Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

George Garrington

27/04/2021

Syphon: a domain specific language for functional, declarative GUI design using the MVU architectural pattern

Project supervisor: Nicholas Gibbins
Second examiner: Markus Brede

A project report submitted for the award of
BSc Computer Science

Abstract

Functional Reactive Programming (FRP) has frequently been used for creating functional GUIs, however recently the Model-View-Update (MVU, a.k.a. The Elm Architecture) pattern has emerged as a potentially simpler and more effective tool for this task; this paper will explore the benefits of the MVU pattern through the implementation of Syphon: a domain specific language for functional, declarative GUI design. The end-to-end implementation of a functional programming language is detailed exploring parser-combinators, a Hindley-Milner type system and transpiling Haskell-style syntax to ES6 JavaScript syntax.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

I have acknowledged all sources, and identified any content taken from elsewhere.

I am using resources built by others such as Megaparsec[1], Electron-React boilerplate[2], pure-rand[3] and Material-UI[4].

I did all the work myself, or with my allocated group, and have not helped anyone else.

The material in the report is genuine, and I have included all my data/code/designs.

I have not submitted any part of this work for another assessment.

My work did not involve human participants, their cells or data, or animals.

[1]https://hackage.haskell.org/package/megaparsec - accessed on 27/4/21
[2]https://github.com/electron-react-boilerplate/electron-react-boilerplate - accessed on 27/4/21
[3]https://github.com/dubzzz/pure-rand - accessed on 27/4/21
[4]https://material-ui.com/ - accessed on 27/4/21

Acknowledgements

I would like to thank my supervisor Dr. Nicholas Gibbins for all his advice and support, and for allowing me to pursue a project of my own passion. I would like to thank my mother for all her support.

Contents

1 Introduction

1.1 Motivation

Contemporary GUI programming is often carried out with an object-oriented framework such as JavaFX and Swing[5], Cocoa[6], GTK[7], and QT[8]. This paradigm encourages solving problems by modelling real world entities as "objects": units where state and behaviour are tightly coupled, and have pointers to other objects whose state can be queried, and mutable state changes can be invoked by objects on themselves and other objects. Tight coupling can create a system where components are not referentially transparent, as an object's behaviour is not solely determined by it's own contents and methods but is affected by how other objects interact with it, meaning as the complexity and interconnectedness of OOP code grows it becomes increasingly difficult to reason about and test functionality of isolated units. Side-effects, mutability, lock-based concurrency and other unsafe features such as type-casting common in OOP languages mean that little correctness can be guaranteed at compile-time. The pure functional paradigm offers some benefits over OOP: the complete lack of implicit, mutable state or variable reassignments forces loose coupling, allows for greatly enhanced compile-time guarantees (namely complete type correctness), ease of refactoring by following compiler error messages as a result of improved compile-time guarantees, and the treatment of functions as first-class values allows for improved abstractions and reusable code. Joe Armstrong the creator of Erlang[9] summarizes the benefits of functional programming over OOP well: "I think the lack of reusability comes in object-oriented languages, not functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle. If you have referentially transparent code, if you have pure functions all the data comes in its input arguments and everything goes out and leaves no state behind it's incredibly reusable." (Seibel (2009)).

FRAN (Elliot & Huddak (1997)) was one of the first frameworks to bring the benefits of functional programming to simple GUI design, pioneering the functional-reactive-programming paradigm (FRP) where things like the mouse coordinates and time itself are dynamic values that vary over time. Elm[10] started out adhering to this paradigm, but has since evolved into a distinct paradigm (MVU), where the GUI "view" of an application is treated as a function of some data type that represents state. It has been argued that Elm's approach is easier to reason about and has a smaller learning curve than FRP (Czaplicki (2016), Fowler (2020)), whilst remaining a purely functional language thereby holding the same advantages over an imperative framework. MVU neatly separates GUI code from the logic for changing states, effectively better achieving MVC's goal of seperating the "view" and "logic", where the line between "Controller" and "Model" components is often blurred.

Elm has done an excellent job at eliminating much of the boilerplate code associated with GUIs, however I wish to explore if the syntax and semantics of an Elm-like language could be further simplified. The literature is lacking in content that details the end-to-end implementation of a programming language exploring all of parsing, type checking, transpiling and run-time behaviour, usually one main area is explored; therefore, another goal of this paper is to give an overview of how one could implement the "full stack" of a high-level language. Vague familiarity with Haskell syntax and semantics is assumed.

1.2 Etymology

A key concept of the MVU pattern is a unidirectional flow of data. There is a single source of state accessed through system primitive functions, so a common design pattern is to "syphon off" separate parts of the state into separate functions solely responsible for handling those parts, hence I thought a plumbing related term was appropriate. From Wikipedia[11]: "A siphon (from Ancient Greek: "pipe, tube", also spelled non-etymologically syphon) is any of a wide variety of devices that involve the flow of liquids through tubes".

[5]https://docs.oracle.com/javase/8/javase-clienttechnologies.htm - accessed on 12/4/21
[6]https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html - accessed on 12/4/21
[7]https://www.gtk.org/ - accessed on 12/4/21
[8]https://www.qt.io/ - accessed on 12/4/21
[9]https://www.erlang.org/ - accessed on 12/4/21
[10]https://elm-lang.org/ - accessed on 12/4/21
[11]https://en.wikipedia.org/wiki/Siphon - accessed on 14/4/21

2 Background

2.1 The origin of functional GUIs: Functional Reactive Programming

Functional Reactive Programming (FRP) is a relatively new paradigm pioneered by FRAN. FRAN introduced event and behaviour primitives for modelling an animation through composable functions of dynamically changing values, rather than imperatively defining an animation. Events are discrete occurrences in time such as a mouse click, and behaviours (also known as signals in the literature) are dynamic values that change over time such as the x/y coordinates of the cursor, more formally a Behaviour of type `a` is a function from time to some value of type `a`. FRP has also been applied to other domains such as telecoms (Toczé et al. (2016)), robotics (Soshnikov & Kirilenko (2014)), music production (Nilsson & Chupin (2016),Negrão (2018)) and low-level programming on Arduino (Helbling & Guyer (2016)).

```
wiggle = sin (pi * time)
wiggleRange lo hi = lo + (hi - lo) * (wiggle + 1)/2
pBall = withColor red (bigger (wiggleRange 0.5 1) circle)
```

Figure 2.1: A FRAN example making use of behaviours Elliot & Huddak (1997)

Consider Figure 2.1, time itself is a behaviour whose value is increasing at the rate of the system time resolution, wiggle is a behaviour whose value is alternating between 1 and -1 over time due to the application of a sin function, hence `wiggleRange` is a behaviour whose value will alternate between the arguments over time. `bigger` renders the second argument scaled by the value of the first argument, therefore, an animation will be rendered of a circle of some size continuously alternating between a scale of 0.5 and 1.

2.2 Manipulating signal values

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

Figure 2.2: The general form of the signal lifting function in FRP

Discrete events and standard non-signal values can be lifted into a signal context (Perez et al. (2016)), for example a normal boolean value of True becomes a continuous boolean signal with the value `True`, and a discrete event can become a continuous signal with the data type `Event a`, where the signal has the value `NoEvent` or `Event a` depending whether the event is currently happening. Functions can be lifted into a signal context (Czaplicki, E. and Chong, S., 2013, p.2), for example a function that negates an integer becomes a function that produces a time varying integer signal of an input integer signal negated; this is done using a signal lifting function (Figure 2.2).

```
f :: Signal a -> Signal b
g :: Signal b -> Signal c
```

(a) Function signatures that are composed

```
h :: Signal a -> Signal b
k :: Signal a -> Signal c
```

(b) Function signatures for parallelism

Figure 2.3

Signal functions can be composed using signal combinators (Perez et al. (2016)), taking signal functions as arguments and producing new signal functions; for example composition, where a signal of type `a` is fed into the composed signal function `f . g` (Figure 2.3a) producing an output signal of type `c`, parallelism where a signal of type `a` is fed to two signal functions (Figure 2.3b), returning their results paired together with type `Signal (b,c)`, and the signal equivalent of the fixed point combinator `loop`, used for creating feedback loops in a signal network. All manipulation of signals uses these primitives to form the core logic of FRP: a network of interconnected signals.

2.3 Pull-based vs push-based FRP

FRP implementations are classified into 2 main categories: *pull-based* (the majority of implementations) where the current signal values are *polled* at regular sampling intervals in time, or *push-based*, an event-driven approach whereby events propagate signal value updates through the signal network as they occur, and no computation takes place unless an event occurs. Although signals have conceptually continuous semantics, this is not quite the case (Sculthorpe & Nilsson (2009)); they are an *approximation* of the current value, especially in the case of pull-based systems whose accuracy is determined by the sampling rate. I believe it makes more sense to think of them as a continuous stream of discrete values as is Elm's approach (E. & S. (n.d.)); this is definitely better suited to GUI applications for example the cursor may remain stationary for some time whilst a user is typing in a word processing application meaning no new values will be added to the stream representing the cursor position signal until the cursor starts moving again. (The original) Elm and Flapjax (Meyerovich & Krishnamurthi (2009)) are both examples of push-based FRP systems.

2.4 The MVU architectural pattern

2.4.1 Emergence

Although Elm was originally based on FRP signals, a page from the documentation[12] mentions how members of the community seemed to independently derive the same architectural pattern and opt for this approach instead: a single data type for storing the state of the application (Model), event "message" data types, a function to update the state based on the current state and an event message (Update), and a function to render the application based on the updated state (View); for this reason the MVU pattern is often referred to as the Elm architecture (Fowler (2020)). In 2016 Elm removed support for FRP and replaced it with the MVU pattern entirely and an abstraction called subscriptions (Czaplicki (2016)); instead of manipulating time varying signals, values that change can be "subscribed" to and emit standard event messages upon changing, for example whenever the cursor is moved a `MouseMoved (Int, Int)` message is sent to the update function or a `Tick` message whenever the system time increases by 1 second, just like the `Increment` message sent by the button in Figure 2.4. The MVU pattern has influenced numerous frameworks including Avalonia.FuncUI[13], Miso [14], Redux[15] and Flux[16] commonly used with React[17] (and recently the `useReducer` hook in React itself[18], indirectly inspired by Elm via Redux), and the upcoming .NET MAUI framework (S. (2020)) to name a few.

