and a URL from which the project is to be cloned. The compilation of Elm modules is described in detail in section 6.4.2.

The routes return JSON-encoded versions of the given datatypes. These are structured in such a way that they can be automatically transformed into Elm types in the frontend.

#### 6.4.2 Compilation

In order to provide a structural editing interface, Arrowsmith needs to be able to access an Elm file’s abstract syntax tree (AST). The Elm compiler by default does not expose the AST. Instead, a call to the public `compile` function (in the `Elm.Compiler` module) returns an *interface*, as well as the compiled JavaScript code. The interface is a high-level description of a module’s exported definitions, types and dependencies, and is used for cross-module typechecking. This unfortunately does not provide us with enough information about the structure of the code. I thus modified the `compile` function to also return the module’s full AST.

Elm files are normally compiled using the `elm-make` command, a build tool which installs a project’s dependencies and can compile multiple files. `elm-make` writes a compiled module’s interface and JavaScript code into *interface files* (with extension `.elmi`) and *object files* (with extension `.elmo`). I modified the `elm-make` package to depend on the custom `elm-compiler`. Since the modified `elm-compiler` returns a module’s AST along with the interface and JavaScript code, I modified `elm-make` to additionally write an AST file with extension `.elma` to the same location as the other two files. The AST file is simply a JSON-encoded representation of an Elm AST.

---

<sup>13</sup>A basic knowledge of the `git` version control system is assumed throughout.

In the `Arrowsmith.ElmFile` module, we compile an Elm file by launching the modified version of `elm-make`, and reading the resulting files from a temporary directory. Rather than exposing the raw Elm AST to the rest of the program, I created a much simpler model of the structure of an Elm program, the `Module` type.

#### 6.4.3 Module

The `Module` type is one of the most important types in the entire system. It is a simplified model of the Elm AST which exposes only the information required by the frontend to edit an Elm file, and is defined as follows:

```
data Module = Module
  { name :: ModuleName
  , imports :: [Import]
  , types :: [(VarName, Type)]
  , datatypes :: [(VarName, AdtInfo)]
  , aliases :: [(VarName, ([VarName], Type))]
  , defs :: [LocatedDefinition]
  }
```

The `types` field contains the types inferred by the Elm compiler for every definition in the module. The `datatypes` field contains the definition of every sum type (“ADT”). The `aliases` field contains the definition of every type alias, ie. alternate names given to previously-existing types. These fields, as well as the `name` and `imports` can be retrieved from the AST in a straightforward way.

Definitions are more complicated. They are defined as follows:

```
type LocatedDefinition =
  (VarName, Maybe Type, ElmCode, Location {- start -}, Location {- end -})
```

The two locations contain line & column numbers, and represent the range in the source code which bounds the definition. Since we want to allow the programmer to edit definitions as text, we need to be able to provide a mapping between a definition and its location in the original code.

The Elm compiler provides location information for expressions. Unfortunately, this does not include the left-hand side of definitions or a definition’s type declaration. Thus, the `Arrowsmith.Module` module contains some hacks to ensure that the reported locations bound the entire definition.

This approach is error-prone and severely limits the current implementation of Arrowsmith: it is currently not possible to edit imports, type aliases or type definitions, as these do not expose any location information at all. This is an

unfortunate limitation of the current Elm AST: if every node contained location information, changes in the abstract structure could be performed by editing the program text.

A different solution I briefly explored was to edit the abstract structure of the program directly, and then “pretty-printing” the AST. The results are unfortunately anything but pretty, since the pretty-printing does not take into account any “syntax sugar”<sup>14</sup> or code formatting, and removes comments. Reformatting the entire code is not acceptable for any tool which claims to integrate with existing systems and workflows.

#### 6.4.4 Editing

In the same way that Elm programs are structured as a sequence of pure function applications on an immutable state, edits of an Elm file are represented as pure functions applied to the Elm file.