2.4.2 Example

```
type alias Model = Int

type Msg
  = Increment
  | Decrement

init : Model
init = 0

update : Msg -> Model -> Model
update msg model = case msg of
    Increment -> model + 1
    Decrement -> model - 1

view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

(a) A simple MVU example from the Elm website

(b) Rendered view of the example

Figure 2.4

Consider Figure 2.4, the buttons respectively send a `Decrement` and an `Increment` message to the `update` function when clicked. The model (state) is a type synonym for an Int, so the `update` function pattern matches the event

[12]https://guide.elm-lang.org/architecture/index.html - accessed on 12/4/21
[13]https://github.com/fsprojects/Avalonia.FuncUI - referenced on 12/4/21
[14]https://haskell-miso.org/ - accessed on 12/4/21
[15]https://redux.js.org/ - accessed on 12/4/21
[16]https://facebook.github.io/flux/ - accessed on 12/4/21
[17]https://reactjs.org/ - accessed on 12/4/21
[18]https://reactjs.org/docs/hooks-reference.html - accessed on 12/4/21

message and either increments or decrements the model. The new model produced is sent to `view` which renders buttons and a text label which displays the model, re-rendering whenever the model is changed.

2.4.3 "Time-travel" debugging

The benefits of the MVU architecture for visualising runtime behaviour have been explored (Litt (2020)); states and events can be recorded with timestamps and "replayed", for example the Elm "time-travelling debugger" (James (2014)) where due to Elm's functional purity and lack of implicit state, events and states can be recorded whilst running some code so the developer can "scrub" back and forth through this runtime period and observe how the value of the state data type changes, they can change any erroneous lines of code and "replay" the code where the events are sent to the update function at the same relative times, creating the same conditions which generated the error. Although this article was written using the old FRP-based Elm, you can see in the Mario example that it is actually something alot like the MVU pattern being used; this is an example of the community deriving the same pattern as mentioned previously.

2.5 MVU benefits over FRP

A clear benefit of MVU over FRP is that we are not working in a signal context requiring the use of special primitives, we are working directly on plain data types and declaring reactive behaviour based on these and a state data type which is much easier to reason about; we do not have to worry about "lifting" values and functions into a signal context which may be a hindrance for beginner functional programmers to grasp. It would appear there is a slight trade off between expressiveness and readability of functional code where FRP implementations sit firmly on the latter end of the spectrum, perhaps MVU sacrifices some expressiveness however programs are arguably easier to reason about and therefore possibly more maintainable. The way that MVU organically emerged from the Elm community is an indicator that this is a more natural abstraction for functional GUIs than signals.

2.5.1 A formal model of an MVU-based language

The original thesis on Elm was a standard implementation of FRP utilizing signals, the MVU pattern had not yet emerged so it was not explored here at all. Fowler (Fowler (2020)) gave the first formal model of an MVU based language by extending a concurrent lambda calculus to support subscriptions, and proved that the type system is sound.

3 High-level system design

3.1 Introduction

Syphon is a Haskell-like language inspired by Elm for declarative GUI design using the MVU architectural pattern, it is implemented as a CLI tool written entirely in Haskell that parses a Syphon source code file, ensures correctness with the type system and transpiles it to a JavaScript Electron[19] application.

3.2 MOSCOW requirements specification

The following requirements were identified in the progress report. Henceforth each of these requirements are referenced as M<number>, S<number>, C<number> e.g. M1 is the first requirement from the "Musts" list. M9 has been identified as an additional requirement since the progress report.

Musts

1. Produces cross platform desktop applications from a single codebase
2. Clear terminology, syntax and referentially transparent code providing an ideal platform for beginner functional programmers to learn
3. An implementation of the MVU pattern with support for subscriptions or a similar abstraction
4. CLI transpiler tool with useful static analysis such as type checking and clear error messages
5. Basic GUI components including panels, scrollable panels, buttons, sliders, text fields, text labels and images displayed from files
    (a) All "container" components like panels should have the ability to nest within one another
6. Layout/alignment and styling options for GUI components like padding and colour
7. A purely functional syntax where side effects are only permitted from system primitive functions like update and subscribe and responses are handled using data types such as event messages
8. Support for Haskell style ad-hoc polymorphism (type classes)
9. Provides a declarative GUI programming style with minimal boilerplate required

Shoulds

1. Support for concurrent/background computations
2. Support for user created modules allowing code to be separated into multiple files

Coulds

1. A formal model of the language with type derivations and small-step operational semantics
2. Support for separate windows communicating with each other
3. Support for simple animations
4. A language primitive to enforce lazy evaluation
5. Installer script for the CLI
6. Runtime visualization tools such as a "time-travelling debugger"
7. A simple centralised package management tool and repository similar to Cabal
8. Implementing a subset of the language that targets an alternative platform such as Flutter

3.3 Justifications

3.3.1 Target platform

Electron is chosen as the target platform as it is a cross-platform GUI framework with good documentation and a large community, is is more actively maintained and widely used than other options such as GTK and QT and has frequently been hailed as producing much nicer looking GUIs than the latter 2 which can appear slightly "alien" on non-Linux platforms. There are several industrial cross-platform GUI applications powered by Electron such as VSCode[20], Atom[21], Evernote[22] and Github Desktop[23]. This fulfills M1, there are examples of how C2 could be

[19]https://www.electronjs.org/ - accessed on 13/4/21
[20]https://code.visualstudio.com/ - accessed on 13/4/21
[21]https://atom.io/ - accessed on 13/4/21
[22]https://evernote.com/ - accessed on 13/4/21
[23]https://desktop.github.com/ - accessed on 13/4/21

implemented[24] and the NodeJS runtime which powers Electron supports "worker threads"[25], allowing for S1 to be implemented. JavaScript is a good target language as it natively supports functional programming features like functions as first-class values, lambdas and currying, meaning it can be used in a specific way to mimic the semantics of a Haskell-like language.

3.3.2 GUI library

The React library is used for creating GUIs as it is mature, has good documentation and a large community. The Material UI library is also used as it provides modern, aesthetic React components[26]. React recently introduced functional components and hooks[27] which are functions that allow accessing and manipulating state of components, and now encourages a functional-first rather than imperative approach; the `useReducer` hook in-fact implements the MVU pattern, the `useEffect` hook can be used to implement subscriptions, and a custom hook based on `useReducer` can be created that allows for side effects in response to event messages. This fulfills M3, M5, M6 and M7. There exists a React time-travel debugger implementation [28], demonstrating the possibility of C6.

3.3.3 Syntax and semantics

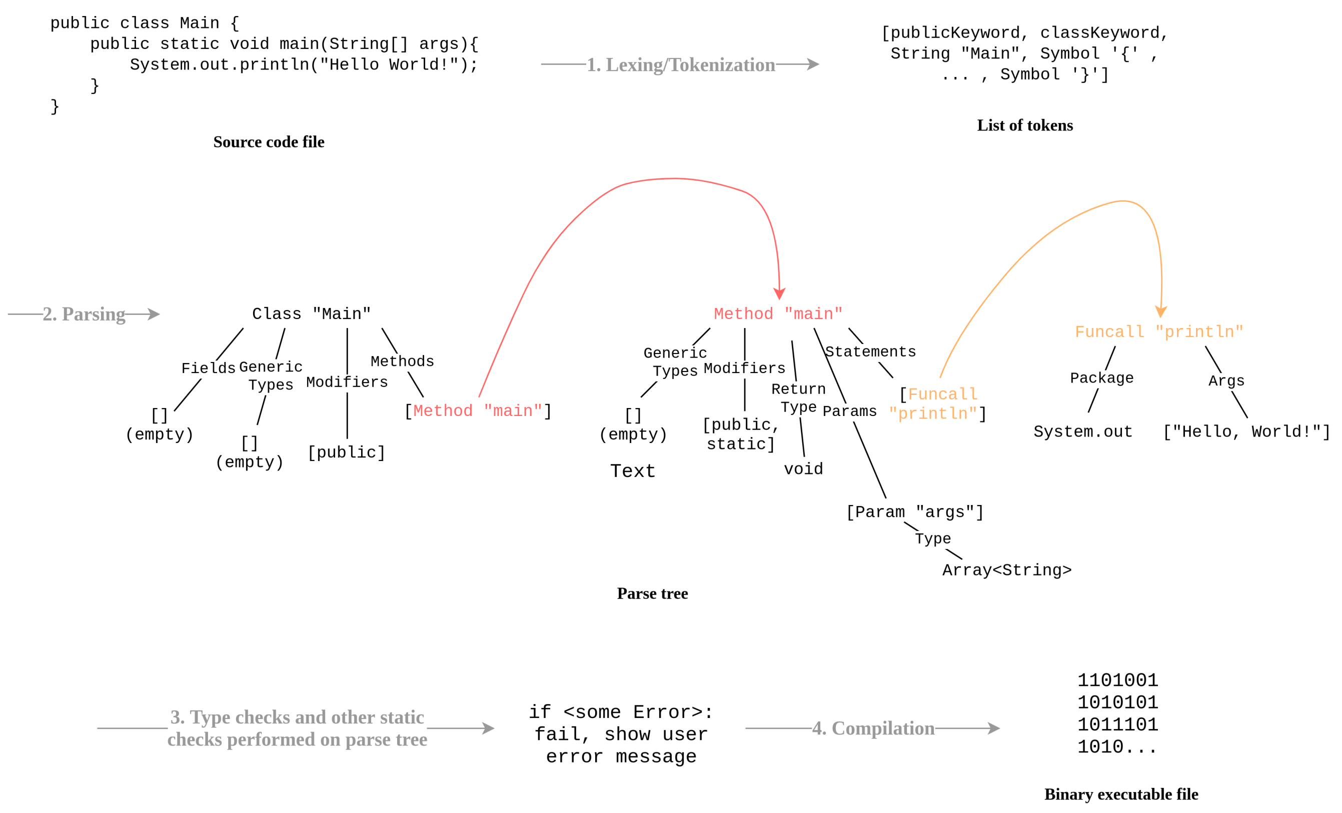
A Haskell-like syntax is chosen as it has become a sort of "lingua-franca" in the functional programming community and has a declarative nature with minimal boilerplate, fulfilling M2 and M9. Conversely to Haskell, strict evaluation is default as in Purescript and Elm, also both Haskell-like languages transpiled to JavaScript; this is because JavaScript is strict by default so lazy evaluation incurs an overhead. Elm includes an explicit lazy evaluation primitive [29], showing that C4 is possible.

3.3.4 Parser

Parsing (possibly preceded by lexing) is the first stage in a language pipeline, Figure 3.1 gives a general idea of how an overall pipeline might look; it is a classic, common approach where a lexer library groups characters into "tokens" conveying significant meaning to the parsing stage, then a parser-generator library takes as input the list of tokens and a context-free grammar, attempting to generate a parse tree from the tokens which is a valid program according to the grammar; this approach can only recognise a context-free grammar (think any language that uses curly braces and semicolons instead of indentation) so the syntax of a language like Python or Haskell where indentation conveys meaning, more formally context-sensitive, cannot be recognised (This is not entirely true, it *can* be done but is quite complex. Haskell uses a parser-generator as it needs industrial performance). I have used Java with an example parse tree of how you *could* represent the information from a Java source code file. Observe how the relatively small source code file in-fact has quite a lot of information that must be captured and arranged into a tree structure that is analysed and manipulated by the various stages. Compiling parse trees to a low-level language leading to a binary executable is outside the scope of this paper, instead we explore *trans*piling code from one high-level language to another.

Syphon's parser is implemented with the parser-*combinator* library Megaparsec[30], different from well known tools such as lex and yacc and their Haskell equivalents Alex[31] and Happy[32] which are parser-*generators* as discussed previously. Parser-combinators are more versatile in the grammars they can recognise as they are mainly powered by backtracking. Parser-combinators compose individual parser functions that each attempt to recognise some sequence of characters and return either a parsed data type with all characters consumed from the input stream, or some data type to denote a failure. Although performance is slightly worse than parser generators, there are a few benefits of this approach: there is a single parsing stage with no prior lexing stage, the parser is written directly in the same language as the rest of the pipeline rather than a domain specific language as in Alex and Happy files, and it's compositional nature allows for improved abstractions and modularity (Willis & Pickering (2020)). Megaparsec has excellent documentation[33] and support for custom error messages (Karpov (2017)), fulfilling part of M4. Performance is generally the best amongst the well known Haskell parser combinators (apart from Attoparsec[34], but this is not intended for source code files and lacks many features of Megaparsec) (Willis & Pickering (2020)) and it appears to be the most well maintained out of them.

[24]https://github.com/akabekobeko/examples-electron/tree/master/multiple-windows - accessed on 13/4/21
[25]https://nodejs.org/api/worker$_t$*hreads.html – accessedon*13/4/21
[26]https://material-ui.com/ - accessed on 27/4/21
[27]https://reactjs.org/docs/hooks-overview.html - accessed on 13/4/21
[28]https://github.com/reactrewind/react-rewind - accessed on 26/4/21
[29]https://package.elm-lang.org/packages/maxsnew/lazy/latest/Lazy - accessed on 26/4/21
[30]https://hackage.haskell.org/package/megaparsec - accessed on 12/4/21
[31]https://www.haskell.org/alex/ - accessed on 12/4/21
[32]https://www.haskell.org/happy/ - accessed on 12/4/21
[33]https://markkarpov.com/tutorial/megaparsec.html - accessed on 12/4/21
[34]https://hackage.haskell.org/package/attoparsec - accessed on 12/4/21



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

```
[publicKeyword, classKeyword,
 String "Main", Symbol '{' ,
     ... , Symbol '}']
```

Figure 3.1: A classic programming language pipeline

3.3.5 Type system

Syphon's type system uses Algorithm W (Milner (1982)) (a.k.a. a Hindley-Milner system), which powers the type inference system of many functional languages such as Haskell, Elm, OCaml, F# and ML. It allows for *parametric* polymorphism (et al (n.d.)), where a single function definition can work on arguments of different types for example `length :: [a] -> Int` which takes a list of any type and returns the length. Haskell has extended algorithm W to support *ad-hoc* polymorphism through type-classes, where the same function has different behaviour depending on the type of arguments (essentially overloading the function), like a `map` function with a different definition for `[a]` (maps the function over each element in the list), and for `Maybe a` (maps the function over the element in the `Just` constructor). This fulfills M4, and Haskell demonstrates possible extensibility for M8. The type system (Section 6) provides a formal model, partially fulfilling C1.

3.3.6 Final remark

It makes sense for the entire CLI tool to be implemented in Haskell; the parser is a Haskell library, the transpiler and type system being written in Haskell means the entire language pipeline is in Haskell, allowing for a well-designed pipeline as all stages work with the same parse tree data types. GHC[35] can produce a "fat binary" of the entire pipeline (the CLI tool) for Windows, Mac-OS and Linux, making later implementation of C5 trivial.

[35]https://www.haskell.org/ghc/ - accessed on 13/4/21

4 Syphon syntax and semantics

4.1 Introduction

I have crafted several programs to demonstrate Syphon's capabilities which are discussed more in Evaluation, I also discuss them here to give syntax examples. The full source code for programs can be found in Appendix B.

4.2 Preliminaries

4.2.1 The Lambda Calculus

```
--Observed syntax
(λ x → λ y → x + y) 1 2

--Actual representation and evaluation
    steps
((λ x → (λ y → ((+) x) y)) 1) 2
(λ y → ((+) 1) y) 2
((+) 1) 2)
3
```

(a) Simple lambda calculus example showing evaluation

```
map (Person 20) ["Michael", "Angela", "Jonathon"]
[Person 20 "Michael", Person 20 "Angela", Person 20 "Jonathon"]
```

(b) Evaluation of curried constructors, where `Person` is a data type storing age and name

Figure 4.1: Lambda calculus introduction

Created by Alonzo Church in the 1930s (Wegner (2003)), the lambda calculus is a high-level language for reasoning about the application of functions. It forms the core syntax and semantics of most functional languages like Haskell, Elm, F# and OCaml. White-space denotes application. All functions are *curried*, meaning functions are *expressions* that can only be applied to one other expression, as in Figure 4.1a where the "function" expression becomes a new function applied to a single argument at each evaluation step. A more practical example in an extended language supporting lists is Figure 4.1b where a partially applied constructor can be mapped over a list of arguments to form a list of fully applied constructors. Notice how constructors themselves are just functions. Lambda calculus allows reasoning about programs as pure expressions, allowing arbitrary nesting of expressions and a type inference system that can verify programs by defining rules for the behaviour of each expression.

4.2.2 Side effects and IO

Reasoning about functions as pure expressions means they cannot have side effects; something like a print statement simply cannot be expressed in a pure lambda calculus language as it does not evaluate to any value. Of course an IO facility is required for any programming language, however functional languages tend to keep the majority of the language "pure" and isolate side effects to specific structures, for example Haskell hides IO in a monadic context [36] and Miranda treats IO as streams [37]. Elm treats side effects as data, where side effects can fire response messages sent back to the update function upon completion[38]; for the MVU architecture this is the most natural approach whilst maintaining functional purity of code, so Syphon uses this.

[36]https://www.haskell.org/tutorial/io.html - accessed on 26/4/21
[37]https://courses.cs.washington.edu/courses/cse505/94au/functional/functional-io.txt - accessed on 26/4/21
[38]https://package.elm-lang.org/packages/chrilves/elm-io/latest/ - accessed on 26/4/21

4.2.3 Patterns

```
--In a match expression
match [1,2,3] with
   --Binds 2 to x, predicate stating first element is 1 and list length is at least 2
   1:x:_ -> [x]
   --Binds [2,3] to xs , predicate stating list length is at least 1
   _:xs -> xs
   --Binds x and z to 1, 2 respectively,
   --predicate stating list length is exactly 3 and second element is 2
   [x,2,z] -> [x,z]
```

(a) Basic pattern examples