Every edit is represented by a specific *Edit Action*. The `EditAction` datatype is defined as follows:

```
data EditAction
= AddDefinition Definition
| ChangeDefinition VarName ElmCode
| RemoveDefinition VarName
| ReplaceText ElmCode
```

The file `Edit.hs` contains a set of functions corresponding to each of these actions, which perform the necessary changes to the source code and `Module`. Importantly, these are *pure* functions: they only require an `ElmFile` and an `EditAction` as input, and produce exactly the same output any time they are invoked with the same arguments.

Once an Edit Action has been applied, the resulting source code is committed to the project’s `git` repository, and a machine-readable *Edit Update* tag is written to the commit message.

An Edit Update is a tuple, `(ModuleName, Maybe RevisionId, EditAction)`. If the edited code fails to compile, the `RevisionId` field will be populated with the revision ID of the last working version of this Elm file.

This is an essential feature of the system. If an Elm file can not be parsed, no AST is produced by the compiler. If no AST is produced, we can not easily provide a structural editing interface, since we don’t know the structure of the program. In order to provide a structural editing interface in that case, we

---

<sup>14</sup>As an extreme example, the simple `if`-expression, eg. `if foo then bar else baz`, is actually syntax sugar for a “multi-way” `if`-expression using guards. The example expression would be pretty-printed as `if | foo -> bar | True -> baz`, which is hardly “pretty”.

compile the Elm file by “checking out” its last working revision. We then retrieve the Edit Updates from all the revisions which follow the last working revision, and apply the corresponding Edit Actions to the Elm file.

If the current revision contains a broken file and does not contain a corresponding Edit Update tag (such as when the file has been edited using a different tool), we have no option but to display a plain text editor to the programmer. Once the error is fixed, and the compiler is able to parse the file, we can obviously display the structured editing interface again.

This approach is an interesting application of the functional programming paradigm to the structural editing problem, with several benefits. Since every change to the program is represented as a pure function, we are able reconstruct a valid AST from a previous AST, even when the current version is not parseable. Storing each change in version control enables some interesting features. Since changes are described at a semantic level (ie. “add definition `foo`” rather than “add these lines, remove these lines”), the editor could provide a much more meaningful view of the history of the file to the programmer. The editor can also support truly “infinite undo” – the “undo state” is never lost, since every single change is persisted in the version control system.

There are still some outstanding improvements that could be made to this feature. In its current implementation, every change triggers a new commit. This can result in a very “noisy” `git` history. The editor should provide the programmer a way to “squash” these changes into a single commit, and to give this commit a meaningful commit message, while preserving the individual actions contained in the commit.

## 6.5 Limitations

While Elm is an absolute pleasure to work with, it currently lacks some essential features. None of these are fundamental limitations of Elm, instead most of the issues I discovered stem from its immaturity. The Elm community is dedicated to the thoughtful design of the language and libraries, and I am confident that most of the issues mentioned here will be addressed satisfactorily.

To illustrate the level of immaturity of the language: The project was originally written in Elm 0.14.1, and was ported to Elm 0.15 around April 2015. This required quite a bit of work, since I had modified the Elm compiler and the `elm-make` tool. However, upgrading the compiler was not simply an aesthetic choice; the language underwent some breaking changes in syntax, as well as some much-needed improvements to the core library. In particular, the `Mailbox` type was added, which is an essential aspect of the Elm architecture (as described in section 4.6), and hugely simplifies the treatment of sub-components in a complex Elm application. Additionally, the `Task` abstraction for describing effectful computations in Elm was added. Without this, it was previously not possible to send network requests from within an Elm application – the necessary requests

needed to be sent from JavaScript code communicating with the Elm program through ports. Since I had at that point already created a robust implementation for communicating with the backend in the “Environment” CoffeeScript code, I chose not to rewrite this part of the system.

*Ports*, Elm’s feature for communicating with surrounding JavaScript code, proved to be the biggest source of limitations in this project. The most glaring omission of this feature at the time of writing is that it is not possible to send values with sum types through a port.

This is due to the fact that values flowing through ports are represented as JavaScript (JSON) objects, and there is no obvious way to express sum types in JSON. Aeson[1], the de-facto standard Haskell JSON library, solves this using *Template Haskell*, the Haskell meta-programming extension. This automatically generates JSON de- and encoders for the specified types, with various options for representing sum types.

A similar approach is not possible in Elm, since it does not have any meta-programming capabilities such as macros or type introspection. However, Elm does expose a “raw JSON” datatype, from which the relevant de- and encoders can be generated manually. The discussion around solving this problem has been ongoing since February 2014[11], but a solution has yet to be implemented.

The biggest limitation this imposes on Arrowsmith is that types can not be represented structurally in the frontend code; instead they are currently represented as (“pretty-printed”) strings. Since the representation of types is such an important aspect of our design goal specified in section 5, I attempted to work around this on two separate occasions – in both cases deciding that the change would result in a disproportionate amount of useless code in the system.

Writing JSON de- and encoders is a routine task so simple that it can be automated, as Aeson does. The real issue in our case is that values of type `Type` occur in various deeply-nested places within the types used to represent Elm modules. These types would all have to be parameterised, and conversion functions written for each. Additionally, the `Type` type is a complex, nested structure which is itself parameterised. In the end, I decided that there were more interesting and/or important challenges to be solved, since this should really be handled by the compiler. (Unfortunately this issue was not fixed by the project deadline.)

This problem has a more general implication: currently, the only way two Elm programs can communicate is via ports “wired up” in JavaScript. They are thus limited to a small subset of types in their communication. For example, it is currently impossible to send a value of type `Element` between two Elm programs, only because `Color` is represented as a sum type.

Additionally, it is currently not possible to send key-value mappings expressed as JSON Objects through ports. The obvious interpretation would be to expose this as a value of type `Dict` in the Elm code, but since the contents of the Object also

need to be type-checked to ensure type safety, this is currently not implemented. I solved this issue by exposing all key-value mappings as associative lists<sup>15</sup>, and manually constructing the corresponding `Dict` values.

I would have liked to add more sophisticated code editing with syntax highlighting to the definition views using the CodeMirror[6] JavaScript editor library. Since it is not possible to run arbitrary JavaScript code within an Elm program, I was not able to get the library to work. The main complication is that a dynamic number of CodeMirror instances need to be instantiated, one for each definition being displayed in the edit view. Various “hacks” to solve this were unsuccessful – the real solution would be to create an Elm package exposing the library using Elm’s `Native` feature which allows the creation of Elm modules using JavaScript. This would be a nice future improvement, but is not an essential feature.

The Elm compiler itself was not designed with tools like this in mind. For example, compilation errors are currently not exposed in a machine-readable format. We would thus have to parse the error messages for location information and other metadata. One resulting limitation is that we currently do not display error messages inline with the offending definition, as described in section 5.6. Fortunately, this feature is already implemented for the next version of Elm, so the implementation of this feature will be trivial in future.

As mentioned in section 6.4.3, the AST does not expose an adequate mapping between the source code and the abstract structure. This is something that would be very useful for implementing similar tools in future.

At the beginning of the project I was worried that Elm was fundamentally not powerful enough to implement this project satisfactorily, since most Elm examples found online were either very simple or very buggy (or both). This fear has proven to be unfounded; Elm is a truly wonderful language which solves the specific problems of interactive user interface code in innovative ways. There is a small but enthusiastic community surrounding the language, and I expect that the above-mentioned kinks will be ironed out in due course.

---

<sup>15</sup>An associative list is a list of tuples containing the keys and values, ie. `List (key, value)`.

## 7 Evaluation

In the development of programming tools, quantitative measurements have proven to be effectively worthless. Programming efficiency is notoriously difficult to measure, and most quantitative measures are at best meaningless, and at worst misguiding and harmful. For example, “lines of code written” is an actively harmful measurement, since it provides no notion of whether a given problem has actually been solved: in most cases, solutions requiring less code are preferable.

Qualitative evaluations for these kinds of tools suffer from a problem of vagueness and lack of actionable data. Attempts to solicit feedback about the project from other programmers have resulted in insights that go not much further than “this looks really interesting”.