```
type Event = Clear | Eq | OpPress Operator | Digit Int

--Cases in a match expression
OpPress op -> State "" op $ parseInt s.display
Digit i -> {s | display = append s.display (toString i)}
```

(b) Patterns being applied in the calculator program; `op` and `i` bind respectively to the `Operator` and `Int` fields of the data types

Figure 4.2: Patterns

Patterns (Figure 4.2) combine predicates and variable bindings on a data type into one ingenious syntax; they are a powerful, expressive tool for deconstructing data types whilst also maybe ensuring a predicate. If part of a data type is not required it can be replaced with a `'_'` wildcard pattern. Patterns can be used in Syphon in `match` expressions, function definition, lambda arguments, `let` expressions (Figure 4.2a) and `is` expressions (described later). Patterns may cause a runtime error if they are matched with an expression that conflicts with the predicate, for example `(\[x,y,z] -> x + y + z)[1,2]`.

4.3 Basic syntax

4.3.1 Function definitions

```
--Minimum that never goes below 0
posMin :: Int -> Int -> Int
posMin 0 _ = 0
posMin _ 0 = 0
posMin x y
   --x < y desugars to ((<) x) y as in lambda
      calculus
   | x < y = x
   --otherwise is a synonym for True
   | otherwise = y
```

(a) Function syntax

```
ballSpeed = 1
```

(b) A static constant being declared in the Pong program

Figure 4.3: Function and constant definitions

Functions are defined with a type signature at the top followed by one or more cases (so 3 cases in Figure 4.3a). The type signature is essential unlike Haskell, except for literal static constants (Figure 4.3b). Binary operators are treated the same as regular functions, they are syntactic sugar for curried application.

4.3.2 Algebraic data types and records

The `type` keyword declares an algebraic data type or record type (Figure 4.9a, Figure 4.4b), an algebraic data type as in Haskell can have several alternative constructors whereas record types unlike Haskell may have only a single constructor. Single constructor record types are actually safer, for instance in Figure 4.4c there is nothing to stop

```
type RecExample = {
    field1 :: Int,
    field2 :: Int
    }
```

(a) Syphon record declaration syntax, the right hand constructor is omitted as there is only a single constructor

```
type ADTExample
    = Constr1
    | Constr2 Int
    | Constr3 String
```

(b) Syphon ADT declaration syntax

```
data Rec1OrRec2
    = Rec1 {field1 :: Int}
    | Rec2 {field1 :: Int,
        field2 :: String}

getField2 :: Rec1OrRec2 -> String
getField2 rec = field2 rec
```

(c) Demonstrating the unsafeness of multiple record constructors in Haskell

(d)

Figure 4.4

a `Rec1` constructor being passed to `getField2`. This will cause a run-time error as `Rec1` does not have `field2`, this is not something that can be detected at compile time.

4.3.3 Type aliases

```
--e.g. (+), (-), (*) have Operator type
alias Operator = Int -> Int -> Int

type State = {display :: String, operator :: Operator, operand1 :: Int}
```

Figure 4.5: A type alias declared in the calculator program

The `alias` keyword declares type synonyms; they are a simple but useful feature, for (Figure 8.3) where the `Operator` alias improves both readability and conciseness. Any type definitions given by the developer can comprise of aliases, whose occurrences are substituted with their true type values prior to type checking.

4.3.4 Lambda expressions

```
\arg0 arg1 -> bodyExpr
```

Figure 4.6: Lambda functions in Syphon

Lambda syntax is the same as Haskell, with one or more argument patterns and a body expression.

4.3.5 Match expressions

```
update :: State -> Event -> State
update s e = match e with
    Inc -> s + 1
    Dec -> s - 1
```

Figure 4.7: A match expression being used in the increment counter program

`match` expressions (Figure 4.7) can be used for in-expression pattern matching, OCaml's `match <expr> with` syntax[39] replaces Haskell's `case <expr> of` syntax as I find it clearer.

4.3.6 Let expressions

Let expressions introduce variables inside the local scope of an expression. They are expressions themselves so can be nested within other expressions (Figure 4.8a). They are declared with a `let` keyword, followed by an

[39]https://ocaml.org/learn/tutorials/data$_t$ypes$_a$nd$_m$atching.html – accessedon26/4/21

```
--Case in a match expression
Reset -> let
   (newBombCoords, newGen) = genBombCoords s.gen 10
   newGrid = replicate2D 10 Hidden
   in
   {s | gen = newGen, bombCoords = newBombCoords, gameState =
        Playing, grid = newGrid}
```

(a) Example of a let expression being used in a sub-expression of a match expression in the Minesweeper program

```
square :: Int -> Int
square arg = calcSquare arg
    where
        calcSquare = \x -> x * x
```

(b) where clause of a simple function

Figure 4.8

indented block of definitions, the `in` keyword and the body expression within which the let definitions are in scope. `where` clauses in function cases are just syntactic sugar for let expressions, for example Figure 4.8b desugars to `let calcSquare = \x -> x * x in calcSquare arg`

4.3.7 Record expressions

```
--Get the value of field0
recordExpr.field0

{-
Results in a new record expression with all
the same fields as recordExpr except field0
now has the value "new value"
-}
{recordExpr | field0 = "new value"}
```

(a) Record updating and field selection in Syphon

```
--Selection
field0 recordExpr

--Updating
recordExpr {field0 = "new value"}
```

(b) Record updating and field selection in Haskell

```
type Person = {name :: String, age :: Int}

--Both valid constructors
Person "Matthew" 20
Person {name = "Caroline", age = 20}
```

(c) The 2 different record constructor syntaxes

Figure 4.9: Record syntax

Record syntax is borrowed from Elm (Figure 4.9a), with "dot" syntax for field selectors and "`|`" syntax for record field updating, this is much clearer than Haskell's record syntax (Figure 4.9b). As in Haskell, records are constructed using either the normal data type syntax or the field declaration syntax (Figure 4.9c).

4.3.8 Error funcionality

Sometimes it makes sense for a function to just *fail*, for instance getting the head of an empty list is illegal; a list can contain any type so how can this return a single value that matches with any type? This can be done anywhere in an expression with the `error` primitive(Figure 4.10).

```
sumThatFails :: Int -> Int -> Int
sumThatFails x y = x + y + (error "Demonstrating the error primitive")
```

Figure 4.10: Example of `error` being used in an expression

4.4 Novel syntactic structures in Syphon

4.4.1 `is` expressions

```
filter (\x -> x is Nothing) [Just 1, Nothing, Nothing, Just 2]
```

(a) Example use of an `is` expression in Syphon

```
let
isNothing x = case x of
   Nothing -> True
   _ -> False
in
filter isNothing [Just 1, Nothing, Nothing, Just 2]
```

(b) Achieving the same functionality in Haskell without `is` expressions

Figure 4.11: The convenience of `is` expressions

`is` expressions allow for shorthand matching of an expression with a pattern predicate (Figure 4.11a), eliminating the need for boilerplate needed without them (Figure 4.11b).

4.4.2 Optional structures

```
--First button assumes default background colour of Grey
--Both buttons assume default border radius value of 2
view :: State -> Widget
view = [Button (Text "click me") Button1Pressed,
   Button #{bgColor = Red} (Text "click me") Button2Pressed]
```

An optional structure can be thought of as a property of a variable itself that is a kind of optional "argument" to it. Declared with `#{}`, it is similar to a record except all fields have default values, it can be omitted as an argument completely meaning all default values are assumed, or it can be declared with any subset of fields which override the default values. For GUI design this is useful as widget data types have many properties that can be tweaked like font size, background colour and so on. I believe this is a unique feature for passing hidden data not found in other functional languages whilst maintaining functional purity and type correctness with a simple constraint: an optional structure may only ever appear as an initial "argument" to the variable whose properties are being tweaked; this means for instance one cannot map a constructor over a list of optional structures.

4.4.3 Multi-line structures

```
view :: State -> Widget
view s = Column []<<
   Container #{dim = (275,35), bgColor = Grey} $ Text s.display
   Row $ (map mkDigitButton [7..9]) ++ [Button (Text "X") (OpPress (*))]
   Row $ (map mkDigitButton [4..6]) ++ [Button (Text "/") (OpPress (/))]
   Row $ (map mkDigitButton [1..3]) ++ [Button (Text "-") (OpPress (-))]
   Row [mkDigitButton 0, Button (Text "AC") Clear, Button (Text "=") Eq,
      Button (Text "+") (OpPress (+))]
   where
      mkDigitButton i = Button (Text $ toString i) (Digit i)
```

(a) Multiline list in the calculator program

```
Canvas $<<
   #{onMouseMoved = if s.dragging then PixelEntered else \_ _ -> Dummy,
      onMousePressed = \_ _ -> MousePressed, onMouseReleased = \_ _ ->
           MouseReleased}
   700
   500
   s.strokes
```

(b) Multiline application in the paint program, where the Canvas is applied to it's optional structure and 3 mandatory arguments: width, height and list of shapes to draw

Figure 4.12

Multi-line structures are a neater way of declaring something over multiple lines, with elements on indented lines below forming members of structures. Multiline lists are declared with `[]<<` (Figure 4.12a) omitting the need for commas and a closing `']'`, and multi-line applications with `$<<` (Figure 4.12b) which applies the expression on the root line to all expressions in the "block".

4.4.4 Conditional expressions

```
scoreOrNewBallInfo = cond
    | iscLeftPad -> Right (bounce Lft s.ballDir)
    | iscRightPad -> Right (bounce Rght s.ballDir)
    | iscTop -> Right (bounce Top s.ballDir)
    | iscBot -> Right (bounce Bottom s.ballDir)
    | iscLeftWall -> Left RightPlayer
    | iscRightWall -> Left LeftPlayer
    | otherwise -> Right (s.ballDir, uncheckedY,uncheckedX)
```

Figure 4.13: Conditional expression in the Pong program

Inspired by MultiWayIf in Haskell[40], conditional expressions generalize an if expression to arbitrarily many cases, analogous to an "if/else" chain in imperative languages. Figure 4.13 shows it's usefulness, we must check several conditions for where the ball is intersecting within a `let` definition.

[40]https://ghc.gitlab.haskell.org/ghc/doc/users$_guide/exts/multiway_if.html - accessed on 26/4/21$

4.4.5 Pipeline blocks

```
pipeline
    1
    \x -> x + 2
    \y -> y * 3
```

(a) A simple pipeline syntax example

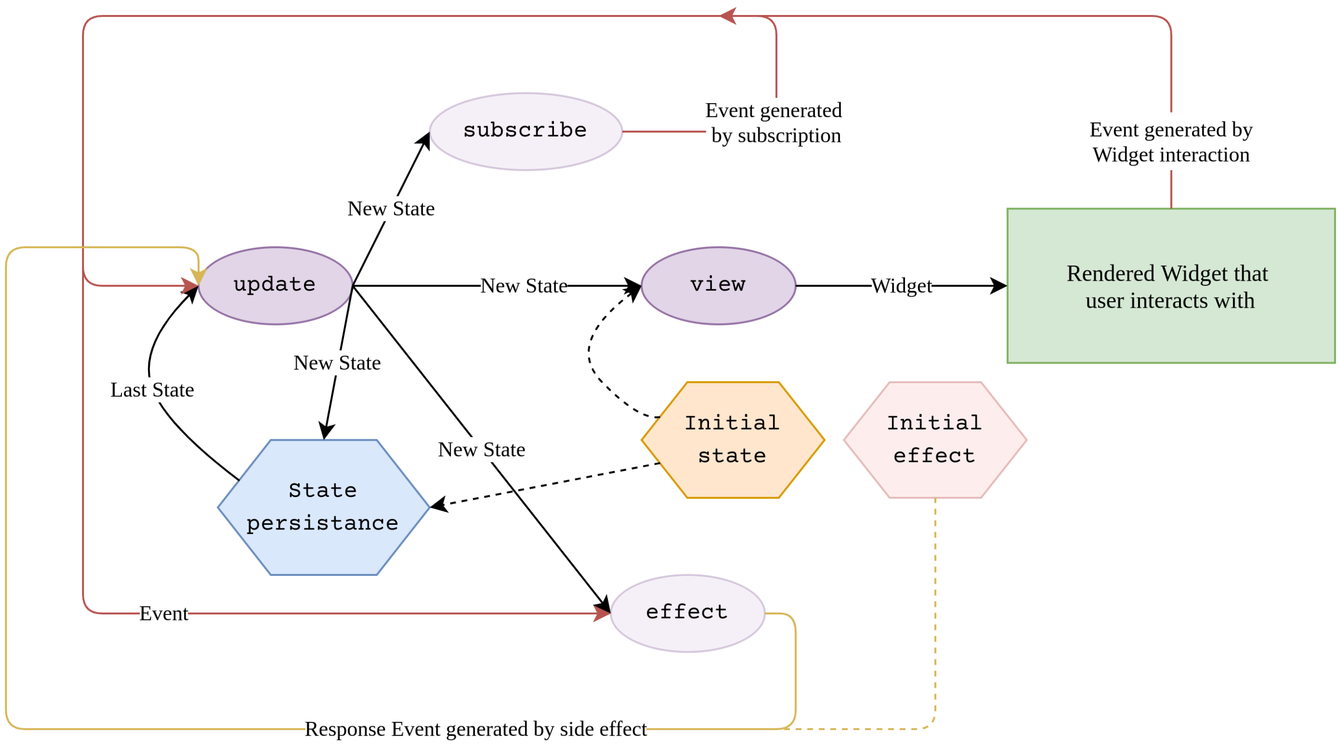
```
checkWon = pipeline
   s.grid
   filter2D $ \cellState -> cellState is Revealed _
   concat
   length
   \result -> result == 90
```

(b) Pipeline syntax being used in the minesweeper program, also demonstrating convenience of `is` expressions

Figure 4.14: A pipeline block in the Minesweeper program

The `|>` pipe operator is commonly used in F# and Elm, read as "feed the the left hand side into the right hand side", useful for reasoning about many consecutive function applications. Syphon generalizes this concept into a `pipeline` block, where the expression on the first line is fed into the expression on the second line and so on, it in-fact desugars to an application of the bottom line to all expressions on the lines above so Figure 4.14a is just `(\y -> y * 3)((\x -> x + 2)1)`. Partly inspired by Haskell do notation which generalized monadic binds with >> and >>= into a syntactic block, it provides some of the same benefits namely syntactic sugar and reasoning about code in an almost "imperative" manor, whilst having no monadic context like do notation; it should be noted that the similarities to do notation are merely superficial and pipeline blocks are just a generalisation of `|>` between several expressions; Figure 4.14b shows an application of pipeline syntax in the Minesweeper program where we want to count how many `Revealed` constructors there are in the 2D list storing cell states, and if it is 90 then the game is won.

4.5 System-primitive definitions



Event generated by subscription
Event generated by Widget interaction
subscribe
New State
update
New State
view
Widget
Rendered Widget that user interacts with
Last State
New State
New State
Initial state
Initial effect
State persistance
Event
effect
Response Event generated by side effect

Figure 4.15: Interaction between the system-primitive definitions to form a functioning, enclosed system with a unidirectional flow of data

The system-primitive definitions determine the behaviour of a program. Figure 4.15 accompanies the descriptions of each below to better demonstrate how they interact.

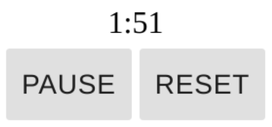
4.5.1 Core logic

Unlike Elm Syphon has no main function, instead the logic of the program comes from the definitions for the initial state `init`, the `update` function and the `view` function. Each time a new event message is generated the `update` function is called taking this and the last state as arguments, creating a new state which causes the `view` function to re-render based on the new state (the React virtual DOM algorithm[41] allows for intelligent partial re-rendering of only the components that have changed since the last render, improving performance).

4.5.2 Subscriptions