By systematically asking the right questions, we can perform a qualitative evaluation which can be as useful as a quantitative one in this specific context. In particular, constricting ourselves to a specific set of questions allows us to draw comparisons between systems, and allows us to identify exactly what is novel about our contribution.

In the following, we will use the conceptual framework stated by Bret Victor in his essay “Learnable Programming”, as introduced in section 2.2. This framework decomposes the vague notion of “what makes a good programming environment” into a set of specific, meaningful questions. It is particularly useful since it covers a wide range of problems which need to be addressed in some way by any system for programming. Notice in the following that these questions are entirely *technology-independent* – indeed they provide meaningful answers for the majority of programming systems, even such simple ones as pen and paper. In fact, the solutions proposed in the essay itself are quite different to our solutions, but the questions themselves have proven to be useful guides. In this section, I will attempt show how Arrowsmith addresses the problems hinted at by each of these questions, and how the system can be improved further.

### 7.1 read the vocabulary – what do these words mean?

Arrowsmith in its current state only provides a marginal improvement on current systems with regards to this question.

In the functional programming paradigm, types are by far the most useful way to figure out “what the words mean”. Within a single module, Arrowsmith provides some useful information: the inferred types are shown for each definition in the module, even if they are not explicitly declared in the underlying code. This is already quite helpful, but this does not go far enough.

Arrowsmith should provide an easy way to see the types of definitions from outside the current module. This is currently possible by naming a definition, and looking at the inferred type which appears. However, this is hardly “immediate”.

The type of any “word” being entered by the programmer should be immediately visible on the screen. That this is possible is demonstrated by the “Try Elm” code editor featured on the Elm website[31]. This provides type information for any definition under the programmer’s editing cursor. Integrating this into the current Arrowsmith system should not be a difficult task.

Additionally, Arrowsmith currently does not support “docstrings”, which allow the programmer to provide documentation for every definition in a module. This documentation is currently exposed in the Elm package repository in a similar way to Java’s “Javadocs”. This is quite a glaring omission, which I only realised late in the development process. This feature could be added quite easily – Arrowsmith’s user interface is perfectly suited for exposing this documentation in an unobtrusive way, and the Elm AST provides a simple way to access the docstring for any definition. Each definition’s editor could expose an additional button which shows the documentation corresponding to that definition, in the same way that the mockup in figure 23 shows an additional button for editing a definition as text.

## 7.2 see the state – what is the computer thinking?

Learnable Programming demands the following of a programming system to enable the programmer to *see the state*:

- The environment must **show the data**. If a line of code computes a thing, that thing should be immediately visible.
- The environment must **show comparisons**. If a program computes many things, all of those things should be shown in context. This is nothing more than data visualization.
- The system must have **no hidden state**. State should either be eliminated, or represented as explicit objects on the screen. Every action must have a visible effect. [4]

Arrowsmith takes advantage of Elm’s programming model to tackle these issues: since computations in Elm are exposed as *definitions*, no line of code “computes a thing” until it is *evaluated*. Evaluation in Arrowsmith is exposed as a simple “play button” on each definition, which makes the computed value immediately visible to the programmer. The added feature of *Value Views* is another improvement: for example it allows the programmer to values of type `Color` as actual colours, rather than a vector of its RGB components.

Value Views can also be used to “show comparisons”: the elements of collections such as lists or key-value mappings are displayed side-by-side (as shown in figure 29). This feature could be improved further: more tools for data visualisation could be exposed to the programmer, such as plotting numerical values in a list.

Arrowsmith provides the tools to do so: Value Views are extendable, so additional views for specific types can easily be added. Most of these will be highly context-dependent, providing information relevant to a specific type.

Elm excels at the third point: Elm programs contain absolutely no hidden state. The only construct for storing state is the `foldp` function on `Signals`. Arrowsmith allows the programmer to represent this state as an explicit object on the screen in exactly the same way that every other value can be displayed.

These simple user interface components are transformative to the programming experience: any change in the code is immediately reflected in the resulting values. The programmer is thus always aware of “what the computer is thinking”: she can inspect the value of any pure definition at any time, and time-varying values can be inspected just as easily.

The *Light Table* project shows that further improvements can be made: the *InstaREPL* feature in Light Table shows the programmer the value of every single sub-expression when an expression is evaluated. This is an immensely useful tool for understanding programs, particularly for rapidly locating the source of a bug. Unfortunately this feature would require a significant amount of additional work in both the Elm compiler and runtime, as well as in Arrowsmith itself.

### 7.3 follow the flow – what happens when?

The programming interactions in Learnable Programming are described in the context of imperative programs. In imperative programming, the question of “what happens when” is much more important than in functional programming: since operations are executed in order, line by line, every operation is temporally related. In contrast, the values of pure definitions are independent of execution order, since the output of a pure function is always the same, no matter “when” it is invoked. Of course, a user-facing program can not be entirely pure, since the user expects to be able to interact with it. This is where Elm’s programming model shines – it confines time-varying values to specific definitions which can be reasoned about independently.

The specific requirements on the programming system with regards to this question are:

- The environment can make flow tangible, by enabling the programmer to explore forward and backward at her own pace.
- The environment can make flow visible, by visualizing the pattern of execution.
- The environment can represent time at multiple granularities, such as frames or event responses, to enable exploration across these meaningful chunks of execution. [4]

Projects such as Elm’s time-travelling debugger and Elm Reactor demonstrate that Elm is perfectly suited to making flow *tangible*. These tools allow the programmer to explore forward and backward by letting the programmer pause time and move through time using a slider. I was not successful in adding these features into the current version of Arrowsmith, but the system is perfectly suited for exposing such features to the programmer.

Arrowsmith is quite successful in making flow *visible*: for example, time-varying numeric values can be plotted over time (as shown in figure 30). Signals of other types currently simply display their latest value. This can be improved further by creating a “timeline” Value View, which would allow the programmer to additionally inspect the past values of a signal. This can be easily achieved within the current system.

The third point is solved by Elm: “time” in Elm only ever proceeds in discrete, meaningful events. Our editing model follows this philosophy: we store edits of a program as meaningful events, such as “add definition `foo` with value `bar`”. This has some interesting implications for features which allow the programmer to explore past states of the program, as well as understanding the evolution of a program.

Elm’s time-travelling debugger features an even more powerful tool for making the entire flow of the program visible: it displays a labelled graph of all the signals in a given program (shown in figure 4). This would allow the programmer to see literally the entire state of a program as it evolves over time. However, integrating this feature into a usable system has proven to be a huge user interface design challenge: the second version of Elm’s time-travelling debugger, *Elm Reactor*, removed the “signal graph” view due to fears of it being too complex for the average user.

## 7.4 create by reacting – start somewhere, then sculpt

Arrowsmith and Elm provide an excellent set of tools for “creating by reacting”. Compared to most imperative programming languages, functional programming languages provide a much easier way to “sculpt” programs. Once defined, pure functions are easily composable. Because they are pure, the programmer can write and test each function individually, without having to worry about the function affecting the execution of any other part of the program.

However, the demands posed by Learnable Programming are more stringent:

- The environment must be designed to **get something on the screen as soon as possible**, so the programmer can start reacting. This requires modeling the programmer’s thought process, and designing a system that can pick up on the earliest possible seed of thought.

- The environment must **dump the parts bucket onto the floor**, allowing the programmer to continuously react to her raw material and spark new ideas. [4]

Arrowsmith allows the programmer to instantly inspect the value of any definition, once it written. As the result updates immediately, the programmer can easily react to the resulting changes.

In the essay, Bret Victor focuses on features “which can jump-start the create-by-reacting process”. These include an auto-complete feature which additionally populates default values for parameters, as well as ways to manipulate these values using sliders. This is currently not implemented in Arrowsmith, but this is a perfect example of a possible editing view enabled by the ideas outlined in section 5.5.