```
@subscribe
  onTimePassed 10 Tick ?<< not state.paused

update :: State -> Event -> State
update s e = match e with
  Tick -> cond
    | s.millis >= 99 -> {s | secs = s.secs + 1, millis = 0}
    | otherwise -> {s | millis = s.millis + 1}
  TogglePause -> {s | paused = not s.paused}
  Reset -> init
```

(a) The subscribe block and update definition in the stopwatch program, with each tick incrementing milliseconds or seconds



(b) The UI of the stopwatch program

Figure 4.16: An example stopwatch program using subscriptions

New states are sent to the `subscribe` handler, for example in the stopwatch program (Figure 4.16) we only wish to subscribe to time with the `onPassed` function if the `paused` field is true. Subscriptions can only be used if the state is a record type. Subscribe blocks are declared with `@subscribe`. Subscription functions are written on the left, with the predicate to the right of `?<<` determining if it is "switched on". The latest state is within scope as the variable `state`.

[41]https://reactjs.org/docs/reconciliation.html - accessed on 25/4/21

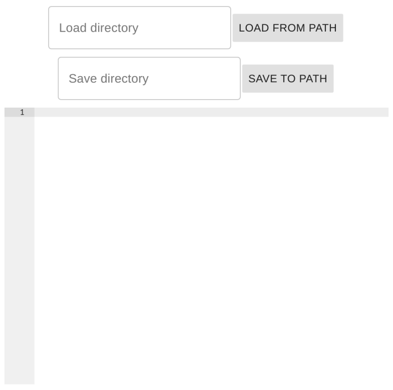
4.5.3 Side effects

```
type Event
   = LoadPressed
   | SavePressed
   | FileRead String
   | LoadUpdated String
   | SaveUpdated String
   | EditorTyped String

effect :: State -> Event -> Effect
effect s e = match e with
   LoadPressed -> readFile s.loadField FileRead
   SavePressed -> writeFile s.saveField
        s.contents
   _ -> NoEffect

update :: State -> Event -> State
update s e = match e with
   EditorTyped str -> {s | contents = str}
   LoadUpdated str -> {s | loadField = str}
   SaveUpdated str -> {s | saveField = str}
   FileRead str -> {s | contents = str}
   _ -> s
```

(a) The `effect` function performing side effects in the text editor, the file path string typed in the "Load directory" field is read and sent to the update function contained in a `FileRead` data type

```
type Event
   = LeftClick Int Int
   | RightClick Int Int
   | Reset
   | SeedRecieved Int
   | Dummy

initEffects :: [Effect]
initEffects = [requestSeed SeedRecieved]
```

(b) Side effects performed on startup by the Minesweeper program, requesting a random seed for subsequent pure random generations based on the seed



(c) The UI of the text editor, the text fields respectively generate `LoadUpdated` and `SaveUpdated` events when they are typed

Figure 4.17: Side effect facilities

Side-effects are only permitted from the optional definitions `effect` and `initEffects`. `effect` recieves the same event messages as `update` and can perform side-effects in response to these, e.g. Figure 4.17a. `initEffects` runs the list of side-effects on startup, useful in many situations for example Minesweeper (Figure 4.17b) where we just need to obtain a random seed from IO on startup and subsequent randomness can be achieved through a pure random generator without side effects.

Side effecting functions that retrieve data take a "reply handler" function used once the side effect is complete, for instance `readFile` takes the string path of the file and a function that takes a string argument and returns an Event type (containing the read file as a string) that is sent to the update function upon reading (Figure 4.17a).

5 Parser

Only select parts of the parser are explored, see Appendix C for the remaining description.

5.1 Preliminaries

5.1.1 Monadic contexts

```
pFun :: Parser Dfn
pFun = do
  lineNum <- fmap (unPos . sourceLine)
      getSourcePos
  name <- try pIdentifier
```

(a) An excerpt from the function definition parser showing retrieval of line number information from the context

```
indent :: Parser ()
indent = void $ do
  lvl <- asks tabLvl
  try $ do
    char '\n'
    count lvl $ char '\t'
    option () $ (char '\t') >> (fail "Wrong
        indentation level!")
  sc
```

(b) An excerpt from the indent parser, causing parsing to fail if there is an unexpected tab ahead

Figure 5.1: Using monadic functions inside a parser context

For the purposes of this paper, a monad can be thought of as a computational context a function works in which provides some facility such as error handling or state whilst abstracting away explicit arguments that would be required to achieve this functionality if no monadic context was used, essentially "hiding" something. This is useful in several situations, for instance in Megaparsec, functions are written in a parser context storing accessible information such as the current line number (Figure 5.1a) and the underlying stream of characters remaining to be consumed, and the ability to prematurely end parsing in the event of a failure (Figure 5.1b). The character stream being stored in the context is useful as we can thread together multiple parser functions and as they are working within the same context, any stream consumptions are handled by the context and passed to further function calls. Different monads can be combined to form one context, known in Haskell as a monad transformer stack; for instance, the type system requires both stateful and error functionality, so a stack consisting of the State[42] and Except[43] monads is used. The `return` function frequently appears in code excerpts, this just lifts it's argument into the current context; take note of the specific meaning despite superficial similarity with a "return" keyword in many languages.

5.1.2 Line folds

```
reallyLongList :: [String]
reallyLongList = ["the", "quick", "brown", "fox", "jumped",
    "over", "the", "lazy", "dog"]
```

Figure 5.2: Example of a line fold in a variable definition

A line fold (Figure 5.2) is a syntactic structure across multiple lines, where subsequent lines are indented after the "root" line

5.2 The Parser context in Megaparsec

The parser works in is the provided `ParsecT` monad transformer context (Figure 5.3), taking the type to represent errors, the type of the underlying character stream and a second monad that can be set to e.g. the State/Reader/Writer monad, combining to form one context. The Reader monad in essence is a convenient way to pass a hidden value to a function without having to provide it as an explicit argument. ParseEnv acts as an "environment" to execute function calls in, where the environment stores the relative level of indentation `tabLvl` that should be consumed (new line character followed by `tabLvl` number of tabs) at the beginning of any multiline syntactic structure, and whether the parser function is currently working within a line fold.

[42]https://wiki.haskell.org/State_Monad - accessed on 26/4/21
[43]https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html - accessed on 26/4/21

```haskell
type Parser = ParsecT ParseErr String (Reader ParseEnv)
--The environment the parser works in
data ParseEnv = ParseEnv {tabLvl :: Int, inLineFold :: Bool}
--My custom error type
data ParseErr = Misc String deriving (Show, Eq, Ord)
```

Figure 5.3: The parser context

```haskell
incLvlBy :: Int -> Parser a -> Parser a
incLvlBy increment p = local (\env -> env {tabLvl = (tabLvl env) + increment}) p

indent :: Parser ()
indent = void $ do
   lvl <- asks tabLvl
   try $ do
      char '\n'
      count lvl $ char '\t'
       notFollowedBy $ char '\t'
   sc
```

Figure 5.4: Modifying the underlying reader monad environment

The `local` function takes a function to modify the current environment and the function that should be run within that environment. `incLvlBy` (Figure 5.4) takes the amount the indentation level should be incremented by and adds this to the current indentation level, running the parser in this environment. `indent` shows the indentation level in the environment being queried using the `asks` function which takes a function that extracts some value from the environment data type, meaning we can parse exactly this number of tabs with `count` and ensure there is no leading tab.

5.3 Parser-combinators and regular expressions

It helps to instead think of parser-combinators as similar to regular expressions, in Megaparsec: `some`/`many` act like +/*, `lookAhead` and `notFollowedBy` are positive/negative look-ahead assertions and the alternative operator `<|>` represents an alternative like '|' in regexes.

```haskell
(char 'a' >> char 'b')<|> string "aa"
```

(a)

```haskell
(try $ char 'a' >> char 'b')<|> string "aa"
```

(b)

Figure 5.5: The unwanted consumption problem in parser-combinators

Parser combinators deviate from regular expressions in a significant way though: if a parser fails, the characters consumed leading up to it's failure (parsers can consist of several sub-parser functions as in Figure 5.5, where `char 'a' >> char 'b'` is itself a parser but so are `char 'a'` and `char 'b'`) are consumed from the stream (Think of >> as an operator that runs the monadic function on the left, discards the result, then runs the monadic function on the right). Consider a scenario where the contents of the stream is `['a','a',<eof>]` before running the alternative chain in Figure 5.5a. For the first parser to succeed, 'a' should be consumed followed by 'b', but it fails as there is another 'a' instead of a 'b' like it is expecting, however it will successfully consume the first 'a', making the stream now `['a',<eof>]`. The alternative chain detects this failure and runs the next parser in the chain (`string "aa"`), which will also fail as there is `['a',<eof>]` and not 'a','a' as it is expecting. The `try` primitive solves this issue; it takes as argument a parser, attempts to successfully run it and if it succeeds then returns the parsed data type with the corresponding characters consumed from the stream, but if the parser fails, the underlying stream will revert to its state before the parser was ran with the characters remaining unconsumed. So, if we now wrap our first parser in the alternative chain in a try (Figure 5.5b), then `string "aa"` will no longer fail.

Look-aheads are often used in parser-combinators. Figure 5.6a shows the first part of the variable identifier parser, ensuring the variable about to be parsed is none of the control keywords that appear in expressions (`choice` generalises the alternative `<|>` operator to a list). If this was not done, then something like `if fn0 arg0 arg1 then 1 else 2` would wrongly parse "then" and "else" as variable arguments to `fn0` and the if expression parser would fail as

```
notFollowedBy $ choice $ map pKeyword ["if",
    "else", "then", "match", "with", "let", "in",
    "cond", "pipeline", "is"]
```

(a) An excerpt from the variable identifier parser

```
blankLine = label "a blank line" $ do
   char '\n'
  inlineSc
  lookAhead $ choice [void $ char '\n', eof]
```

(b) The blank line parser

Figure 5.6: Look ahead functionality in parser-combinators

the "then" and "else" keywords would be missing from the stream, adding this negative lookahead check essentially tells the function argument parser where to stop. For parsing blank lines (Figure 5.6b), it is important that we don't consume trailing newline characters as this will interfere with parsing subsequent blank lines, by wrapping it in a `lookAhead` parser we check it's there without consuming it.

5.4 A more concrete example: parsing alias and function definitions

```
pAlias :: Parser Dfn
pAlias = inLf $ do
   lineNum <- fmap (unPos . sourceLine) getSourcePos
   --Once we see this keyword we can lock on to
       this branch
   try $ pKeyword "alias"
   name <- pCapsIdentifier
   lChar '='
   ty <- pType
   greedySc
   return $ Alias name lineNum ty
```

(a) `pAlias` definition

```
{-
alias/algebraic data type/record
    type/function/variable parser. this is the
    final result of parsing a file (the root parser
    is literally: many pDfn), a jumbled list of
    aliases, ADTs, records, functions and variables
    that must be organised after parsing
-}
pDfn
   = pAlias
   <|> pTypeDfn
   <|> pFun
```

(b) `pDfn` definition: an alternative chain containing `pAlias`

Figure 5.7: `pAlias` and `pDfn` parsers

A key concept in parser combinators is "locking on" to a branch. Consider Figure 5.7a, once we have seen the "alias" keyword we now know for certain that the leading text after should be a valid alias definition: a capitalised identifier, followed by an equals sign, followed by some type. We use `try` to ensure that we do not consume "alias" as a prefix from a function identifier. If no `try` was used and the file contained a function definition beginning with "`aliasOne :: String`", then the "alias" prefix would be wrongly consumed from the stream as the alias definition parser is first to run in the `pDfn` alternative chain, the function parser would see "`One :: String`" which is not a valid function definition as identifiers must begin with a lowercase character. This is a common pattern in all parsers, wrapping the "minimum pattern" that must be parsed to "lock on" to their respective branch in a `try` primitive, causing backtracking if a branch fails to "lock on". A whole do block can be an argument to a function, as in Figure 5.7a within a "line-fold" environment (`inLf` runs its parser argument in an environment where `inLineFold = True`).

5.5 Composability and reusability of parser-combinator functions

```
ptrns <- sepBy (char ':')pSimplePattern
```

(a) Cons pattern parser

```
exprs <- option [] $ (sepBy (char ',')pExpr)
```

(b) Comma seperated list of expressions parser

Figure 5.8: Reusability of sepBy

A benefit of parser-combinator functions is their composability and reusability, for example `sepBy` which takes "delimiter" parser in some syntactic structure along with an "element" parser of the syntactic structure, returning a list of the parsed elements. In Figure 5.8 `sepBy` is being used to parse a cons pattern of subpatterns separated by `':'`, and also being used to parse a comma separated list of expressions inside a list expression (`option` allows failure of the parser, returning the empty list argument in case of a failure). Other reusable parsers are `inParenths` which takes a parser and parses it surrounded by parentheses, and `betweenChars` which takes 2 character arguments representing the "start" and "end" of the structure, for example '{' and '}' for records, and the parser argument between these.

5.6 Finalising parse results with manual logic

```
return $ case exprs of
   [] -> Var "()"
   [expr] -> expr
   es -> Tuple es
```

(a) Manual logic in the parenthesised expression parser

```
return $ case frstPart of
   --List generator with no interval specified
   [x] -> ListGen x Nothing (last scndPart)
   --List generator with an interval specified
   x1:x2:_ -> ListGen x1 (Just x2) (last scndPart)
```

(b) Manual logic in list generator parser

Figure 5.9: Excerpts from parsers showing manual logic

Manual logic allows for flexibility in determining the final data type. In an excerpt of the parenthesised expression parser (Figure 5.9a), we parse a comma separated list of expressions knowing if there is one element then this is an expression wrapped in parentheses, if there is more than one element then this is a tuple expression and if the list of elements is empty then `"()"` represents the unit value. In the list generator parser (Figure 5.9b), if the first comma separated list before `".."` contains more than one element, then an interval has been specified (i.e. `[1,3..20]`) so we can extract this. If there is only one element (i.e. `[1..20]`), then no interval has been specified, so the run-time list generator assume an interval of 1.

5.7 Labelling and error messages

```
1:14:
  |
1 | if True then
  |             ^
unexpected end of input
expecting newline or the first if branch
expression
```

(a) Space consumer parser label is not hidden on input "if True then"

```
1:14:
  |
1 | if True then
  |             ^
unexpected end of input
expecting the first if branch expression
```

(b) Space consumer parser label is hidden on input "if True then"

```
1:17:
  |
1 | if fn0 arg0 arg1
  |                 ^
unexpected end of input
expecting "then" or a binary operator
```

(c) Default label for "then" on input "if fn0 arg0 arg1"

```
pIf = do
   try $ pKeyword "if"
   pred <- pExpr <?> "an if expression predicate"
   pKeyword "then"
   e1 <- label "the first if branch expression" $ pExpr
   pKeyword "else"
   e2 <- pExpr <?> "the second if branch expression"
   return $ IfEl pred e1 e2
```

(d) The if expression parser containing labels

Figure 5.10: Error messages

Labelling is how most error messages are created, either the `<?>` operator or `label` function is used (Figure 5.10d), taking the parser and the label string resulting in a new parser which behaves exactly the same except it is "labelled" in error messages if missing where it is expected. Often no labels are required as the default label is self-explanatory (Figure 5.10c). We can also *hide* a parser from being labelled using the `hidden` primitive; white-space is present practically everywhere in the file so an error message saying it is expecting "white-space/newline or <something>" is just unhelpful noise, therefore it should be hidden (Figure 5.10a, Figure 5.10b).

6 Type system

6.1 Introduction

Several implementations of Algorithm W exist in the literature (Grabmüller (2006), Jones (2000)). Syphon's system most closely follows the Cornell course[44] (notation also borrowed and slightly extended/formalized) as I find it provides the clearest conceptualisation: unique type variables are generated to represent the types of each expression/sub-expression, generating sets of constraints that denote equality between these types; essentially a large algebraic system that can be solved. The unification algorithm then analyses the constraints, breaking them down into their atomic forms to solve for the unknown type variables and substitute their occurrences in the unsolved constraints and the inferred type with their true values, meaning once unification ends the complete type of the root expression is found. The inference function works in a stateful/error context, holding a counter used for unique type variable identifiers incrementing each time a fresh variable is generated, and also provides the ability to prematurely end inference with the `throwError` primitive. See Appendix D for the full type system description.

6.2 Preliminaries: types as function-like expressions

All types are in-fact type constructors applied to zero or more type constructors, including function types where the true representation of $\mathtt{a} \rightarrow b$ is in fact $((\rightarrow)\ \mathtt{a})\ \mathtt{b}$ with the arrow constructor applied to 2 "curried" type arguments.

6.3 Inference

6.3.1 Notation and example

$$\frac{\Gamma \vdash subexpr_0 :: t_0 \dashv C_0 \qquad \Gamma \vdash subexpr_1 :: t_1 \dashv C_1 \qquad \ldots \qquad \Gamma \vdash subexpr_n :: t_n \dashv C_n}{\Gamma \vdash e :: t \dashv C_e \cup C_0 \cup C_1 \cup \ldots \cup C_n}$$

α_n denotes a freshly generated unique type variable, t_n represents some arbitrary type, and $t_0 \doteq t_1$ represents an individual constraint (i.e. t_0 and t_1 must be equal). The long horizontal line does not have the traditional "condition" meaning from proof notation, rather it is a neat way of showing the type and constraints generated from recursively inferring sub-expressions, results of environment look-ups may also be written atop this line bound to variables. This notation reads as: within the typing environment Γ, the expression e has type t, as long as the set of constraints it generates (C_e) unioned with the sets of constraints generated by inferring any child sub expressions e may contain can be solved by the unification algorithm. Γ consists of 3 sections: a "standard" section for storing variable/type bindings, a section for storing expected field types of records and a section for storing expected field types of optional structures. To demonstrate the notation I explain the rule for inferring the type of an `if` expression:

TyIf

$$\frac{\Gamma \vdash pred :: t_0 \dashv C_0 \qquad \Gamma \vdash e_0 :: t_1 \dashv C_1 \qquad \Gamma \vdash e_1 :: t_2 \dashv C_2}{\Gamma \vdash \mathtt{if}\ pred\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1 :: \alpha_0 \dashv \{t_0 \doteq \mathtt{Bool}, t_1 \doteq \alpha_0, t_2 \doteq \alpha_0\} \cup C_0 \cup C_1 \cup C_2}$$

The set of constraints C_e that the root expression generates is $\{t_0 \doteq \mathtt{Bool}, t_1 \doteq \alpha_0, t_2 \doteq \alpha_0\}$; the predicate expression clearly must be of type `Bool`, and both branches of the if expression must have the same type. By first generating α_0 as our assumption for the complete type of the if expression, we can recursively infer the types of the sub-expressions e_0 and e_1 to form the essential constraints $t_1 \doteq \alpha_0$ and $t_2 \doteq \alpha_0$, communicating to the unification stage that these types should match. We must consider the possibility that sub-expressions generate their own constraints, hence we union all of C_e, C_0, C_1, and C_2 as our overall set of constraints. For example if there is a nested `if` expression inside e_0 or e_1, it would generate a set of constraints stating that it's predicate expression has type `Bool` and it's 2 "branch" expressions are the same type; the nesting of expressions is arbitrarily deep so this is essential.

This notation is used as it is a concise pseudo-code capturing the important inference behaviour, whilst abstracting away irrelevant details of the Haskell implementation. The equivalence with Haskell code should be apparent though, here is an excerpt of the `infer` function for handling the `if` expression case:

[44] https://www.cs.cornell.edu/courses/cs3110/2021sp/textbook/interp/inference.html - accessed on 27/4/21

```
--Inside a case expression which applies the different typing rules depending on the expression detected
IfEl pred e1 e2 -> do

   --Generating a new, unique type variable (α0 in the notation)
  newTyVar <- genFreshVar

   --Recursively infer the 3 sub expressions, obtaining their types and constraints
  (predTy, predConstraints) <- infer (If1st:path) env pred
  (e1Ty, e1Constraints) <- infer (If2nd:path) env e1 --"env" is our environment Gamma from notation
  (e2Ty, e2Constraints) <- infer (If3rd:path) env e2

   --Our overall set of constraints that the entire if expression will generate
  let ifExprConstraints
    = unionConstraints [
        S.fromList [Eq predTy boolTy, Eq newTyVar e1Ty, Eq newTyVar e2Ty],
        predConstraints,
        e1Constraints,
        e2Constraints
    ]

   --Return inferred type and overall set of constraints generated
  return (newTyVar, ifExprConstraints)
```

The "path" argument to `infer` allows the inference algorithm to build up a "path" in the parse tree so that any sub-expressions detected as needing to be modified after inference can be found and replaced (Appendix D).

6.3.2 Literal types and simple variables

The rules for literal values are trivial; no constraints that must hold for an `Int` to be an `Int`, it *just is* an `Int`. Literal Ints, Doubles, Bools, Chars and Strings are represented as i, d, b, c and s respectively. The rule for looking up variables is a case of determining if there exists a binding of the variable name with some type within the standard section of Γ, and then inferring the type as the lookup result. Notice the $^!$ notation: clearly a lookup may fail if no binding exists in Γ. Henceforth $^!$ represents a function that may fail causing inference to end with an appropriate error message.

TyInt
$$\Gamma \vdash i :: \texttt{Int} \dashv \varnothing$$

TyDouble
$$\Gamma \vdash d :: \texttt{Double} \dashv \varnothing$$

TyBool
$$\Gamma \vdash b :: \texttt{Bool} \dashv \varnothing$$

TyChar
$$\Gamma \vdash c :: \texttt{Char} \dashv \varnothing$$

TyString
$$\Gamma \vdash s :: \texttt{String} \dashv \varnothing$$

TyVar
$$\frac{t_0 = \texttt{lookup}^!(\Gamma, v)}{\Gamma \vdash v :: t_0 \dashv \varnothing}$$

6.3.3 Application

$t_a \rightarrow t_b$ denotes an arrow type. e_0 takes an expression with e_1's type and returns an expression with the type of our assumption α_0, hence we know that t_0 is actually an arrow type. By equating this knowledge in a constraint, the unification algorithm will be able to break down these types and solve for α_0.

TyApp
$$\frac{\Gamma \vdash e_0 :: t_0 \dashv C_0 \qquad \Gamma \vdash e_1 :: t_1 \dashv C_1}{\Gamma \vdash e_0\ \ e_1 :: \alpha_0 \dashv \{t_0 \doteq t_1 \rightarrow \alpha_0\} \cup C_0 \cup C_1}$$

6.3.4 Conditional expressions

The conditional expression rule generalizes the `if` expression rule for any number of cases.

TyCond
$$\frac{\Gamma \vdash p_0 :: t_{a0} \dashv C_{a0} \qquad \Gamma \vdash e_0 :: t_{b0} \dashv C_{b0} \qquad \dots \qquad \Gamma \vdash p_n :: t_{an} \dashv C_{an} \qquad \Gamma \vdash e_n :: t_{bn} \dashv C_{bn}}{\begin{array}{c}\Gamma \vdash \texttt{cond}\ \ |\ \ p_0 \rightarrow e_0\ \ \dots\ \ |\ \ p_n \rightarrow e_n :: \alpha_0 \dashv \{t_{ai} \doteq \texttt{Bool}, i \in \{0 \dots n\}\} \cup \{t_{bi} \doteq \alpha_0, i \in \{0 \dots n\}\} \\ \cup C_{a0} \cup C_{b0} \cup \dots \cup C_{an} \cup C_{bn}\end{array}}$$

6.3.5 Optional structures

For optional structures we must ensure all expressions given for each field have their expected types. `lookupOpt` queries the optional section of the environment, obtaining a map of fields and expected types. $m \leq n$ due to the fact any number of fields may be overriden. As with other environment lookups `lookupOpt` may fail, for example the variable being looked up does not have an optional structure associated with it.

$$\text{TyOpt}\quad \frac{\Gamma \vdash e_0 :: t_{a0} \dashv C_0 \qquad \ldots \qquad \Gamma \vdash e_m :: t_{am} \dashv C_m \qquad \{field_0 :: t_{b0}, \ldots, field_n :: t_{bn}\} = \texttt{lookupOpt}^!(\Gamma, x) \qquad t_x = \texttt{lookup}^!(\Gamma, x)}{\Gamma \vdash x\ \ \#\{field_0 = e_0, \ldots, field_m = e_m\} :: t_x \dashv \{t_{ax} \doteq t_{by}, field_x == field_y\ \ x, y \in \{0 \ldots m\}\} \cup C_0 \cup \ldots \cup C_m}$$

6.4 Unification

We again work in a context for convenience of handling failures, making use of the Except monad.

```
unify :: Env -> [Constraint] -> UnificationContext [Substitution]
unify env [] = return []
unify env [Eq tv1@(TyVar {}) tv2@(TyVar {})]
  | tv1 /= tv2 = throwError $ "No unification possible!"
unify env (Eq (TyApp te1L te2L) (TyApp te1R te2R):constrs) =
  unify env $ Eq te1L te1R:Eq te2L te2R:constrs
```

`Eq` is the ADT used to represent constraints. If there is one constraint left, but it involves 2 type variables that are different, then clearly unification has failed as these types do not match after all substitutions have been applied. If we come across a constraint equating 2 type applications, this can be broken down into 2 separate constraints between the "function" and "argument" type expressions.

6.4.1 Type variables

```
unify env cstrs@(Eq tv@(TyVar {}) ty:constrs)
  | ty == tv = unify env constrs
  | ty `containsTyVar` tv = throwError $ "Infinite type error! Constraints are: " ++ show cstrs
  | otherwise = let subst = (tv, ty) in
    (:) (tv, ty) <$> unify env (map (\c -> applySubstToConstraint c subst) constrs)
unify env cstrs@(Eq ty tv@(TyVar {}):constrs)
  | ty == tv = unify env constrs
  | ty `containsTyVar` tv = throwError "Infinite type error!"
  | otherwise = let subst = (tv, ty) in
    (:) (tv, ty) <$> unify env (map (\c -> applySubstToConstraint c subst) constrs)
```

If a constraint states 2 type variables with the same identifier are equal, it is ignored because it does not provide any new information. If the other type is not a type variable but contains the same type variable that is being equated, then this is known as an infinite type error and must fail (recall it is just an algebraic equation, so if it is in terms of itself then clearly there is no solution). If all these checks pass then we have found a solution for the type variable as it is equated with a constructor and not another type variable, so we can substitute all occurences of `tv` with `ty` in the remaining constraints, and also add this to our substitution list that we build up and return (`(:)` <$> lifts cons into the unification context).