The idea of “dumping the parts bucket onto the floor”, which in the essay manifests in a list of possible functions and syntax constructs the programmer can choose from, is not addressed by Arrowsmith at all. This requires some additional interaction design to turn into a feature which works well for the majority of programming tasks. This is an interesting direction for future work. Elm seems to be well-suited for this kind of interaction: its strong type system enables very powerful auto-completion, as demonstrated by *Lamdu*.

## 7.5 create by abstracting – start concrete, then generalize

This is a very interesting aspect of the essay, and one that has not been addressed sufficiently in the current version of Arrowsmith.

- The environment should encourage the learner to **start constant, then vary**, by providing meaningful ways of gradually and seamlessly transitioning constant expressions into variable expressions.
- The environment should encourage the learner to **start with one, then make many**, by providing ways of using those variable expressions at a higher level, such as function application or looping. [4]

Victor demonstrates interactions which allow the programmer to extract a constant value into a variable, as well as turning lines of code into a function. This functionality is provided by some widely-used IDEs such as *Eclipse*, and is indeed very useful. Arrowsmith currently has no such features.

The Elm *language* once again provides a strong foundation for these kinds of interactions: since definitions are *referentially transparent*, the programmer can extract constant values and sub-expressions into separate definitions with the confidence that doing so will not change the meaning of the program. This kind

of feature requires a lot of further engineering effort in the Elm *environment*. The editor needs to be integrated more tightly with the Elm abstract syntax tree in order to provide such highly contextual editing tools.

## 7.6 Conclusion

The conceptual framework put forward in Learnable Programming has proven to be a very useful guide for the design of any programming system. By decomposing the design problem into these specific, actionable questions we are able to demonstrate the precise areas in which we have contributed an improvement, and where further improvements can still be made. These questions do not assume any particular features of the proposed solution. Indeed, our interpretation of these questions is in some cases quite different to their interpretation in the essay. For example, the functional programming paradigm addresses questions of state and time in fundamentally different ways to the imperative paradigm. Instead, they focus on the thinking process involved in programming, which is something that is far too often overlooked in the design of programming environments.

As our use of the Learnable Programming framework shows, Arrowsmith provides significant improvements in some aspects of the programming process, but a lot of further progress can be made. In particular, Arrowsmith gives the programmer a more immediate connection to the program than most other tools for functional programming. However, it can not truthfully be said that Arrowsmith is a particularly more “learnable” environment: using it effectively requires a solid understanding of the Elm programming model, and it provides virtually no assistance to the programmer for understanding the “vocabulary” of the standard library, or other libraries.

This is a significantly more complex task than the one we set out to solve. The Elm community provides a number of sources which aid in the learning process, such as gentle introductions to the Elm programming model (similar to the one in section 4), and comprehensive documentation of the standard library. Some “learnable” aspects can certainly be integrated into Arrowsmith itself, but careful interface design work needs to be undertaken to ensure that the tool does not become too complex or intimidating itself.

## 8 Future Work

The possibilities for future work are effectively infinite: with every innovation in human-computer interaction come new possibilities for creating novel interfaces for programming. In this section I will outline some of the more immediate future work enabled by the progress we have already made, and will then try to outline some more ambitious goals.

Value Views, which are explained in detail in section 6.3.3, have the possibility to enable transformative insights into the values in a program. The views which are currently implemented already give the programmer instantly valuable visual feedback. Further Value Views can easily be created and added into the system. This feature could be hugely improved by allowing programmers to add their own custom Value Views for certain specific datatypes in their program, by means of a “plugin” system.

I was not able to implement several of the ideas charted out in the Design section (section 5). In particular, the user interface treatment of types, as well as drag & drop function application deserve some further work.

Arrowsmith’s user interface provides an excellent framework for designing interactive interfaces for values of specific types. An immediately useful interface would be the “spreadsheet interface” described in section 5.5. Such a feature would also benefit immensely from a well-designed plugin architecture enabling programmers to specify their own editing UIs.