```
unify env (Eq c1@(Constr {}) c2@(Constr {}):constrs)
  | c1 == c2 = unify env constrs
  | otherwise = throwError $ "No unification possible, different constructors! Constructors: " ++ show c1
     ++ " , and: " ++ show c2
```

Unifying type constructors is trivial, we check that they are the same. Clearly (`Constr "Int"`) and (`Constr "Bool"`) should *not* be unified.

```
unify env (Eq EVENT t2:constrs) = unify env (Eq (event env) t2:constrs)
unify env (Eq STATE t2:constrs) = unify env (Eq (state env) t2:constrs)
unify env (Eq t1 EVENT:constrs) = unify env (Eq t1 (event env):constrs)
unify env (Eq t1 STATE:constrs) = unify env (Eq t1 (state env):constrs)
```

The state type and event type are stored in the environment, allowing for system primitive definitions to ensure the state and event types they manipulate match.

```
unify env (Eq t1 t2:constrs)
   | t1 == t2 = unify env constrs
   | otherwise = throwError $ "No unification possible, different types! Types: " ++ show t1 ++ " , and: "
      ++ show t2
```

The final case catches any rare edge cases, for instance a `TyApp` equated with a `Constr` clearly has no solution.

7 Transpilation and run-time behaviour

7.1 Introduction

```
export const $add = x => y => x + y
export const $sub = x => y => x - y
export const $mul = x => y => x * y
export const $div = x => y => x / y
export const $mod = x => y => x % y

export const $and = arg0 => arg1 => arg0 && arg1
export const $or = arg0 => arg1 => arg0 || arg1
```

Figure 7.1: Curried versions of math and boolean operators

The introduction of arrow functions in ES6[45] means conveying the syntax and semantics of a lambda-calculus based language within JavaScript is straightforward. GHC2JS and Purescript, 2 of the most prominent projects for transpiling Haskell-like syntax to JavaScript do not make use of arrow functions in their transpiled code, presumably because when they first appeared ES6 syntax was not widely adopted. The situation is different nowadays where it is natively supported on a wide range of platforms including NodeJS which powers Syphon's runtime, therefore *all* functions are transpiled as arrow functions to achieve the currying behaviour of a Haskell-like language. For example, curried functions have been made from JavaScript's basic operators (Figure 7.1) that are used in place of them. Notice how identifiers are prepended with '`$`', the developer is unaware that '`+`' in Syphon is transpiled as `$add`, so if no dollar is used then a developer's definition of `add` would clash with this. As a general rule, whenever the name of something is changed it is prepended with a dollar to prevent global namespace pollution.

7.2 Indentation

```
--A case of the match expression in writeExpr
IfEl pred e1 e2 -> "(() => {if(" ++ writeExpr lvl pred mod ++ "){\n" ++
    tabs (lvl + 1) ++ "return " ++ writeExpr lvl e1 mod ++ "\n" ++
    tabs lvl ++ "} else {\n" ++
    tabs (lvl + 1) ++ "return " ++ writeExpr lvl e2 mod ++ "\n" ++
    tabs lvl ++ "}})()\n"
```

(a) Transpiling `if` expressions, where the contents in curly braces have one level of indentation greater

```
writeEffectFn :: Dfn -> Module -> String
writeEffectFn (Fun name lineNum _ cases wheres) mod =
   "const effect = ($arg0,$arg1) =>{\n\
   \\tconst $name = \"effect\"\n" ++
   (if length cases == 1 then writeCaseless 1 (cases !! 0) mod
      else concatMap (\cse -> writeCase 1 cse mod) cases) ++
   "}\n"
```

(b) Multi-line strings in Haskell allowing a "visualization" of the transpiled result

Figure 7.2: Demonstrating indentation in the transpiler

Due to the fact that Javascript's syntax can be conveyed purely through curly braces and semicolons, inserting tabs and newline characters is not necessary at all; the entire transpiled file could be written on one line which is Elm's approach. For the purposes of demonstration and my own understanding of transpiled code, I made an effort to make the transpiled code as readable as possible for example ensuring consistent relative indentation; all transpiler functions take a `lvl` argument to enable this (Figure 7.2a). Functions are layed out so they give a sort of "visualization" of the transpilation of a single string, multiline strings (before and after '`\`') help with this (Figure 7.2b).

[45]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions - accessed on 27/4/21

7.3 Algebraic data types

```
--Corresponding Syphon code
type Event = CellClicked Int Int | Flagged Int Int


//Transpiled Javascript code
class CellClicked {constructor(arg0,arg1){this.val0 = arg0, this.val1 = arg1}}
const _CellClicked = arg0 => arg1 => new CellClicked(arg0,arg1)

class Flagged {constructor(arg0,arg1){this.val0 = arg0, this.val1 = arg1}}
const _Flagged = arg0 => arg1 => new Flagged(arg0,arg1)
```

Figure 7.3: Comparing algebraic data types in Syphon/Javascript syntax, excerpts from the Minesweeper event types

Javascript class constructors are not curried, however this is easily achieved by creating corresponding curried functions for each class that is transpiled as in Figure 7.3. This is essential as the semantics of Syphon is based on the lambda calculus like Haskell and other functional first languages where every function must be curried, if the constructor functions were not curried then something like `map (CellClicked 0)[1,2,3]` resulting in `[CellClicked 0 1,CellClicked 0 2, CellClicked 0 3]` would not be possible. Doing this eliminates the need for "new" syntax in expressions making code more concise.

7.4 Application

```
add 1 2
```

(a) Syphon syntax

```
add(1)(2)
```

(b) ES6 arrow function syntax

Figure 7.4: A simple function application expression

JavaScript arrow functions behave semantically the same as Syphon, except space application syntax is replaced with multiple parentheses (Figure 7.4).

7.5 Records

```
//Syphon: type Person = {name :: String, age :: Int}

//Constructors
class Person extends Record({name : undefined, age : undefined}){}
const _Person = arg0 => arg1 => new Person({name: arg0, age: arg1})

//Field accessing, same as Syphon syntax
person1.age

//Field updating, set for one field or mergeDeep for multiple
person1.set("age", 10)
person1.mergeDeep({name : "new name", age : 20})
```

Figure 7.5: Immutable JS record syntax

Records are implemented with ImmutableJS Records [46] as it is a mature, well-maintained library with good performance (Figure 7.5).

7.6 Imports

All Syphon library imports are hardcoded, the developer's code is transpiled into a single file with these.

[46]https://github.com/immutable-js/immutable-js - accessed on 27/4/21

7.7 Assets

```
asset flag = "./lib/GUI/assets/flag.svg"
asset mine = "./lib/GUI/assets/mine.png"
asset explosion = "./lib/GUI/assets/explosion.png"
```

(a) Asset declarations in Minesweeper

```
(Graphic $ flag #{dim = (40,40)}
```

(b) An excerpt of the `view` function in Minesweeper referencing the asset variable

For GUI design, it is sensible to include a convenient facility for importing graphics, in Syphon the developer gives `asset` definitions for this where the image variables can be used in widgets (Figure 7.6a, Figure 7.6b). This is fairly harmless IO as it occurs only once upon startup, and the program will fail to compile with React showing an error message if the image does not exist at the path.

7.8 React components as data types

```
//React text editor implementation
export const _TextEditor = value => minLines => maxLines => onType =>
  <AceEditor theme = "monokai" name = "EDITOR" value = {value}
  onChange = {(newString,_) => window.$dispatchRef(onType(newString))}
  editorProps={{ $blockScrolling: true }} minLines = {minLines} maxLines = {maxLines}/>
```

```
{-
Corresponding Syphon constructor, taking the string to display,
the minimum number of lines, the maximum number of lines, and
the onType response handler function
-}
TextEditor String Int Int (String -> Event)
```

Figure 7.7: Correspondance between React components and Syphon widget data types

React makes use of JSX syntax for declaring GUI components [47]. JSX can be wrapped in curried functions (Figure 7.7), allowing GUI components to be treated as normal Syphon data types.

7.9 Optional structures

```
Syphon: Container #{dim = (275,35), bgColor = Grey}
JavaScript: _Container({dim : [275,35], bgColor : "#A9A9A9"})

Syphon: Text s.display
JavaScript: _Text({})(s.display)
```

Optional structures are implemented as standard JavaScript objects. The transpiler checks if any variable is an optional structure, and if so then includes an additional `({})` argument (recall optional structures can be omitted so we must check). If optional fields are declared then they are placed inside the object argument, and widgets will override any default fields with declared fields upon construction.

7.10 The pattern extraction algorithm

Pattern information must be extracted into C-style syntax, with predicates and bindings separated and used for if statement predicates and variable declarations respectively. The algorithm takes some "source" expression (transpiled as a string) that it will extract predicates and variable bindings from, recursively calling the function on any sub-patterns and modifying the "source" expression string to represent the relationship between the source pattern and the subpattern. For instance, suppose the pattern of the first argument of a function is `"MouseMoved y x"`, the source string will initially be `"$arg0"`, so a predicate `"$arg0 instanceof MouseMoved"` will be extracted from the root pattern, however when the function is called on the subpatterns `y` and `x` the source string needs to be changed:

[47]https://reactjs.org/docs/introducing-jsx.html - accessed on 26/4/21

it becomes `"$arg0.val"<field index>`, in this example the respective source strings in each recursive call will be `"$arg0.val0"` and `"$arg0.val1"`, making the final result the bindings `("y","$arg0.val0")` and `("x","$arg0.val1")`, and the predicate `$arg0 instanceof MouseMoved`.

```
--Minimum that never goes below zero
min :: Int -> Int -> Int
min 0 _ = 0
min _ 0 = 0
min x y = if x < y then x else y
```

(a) Simple Syphon function with patterns

```
([],["$arg0 === 0"])
([],["$arg1 === 0"])
([("x","$arg0"),("y","$arg1")],[])
```

(b) Bindings and predicates extracted from the simple function

```
const min = arg0 => arg1 => {
   if($arg0 === 0){
      return 0
   }
   if($arg1 === 0){
      return 0
   }
   if(true){
      const x = $arg0
      const y = $arg1
      return (() => {
         if($lt(x)(y)){
            return x
         } else {
            return y
         }
      })()
   }
}
```

(c) The transpiled version of the simple function

```
if(true){
   return(() => {
      if($lt($arg0)($arg1)){
         return $arg0
      } else {
         return $arg1
      }
   })()
}
```

(d) Using direct substitutions instead of variable declarations

Figure 7.8

A more concrete example is provided in Figure 7.8a, the first 2 cases of the function involve only predicates whereas the last case involves only variable bindings, there are no conditions that must hold for the values of the first and second arguments to be bound to the variables `x` and `y`. The pattern extraction algorithm returns a tuple of lists of variable bindings and predicates, as seen in Figure 7.8b. Once this information is extracted from the patterns, it can then be transpiled as variable declarations and predicates of if statements as in Figure 7.8c. An alternative approach that would have avoided the need for these variable declarations would be to substitute any occurrences of variables in expressions with their actual values, meaning the final case in the `min` function would look something like Figure 7.8d, however as the variable values extracted from patterns grow more complex this would make the transpiled code harder to reason about, so for the purposes of readability and debugging, variables with their corresponding Syphon names are declared so that the parallels between the Syphon code and JavaScript code can be clearly seen. If there are no predicates in a case, then `true` is used. If there are multiple predicates, these are encapsulated in parentheses and aggregated with `&&` operators.

7.11 Anonymous JavaScript functions

```
(() => {
   if(predicate){
      //const x = predicate.val0
      //Match expressions put bindings here ^^

      return <body expression>
   }
})()
```

Figure 7.9: How anonymous javascript functions are used

Anonymous functions allow converting arbitrary JavaScript code into a structure that can be reasoned about as an expression evaluating to a value. `match`, `cond` and `if` expressions all transpile to this form with if statements used for checking predicates and match expressions introducing variable declarations (Figure 7.9). `let` expressions are also implemented with anonymous functions, however they just introduce variable bindings into an isolated scope without a predicate as is the behaviour of let expressions. An alternative would be nested ternary operators, however these do not allow declaring an isolated variable scope; furthermore, my own investigations found them to be slower (Appendix A).

7.12 Syphon system primitive definitions in React

```
export const App = () => {
   const [state,$dispatch] = React.useReducer(update,init)
   window.$dispatchRef = $dispatch
   return view(state)
}
```

(a) The React component of the application, where the view function renders the UI based on the dynamically changing state

```
export const useSyphon = (updateFn,initState,effectFn) => {
   const [state, dispatch] = React.useReducer(updateFn, initState)
   //First send the event to the effect function and then the update function
   const effectfulDispatch = event => {
      effectFn(state, event)
      dispatch(event)
   }
   return [state, effectfulDispatch]
}
```

(b) The `useSyphon` hook, used instead of `useReducer` if a definition for `effect` is detected

Figure 7.10: Hooks in React

React provides a `useReducer` hook implementing the MVU pattern, taking the initial state and the update function (known as a "reducer" in React) and returning a 2 element array of the state variable that dynamically changes when updated and a function to send event messages to the `update` function (`$dispatch`) (Figure 7.10a). `$dispatch` is immediately placed in a `window` global variable so that any part of the program can reference it to send event messages to `update` like buttons. Widgets within the view function reference this dynamically changing state variable the same as in Syphon code, and the React virtual DOM algorithm optimially re-renders components by detecting which sub-components are affected by which changes inside the state data type.

`useSyphon` (Figure 7.10b) is a custom hook I created that uses an underlying `useReducer` hook, but also sends all event messages to the `effect` function aswell as the `update` function, used if a definition for `effect` is detected.

7.13 Subscriptions

```
@subscribe
   onTimePassed tickFreq Tick ?<< True
   onKeyPressed UpArrow RightPlayerUp ?<< True
   onKeyPressed DownArrow RightPlayerDown ?<< True
   onKeyReleased UpArrow RightPlayerRelease ?<< True
   onKeyReleased DownArrow RightPlayerRelease ?<< True
   onKeyPressed (KeyChar 'w') LeftPlayerUp ?<< True
   onKeyPressed (KeyChar 's') LeftPlayerDown ?<< True
   onKeyReleased (KeyChar 'w') LeftPlayerRelease ?<< True
   onKeyReleased (KeyChar 's') LeftPlayerRelease ?<< True
```

(a) The declared subscriptions in the Pong Syphon program

```
React.useEffect(() => {
   let $time = undefined
   if(true){$time = setInterval(() => $dispatch(_Tick()),tickFreq)}
   return () => clearInterval($time)
},[])
React.useEffect(() => {
   function $handler(e){
      if(e.code === $keyExprToCode(_KeyChar('s'))){$dispatch(_LeftPlayerDown())}
      if(e.code === $keyExprToCode(_KeyChar('w'))){$dispatch(_LeftPlayerUp())}
      if(e.code === $keyExprToCode(_DownArrow())){$dispatch(_RightPlayerDown())}
      if(e.code === $keyExprToCode(_UpArrow())){$dispatch(_RightPlayerUp())}
   }
   if((true)){document.addEventListener("keydown",$handler)}
   return () => document.removeEventListener("keydown",$handler)
},[])
React.useEffect(() => {
   function $handler(e){
      if(e.code === $keyExprToCode(_KeyChar('s'))){$dispatch(_LeftPlayerRelease())}
      if(e.code === $keyExprToCode(_KeyChar('w'))){$dispatch(_LeftPlayerRelease())}
      if(e.code === $keyExprToCode(_DownArrow())){$dispatch(_RightPlayerRelease())}
      if(e.code === $keyExprToCode(_UpArrow())){$dispatch(_RightPlayerRelease())}
   }
   if((true)){document.addEventListener("keyup",$handler)}
   return () => document.removeEventListener("keyup",$handler)
},[])
```

(b) The subscriptions transpiled as `useEffect` hooks, the dependency arrays here are empty unlike the stopwatch example as we never want the subscriptions to stop, also notice the Syphon predicate `True` is transpiled to the JavaScript `true` predicate in the if statements

Figure 7.11: Subscription demonstration

Subscriptions are implemented with the `useEffect` hook in React, allowing a component to call side effects upon re-rendering. `useEffect` takes a dependency array argument containing the variables that should be observed for changes, for example in the aforementioned Stopwatch example we only want to stop or start the time subscription when the `paused` field is toggled. By limiting subscriptions to be used with only record states, the parse tree of each subscription predicate can be searched for the state fields that are referenced which can then populate the `useEffect` dependency array. Subscription generating functions in the left part of a subscribe block are not true functions, at present there is only `onTimePassed`, `onKeyPressed`, `onKeyReleased` and `onMouseMoved`; these explicit "functions" are only what must appear on the left, meaning the name of the subscription generating function can be determined by analysing the subscription definition cases and specific behaviour takes place for each. For example, JavaScript supports having only a single listener for key presses and key releases, meaning each of the separate `onKeyPressed` and `onKeyReleased` definitions in Syphon are aggregated into a single event handler function (Figure 7.11).

7.14 Reasoning with "fake" values

```
React.useEffect(() => {
  $_Cons(requestSeed(_SeedRecieved))($_EmptyList())
}, [])
```

Figure 7.12: A list of effect "values" that are evaluated and performed upon startup

Subscriptions and side-effects are of course not values at all, however we can treat them as such to fit in with the transpiler and type system pipeline. Having `Effect` and `Subscription` types allows the type system to ensure correctness for subscribe blocks and `Effect` return "values" in `effect` and `initEffect`. The list of `Effect` "values" in `initEffect` is transpiled as a regular `$Cons` list declared in `useEffect` (Figure 7.12); although it looks as if we are just declaring a list, as the list is created each elements is evaluated, meaning `requestSeed` will be called and send it's response to the `update` function, the final evaluation result will be a cons list filled with `undefined` which is then garbage collected.

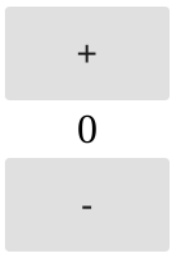
8 Evaluation

8.1 Introduction

A unit testing language SUnit has been created for testing the logic of Syphon programs. The MVU pattern enforces separation of most application logic (namely the `update` function) from the UI making unit testing isolated logic straightforward. To write unit tests, create a corresponding "`.sunit`" file containing test cases for each function (Figure 8.1c). Each case states the arguments and either the pattern that should match the return value (`shouldReturn`), a predicate function that the return value should satisfy (`shouldSatisfy`), or the expected error message for the test case (`shouldFailWith`). Declare test cases for a function with <name>, followed by an indented block of case definitions. The transpiler transpiles the source code as normal, but a function `$runSUnit` is included which can be called to run the unit tests (Figure 8.1d). The same transpiler infrastructure is used for SUnit, namely the pattern extraction algorithm where `shouldReturn` patterns are transpiled to a predicate of the return value. Each time a function fails with an error, it is added to a globally accessible `$errLog` list, in `shouldFailWith` clauses SUnit removes the latest error and checks if it equals the `shouldFailWith` string. The full source code and tests for each program can be found in Appendix *** . I will introduce a simple program (Figure 8.1) to show evaluation:

```
alias State = Int

type Event = Inc | Dec

init :: State
init = 0

update :: State -> Event -> State
update s e = match e with
   Inc -> s + 1
   Dec -> s - 1

view :: State -> Widget
view s = Column []<<
   Button (Text "+") Inc
   Text $ toString s
   Button (Text "-") Dec
```

(a) A complete program, IncrementCounter.sy



(b) The view of the simple application

```
#update

   args: 0 Dec
   shouldReturn: (-1)

   args: 0 Inc
   shouldReturn: 1

   args: (-1) Inc
   shouldReturn: 0

   args: 1 Inc
   shouldReturn: 2
```

(c) IncrementCounterTests.sunit

```
--------------------------------------------------
RUNNING 4 TESTS FOR FUNCTION: update
--------------------------------------------------
1:
   Inputs = 0 Dec, Expected return value: (-1)
   SUCCESS: function returned expected value.
2:
   Inputs = 0 Inc, Expected return value: 1
   SUCCESS: function returned expected value.
3:
   Inputs = (-1) Inc, Expected return value: 0
   SUCCESS: function returned expected value.
4:
   Inputs = 1 Inc, Expected return value: 2
   SUCCESS: function returned expected value.
--------------------------------------------------
TESTING COMPLETE
--------------------------------------------------
4 total tests ran
4 successes
0 failures
```

(d) NodeJS running the SUnit tests

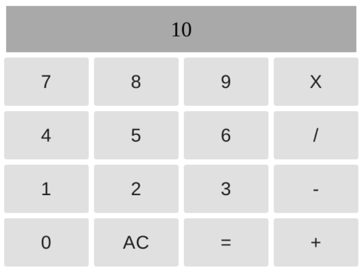
Figure 8.1: Evaluation of a simple program

Notice how some of the boilerplate from the equivalent Elm example *** has been removed, such as no empty list arguments, no main function and inclusion of a list block for more declarative code.

8.2 Calculator

```
type Event = Clear | Eq | OpPress Operator | Digit Int

view :: State -> Widget
view s = Column []<<
  Container #{dim = (265,35), bgColor = Grey} $ Text s.display
  Row $ (map mkDigitButton [7..9]) ++ [Button (Text "X") (OpPress (*))]
  Row $ (map mkDigitButton [4..6]) ++ [Button (Text "/") (OpPress (/))]
  Row $ (map mkDigitButton [1..3]) ++ [Button (Text "-") (OpPress (-))]
  Row [mkDigitButton 0, Button (Text "AC") Clear,
     Button (Text "=") Eq, Button (Text "+") (OpPress (+))]
  where
     mkDigitButton i = Button (Text $ toString i) (Digit i)
```

(a) The view function and event types of the calculator



(b) The UI of the calculator application

```
#update

   args: (update (State "" (+) 0) Clear) Clear
   shouldReturn: (State "" fn 0)

   args: (State "1" (+) 2) Eq
   shouldReturn: (State "3" fn 0)

   args: (State "1" (-) 2) Eq
   shouldReturn: (State "1" fn 0)

   args: (State "3" (*) 4) Eq
   shouldReturn: (State "12" fn 0)

   args: (State "5" (/) 5) Eq
   shouldReturn: (State "1" fn 0)

   args: (State "1" (+) 0) (Digit 2)
   shouldReturn: (State "12" fn 0)

   args: (update (State "1" (+) 0) (Digit 2)) (OpPress (+))
   shouldReturn: (State "" fn 12)
```

(c) CalculatorTests.sunit

Figure 8.2: Calculator code summary and view

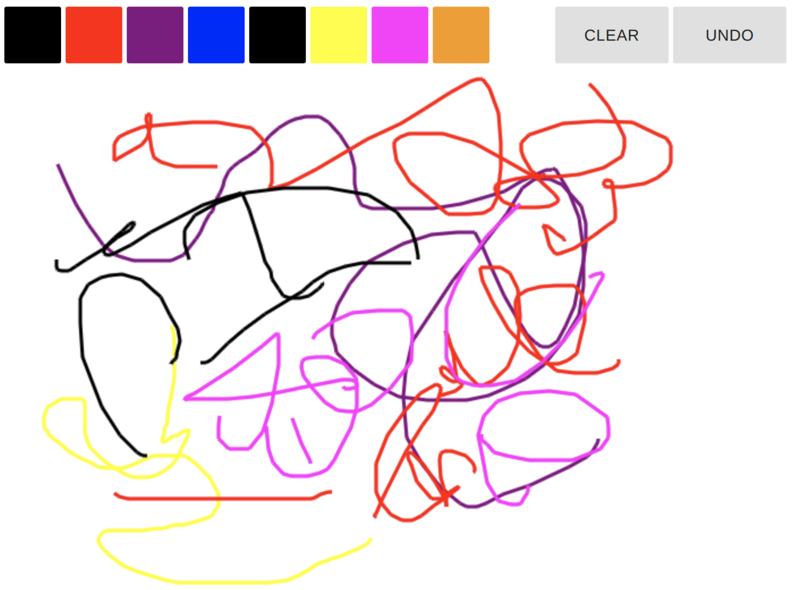
The calculator is a good example where the declarative nature of functional programming shines, with the operand buttons themselves storing math functions. The state stores the first operand and operator function, when equals is pressed the display is parsed for the second operand and the operator is applied. List blocks allow clear reasoning of each "row" of the GUI. The calculator unit tests entail a sort of "proof" (Figure 8.2c); if we are certain that `Digit` events cause the display to append the digit, then in the last case we can be sure that `OpPress` events cause the display to be cleared as we definitely know it was not empty before, this demonstrates the power of treating application states as pure data types.

8.3 Paint

Colours are treated as data types themselves, eliminating any sort of incorrect hex code error; the transpiler converts the data types to their corresponding hex strings. The canvas data type takes a list of shapes and draws the latest list on every re-render, so whenever the user "undos" a stroke or draws a new stroke, it just needs to be dropped from/added to the shape list in the state and this change is instantly observable in the next render. The bulk logic for an entire paint program is implemented in 29 lines of simple code, demonstrating Syphon's declarative

```
update :: State -> Event -> State
update s e = match e with
   PixelEntered y x -> {s | strokes = addPointToStrokes y x s.strokes}
   MousePressed -> {s | dragging = True}
   MouseReleased -> {s | dragging = False, strokes = emptyStroke s.color:s.strokes}
   ColorChanged c -> {s | strokes = updateColor c s.strokes, color = c}
   Clear -> init
   Undo -> {s | strokes = safeDrop s.strokes}
   Dummy -> s
   where
      addPointToStrokes y x ((Stroke c weight tups):strokes) =
         ((Stroke c weight ((y,x):tups)):strokes)
      updateColor c ((Stroke _ weight tups):strokes) = Stroke c weight tups:strokes
      safeDrop list = match list with
         [empty] -> [empty] --always ensure there is at least one empty stroke
         empty:_:rest -> empty:rest