A further immediately obvious improvement would be the integration of Elm’s time-travelling debugging features. This would not be very difficult to implement, since the Elm runtime has been designed with such features in mind.

The treatment of programmer errors could be improved. Errors should be shown close to their source, and the editor should also provide functionality to “automatically fix” some errors. For example, misspellings of names are already exposed by the Elm compiler. The programmer should be able to fix such an error in one click.

Further improvements could also be made with type errors. For example, after changing a datatype, the programmer could be shown all occurrences of this type in a program, so that she can fix all possible type errors immediately. This would be invaluable for performing refactoring work in large applications.

An area which should definitely be explored is “true” structural editing for Elm programs, in the style of *Lamdu*. As outlined in section 3.9, the biggest limitation of currently existing structural editing tools is not that they are less powerful than editing text, but that they do not integrate well with essential parts of the programming ecosystem. Conceptually, such tools offer a huge improvement on plain text editing. In particular, the “hole-driven programming” ideas demonstrated by Lamdu are immensely powerful. Arrowsmith, with its

integration with GitHub and its emphasis on *hiding* rather than *replacing* text, seems like a suitable model for future work on a structural editing tool.

In the Design section I outlined some ideas for interacting with the program using drag & drop. This could be taken much further, with the aim of completely removing keyboards from programming. This has been attempted many times over, but I am confident keyboards will soon be just as archaic as punch cards.

## 9 Conclusion

In 2015, most programmers still develop programs by writing some code, imagine the resulting changes in their head, then run this code to verify their changes. As Bret Victor has demonstrated so forcefully, this is unfortunate: programmers do not have an immediate connection to what they are creating.

Indeed, the fundamentally important insight is this:

That is so important to the creative process: to be able to try ideas as you think of them. If there's any delay in that feedback loop between thinking of something, and seeing it, and building on it, then there is this whole world of ideas which will just never be. [3]

It is this particular problem that I tried to tackle with the design and implementation of Arrowsmith.

Arrowsmith gives the programmer a much tighter feedback loop between thinking of something, and seeing it: the programmer can instantly evaluate any definition in their program, and can see these values update immediately if she changes them.

What's more, the programmer is able to see meaningful alternative representations of these values, using Value Views. These give the programmer a deeper insight into the true meaning of the program. I have demonstrated Value Views for certain types of value, but the true power of this approach is that it is easily extendable. Value Views can be defined for any type, and can enable highly contextual ways of inspecting the values in a program.

Arrowsmith is in some sense a “structural” editor: it exploits the structure of the program to give the programmer separate editing interfaces for each definition. These can in theory be transformed into context-specific structural editing interfaces, although I was not able to implement a working version of this within the scope of this project.

What sets Arrowsmith apart from other structural editors is that it does not disregard the underlying textual representation of the program. Other structural editors completely replace this representation with interactive interfaces, while Arrowsmith simply *augments* the plain text editing experience. Because of this, Arrowsmith is able to expose as much or as little structural editing as is required by the programmer at any given point. In fact, the programmer can always continue to edit the program as text if structural editing tools fail her. In particular, its approach to structural editing allows the programmer to collaborate with other programmers not using this particular tool, since they can continue to use traditional editing environments.

The system was designed with some fundamental principles in mind. These principles are a useful guiding light for the future development of similar systems. In particular, the importance of giving the programmer an immediate

connection to the program is crucial for the successful design of programming environments. With regards to this principle, we have made some great progress with Arrowsmith. However, huge improvements are still possible.

Another crucial insight is that programming is not a “blank slate”: hundreds of thousands of hours have been spent developing a huge number of programming systems and processes. Arrowsmith integrates well with current programming systems and workflows: Elm projects are retrieved from GitHub or any other `git` host, and changes made in the system are preserved in the `git` history of the project.

This principle is ignored by most projects which attempt to provide a novel programming experience. It is very tempting to start from scratch, and to re-invent the world entirely. The end results are invariably lacking.

Even though functional programming is one of the oldest programming paradigms in existence, it has historically tended to remain at the fringes of the mainstream programming world. The paradigm provides a simple conceptual framework for reasoning accurately about programs. Unfortunately, this simple conceptual framework has been shrouded in arcane, academic jargon propagated by communities that value cutting edge research over all else. This has resulted in an environment that is needlessly hostile to beginners, as well as a lack of essential tools for software engineering. Communities such as Elm's, which focus on accessible, understandable documentation and a friendly environment for beginners are a welcome change.

I am confident that our conception of “programming” will change drastically in the coming years as some of the ideas being developed now make it into usable systems.

We will soon escape the horrible prison of monospace fonts, curly braces and semicolons, and will enter the world of thoughts manifested in front of our eyes.

## Bibliography

- [1] aeson: Fast JSON parsing and encoding | Hackage: <https://hackage.haskell.org/package/aeson>.
- [2] Braid: <http://braid-game.com/>.
- [3] Bret Victor - Inventing on Principle - CUSEC 2012: <https://vimeo.com/36579366>.
- [4] Bret Victor - Learnable Programming - September 2012: <http://worrydream.com/LearnableProgramming/>.
- [5] Bret Victor style reactive debugging - Laszlo Pandy: <https://www.youtube.com/watch?v=lK0vph1zR8s>.
- [6] CodeMirror: <https://codemirror.net/>.
- [7] DeepArrow: Arrows for “deep application” | Hackage: <https://hackage.haskell.org/package/DeepArrow>.
- [8] Elliott, C. 2007. Tangible Functional Programming. *International Conference on Functional Programming* (2007).
- [9] Elm - functional web programming: <http://elm-lang.org/>.
- [10] Elm’s Time Travelling Debugger: <http://debug.elm-lang.org/>.
- [11] Feature Request: Arbitrary (First-order) Datatypes in Ports · Issue #490 · elm-lang/elm-compiler: <https://github.com/elm-lang/elm-compiler/issues/490>.
- [12] Gorilla REPL Renderer: <http://gorilla-repl.org/renderer.html>.
- [13] Gorilla REPL: <http://gorilla-repl.org/>.
- [14] Hookway, B. 2014. *Interface*. MIT Press.
- [15] Hopscotch: <http://www.gethopscotch.com/>.
- [16] IHaskell: <https://github.com/gibiansky/IHaskell>.
- [17] IPython Notebook: <http://ipython.org/notebook.html>.
- [18] Lamdu - towards the next generation IDE: <http://peaker.github.io/lamdu/>.
- [19] Light Table - a new IDE: <https://vimeo.com/40281991>.
- [20] Matthias Mueller-Prove - Correspondence with Alan Kay: <http://www.mprove.de/diplom/mail/kay.html#Nov00>.
- [21] MIT Scratch: <http://scratch.mit.edu/>.
- [22] PureScript: <http://www.purescript.org/>.
- [23] Scratch File Format (2.0) - Scratch Wiki: [http://wiki.scratch.mit.edu/wiki/Scratch\\_File\\_Format\\_\(2.0\)](http://wiki.scratch.mit.edu/wiki/Scratch_File_Format_(2.0)).

- [24] Snap: A Haskell Web Framework: <http://snapframework.com/>.
- [25] Swift - Overview - Apple Developer: <https://developer.apple.com/swift/>.
- [26] Tangible Value - Haskell Wiki: <https://wiki.haskell.org/TV>.
- [27] The Elm Architecture - Evan Czaplicki: <https://github.com/evancz/elm-architecture-tutorial>.
- [28] Time Travel Made Easy - Introducing Elm Reactor: <http://elm-lang.org/blog/Introducing-Elm-Reactor.elm>.
- [29] TouchDevelop: <https://www.touchdevelop.com/>.
- [30] Tributary: <http://tributary.io/>.
- [31] Try Elm: <http://elm-lang.org/try>.
- [32] webpack module bundler: <http://webpack.github.io/>.
- [33] Wiki - WishfulThinking: <http://c2.com/cgi/wiki?WishfulThinking>.