view :: State -> Widget
view s = Column []<<
   Row $ (++) $<<
      map mkColorButton [Black, Red, Purple, Blue, Green, Yellow, Pink, Orange, White]
      [txtButton "Clear" Clear, txtButton "Undo" Undo]
   Canvas $<<
      #{onMouseMoved = if s.dragging then PixelEntered else \_ _ -> Dummy,
         onMousePressed = \_ _ -> MousePressed, onMouseReleased = \_ _ -> MouseReleased}
      700
      500
      s.strokes
   where
      mkColorButton c = Button #{bgColor = c, dim = (50,50)} DummyWidget (ColorChanged c)
      txtButton text event = Button #{dim = (100,50)} (Text text) event
```

(a) The paint program update and view functions



(b) The paint program UI

Figure 8.3

nature and lack of boilerplate. Observe how modelling states as explicit data types allows for returning to them at any point without consequence as the rendered view is always only a function of a state data type, so `update` simply returns the initial state `init` if the clear button is pressed.

8.4 Pong

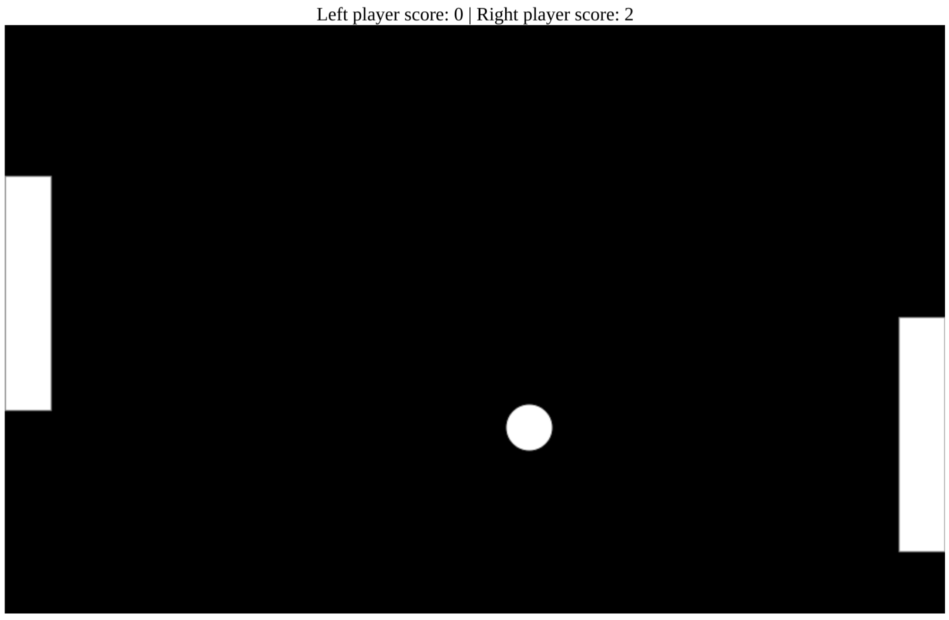


Figure 8.4: The UI of Pong

Pong is an extension of the Canvas capabilities to demonstrates how this can combine with subscriptions for advanced functionality, it also shows Syphon's capability of creating a responsive and fluid GUI. The same Canvas component is in-fact being used; `Oval` and `Rectangle` are also `Shape` constructors just as a `Stroke` is, so the paint program could easily be extended to support drawing circles and rectangles. Time is subscribed to at a set frequency at which `Tick` messages are sent to `update`, in every "tick" the paddles move (or don't) and ball moves by their respective speeds (simply an amount of pixels), and specific key presses and releases send event messages to change the state of the paddles. In each tick the state is checked if the ball intersects any edges or paddles, and bounces off at a "right angle" if intersecting or continues travelling in it's trajectory if not. If the ball intersects a left or right wall then the opposite player's score increments and the ball "re-spawns" in the center travelling in the opposite direction. The power of modelling dynamic states as data types really shines here: the ball can only ever be in one of the four diagonal direction states and the paddles can only ever be `GoingUp`, `GoingDown` or `Stationary` where all calculation of new position values is based solely on these states and the last position values. Hence, you can see that it is more or less impossible that the program will go into an illegal state as the state space is so constrained and explicitly modelled.

I believe this is a much more natural way to model a game than one would imperatively (a simple example[48]), all the code for manual drawing of shapes is abstracted away from the developer; they simply provide a shape list containing 2 rectangles and an oval stating their positions and dimensions and the Canvas widget handles rendering the entire game GUI. The core logic for the Pong program is contained within the `handleTick` function which is less than 60 lines of code, again demonstrating Syphon's concise, declarative nature whilst remaining clear and readable.

8.5 Minesweeper

Minesweeper demonstrates the

8.6 Final remark on evaluation

The SUnit language enables the developer to easily ensure correctness of Syphon programs as I have done. A flaw of parser-combinators is that they cannot guarantee a grammar is ambiguity free as a parser-generator can, however I believe the variety of programs I have created each using different nesting structures of expressions demonstrates functionality of the grammar and parser.

[48]https://www.geeksforgeeks.org/create-pong-game-using-python-turtle/ - accessed on 27/4/21

9 Project management

Meetings with my supervisor were held weekly and were invaluable to me, where we agreed upon miniature goals to complete for the next meeting and I could share any difficulties faced relating to set goals, in which case we would agree on a slightly altered goal for the next meeting.

A private GitHub repository was used to host my project, serving as version control and also as a regular backup to mitigate losing work.

Regular journalling was done in Google Docs to capture my thoughts and ideas.

Figure 9.1 shows the original gantt chart containing a planned timeline of my increments from the progress report. Figure 9.2 shows the actual progress timeline, with pale blue indicating an increment is omitted or unfinished; however, all increments relating to mandatory requirements were completed except for ad-hoc polymorphism, the rest were extensions. The gantt chart served as a helpful model for helping me to plan personal goals. Implementing the type system took longer than expected, however other parts of the project such as implementing GUI components and styling properties turned out to be straightforward so the time compensation worked out.

In hindsight implementing fully functioning Haskell-style type class was possibly too ambitious a goal considering a prior concrete understanding of Hindley-Milner inference is required which needed to be learnt from scratch.

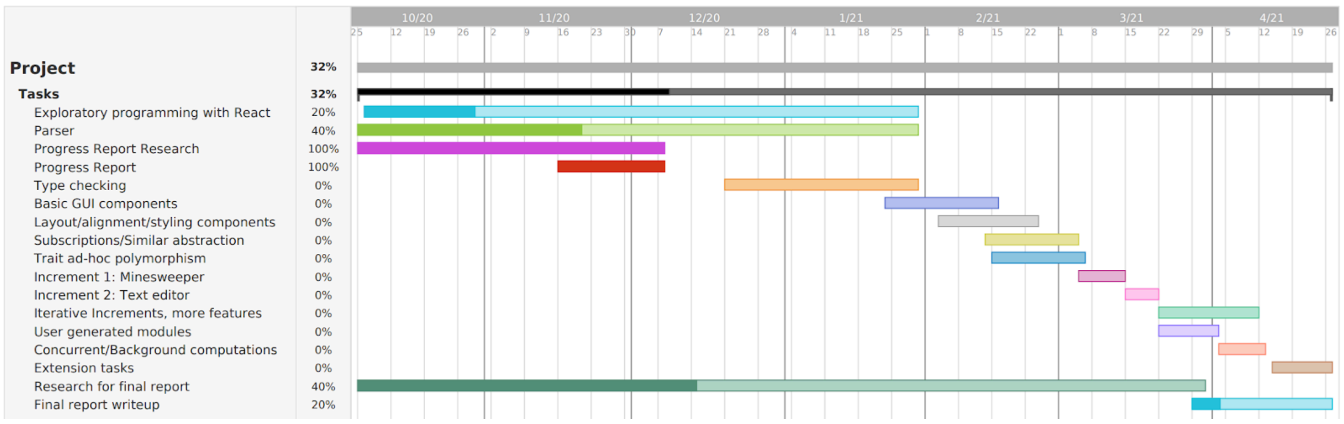


Figure 9.1: The original gantt chart showing planned increments and predicted time periods

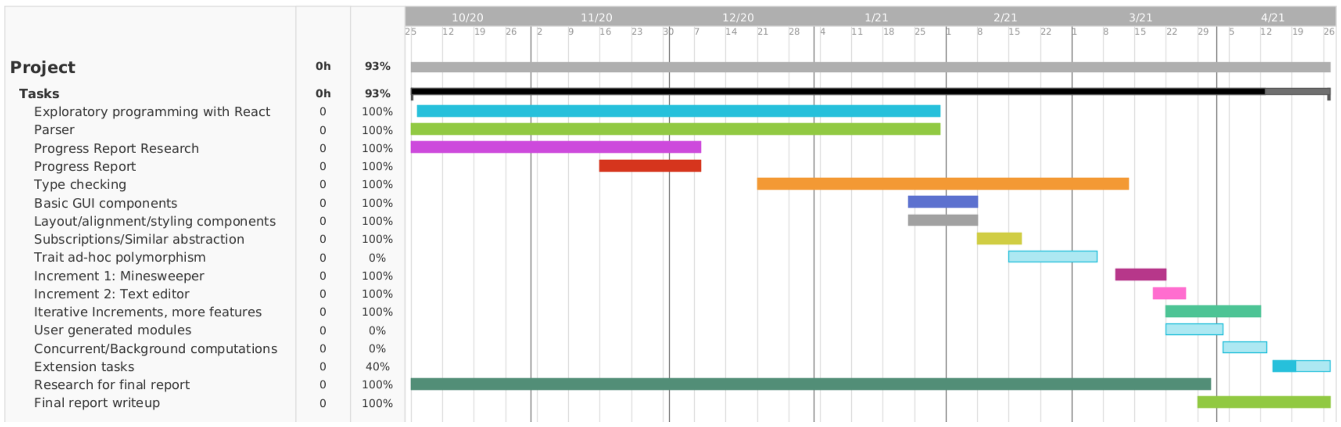


Figure 9.2: Gantt chart showing actual progress made

10 Conclusion

This paper summarizes techniques that can be used for implementing a functional GUI language. A syntax and semantics inspired by Elm but further simplified with several example programs demonstrates the potential of MVU, with minimal boilerplate and logic required to create aesthetic, fluid applications where modelling states as data types greatly limits the state space and the chance of illegal states. Parser combinators are a powerful tool where backtracking and stateful functionality allows for great versatility in parsing a context sensitive grammar, Megaparsec proved to be very scalable and made incrementally extending the grammar straightforward.

The type system shows the strength of treating programs as pure expressions, allowing a solvable algebraic system to be built up based on expression occurrences, going some way to ensure correctness for programs. Further compile-time correctness guarantees could be implemented such as exhaustive checks as in Elm and Purescript[49] that attempt to disallow uncovered pattern cases. As states are explicit data types I believe a "time-travel" debugger implementation for Syphon would be straightforward. My unfulfilled Haskell-style ad-hoc polymorphism goal would be a useful addition. I set out to make Syphon's syntax and semantics generalisable for different target platforms such as Flutterhttps://flutter.dev/ - accessed on 27/4/21; I believe Syphon is suitable for this however I anticipate further investigation. I suspect that simple dependant type functionality (Augustsson (1998)) may better formalize optional structures, for instance allowing lists of optional structures that can be mapped over. An import and module system would be the most valuable addition to Syphon.

[49]http://nicodelpiano.github.io/2015/07/10/GSOC:-Exhaustivity-Checker-for-PureScript/ - accessed on 27/4/21

References

Augustsson, L. (1998), Cayenne - a language with dependen types, *in* 'ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming', pp. 239--250.

Czaplicki, E. (2016), 'A Farewell to FRP', https://elm-lang.org/news/farewell-to-frp. Last accessed on 12/4/21.

E., C. & S., C. (n.d.), Asynchronous functional reactive programming for GUIs, *in* 'Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13'.

Elliot, C. & Huddak, P. (1997), 'Functional reactive animation'.

et al, H. (n.d.), 'Type classes in haskell', https://homepages.inf.ed.ac.uk/wadler/papers/classhask/classhask.ps.

Fowler, S. (2020), 'Model-view-update-communicate: Session types meet the elm architecture'.

Grabmüller, M. (2006), 'Algorithm w step by step', http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7733rep=rep1
Last accessed on 27/4/21.

Helbling, C. & Guyer, S. (2016), 'Juniper: a functional reactive programming language for the arduino'.

James, M. (2014), 'Time travel made easy'.

Jones, M. P. (2000), 'Typing haskell in haskell', https://web.cecs.pdx.edu/ mpj/thih/thih.pdf. Last accessed on 27/4/21.

Karpov, M. (2017), 'How to introduce custom error messages'.

Litt, G. (2020), 'Runtime visualization for model-view-update guis'.

Meyerovich, L., G. A. B. J. C. G. G. M. B. A. & Krishnamurthi, S. (2009), 'Flapjax'.

Milner, R. (1982), 'A theory of type polymorphism in programming', pp. 348--375.

Negrão, M. (2018), 'Nndef: livecoding digital musical instruments in supercollider using functional reactive programming'.

Nilsson, H. & Chupin, G. (2016), Funky grooves: declarative programming of full-fledged musical applications, *in* 'The 19th international symposium on practical aspects of declarative languages', pp. 163--172. Published in LNCS volume 10137.

Perez, I., M., B. & H., N. (2016), 'Functional reactive programming, refactored'.

S., H. (2020), 'Introducing .net multi-platform app ui'.

Sculthorpe, N. & Nilsson, H., . (2009), 'Safe functional reactive programming through dependent types'.

Seibel, P. (2009), *Coders at Work: Reflections on the Craft of Programming*, Apress, p. 213.

Soshnikov, D. & Kirilenko, I. (2014), 'Functional reactive programming: from natural user interface to natural robotics behavior'.

Toczé, K., M., V., P., S. & S., N.-T. (2016), 'Maintainability of functional reactive programs in a telecom server software'.

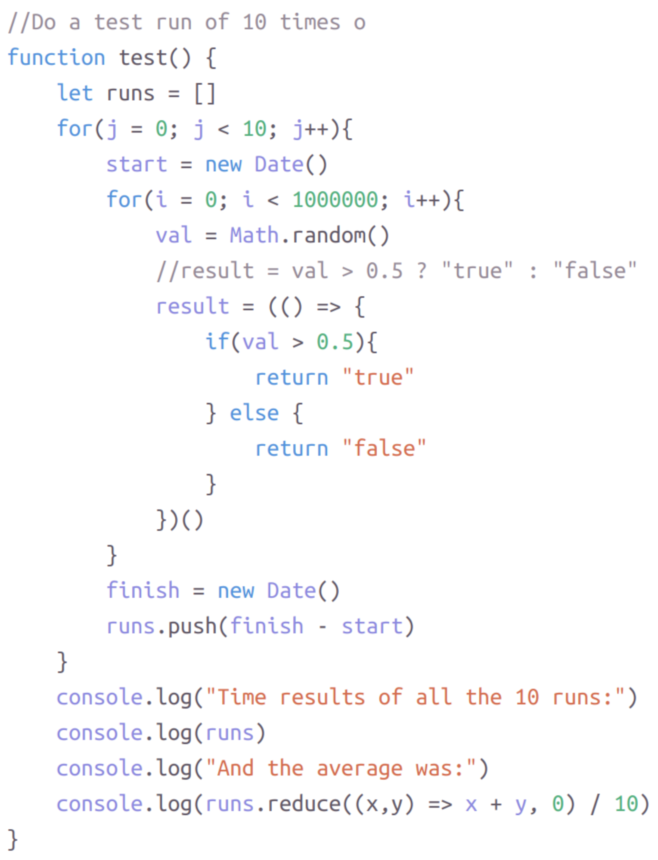
Wegner, P. (2003), *Encyclopedia of computer science*, John Wiley Sons, chapter Lambda Calculus, pp. 953--955.

Willis, J., W. N. & Pickering, M. (2020), 'Staged selective parser combinators'.

Total word count is: 10605

Appendices

A Investigating the speed of ternary expressions vs. anonymous functions



```
//Do a test run of 10 times o
function test() {
    let runs = []
    for(j = 0; j < 10; j++){
        start = new Date()
        for(i = 0; i < 1000000; i++){
            val = Math.random()
            //result = val > 0.5 ? "true" : "false"
            result = (() => {
                if(val > 0.5){
                    return "true"
                } else {
                    return "false"
                }
            })()
        }
        finish = new Date()
        runs.push(finish - start)
    }
    console.log("Time results of all the 10 runs:")
    console.log(runs)
    console.log("And the average was:")
    console.log(runs.reduce((x,y) => x + y, 0) / 10)
}
```

Figure A.1: The code used for investigations, simply testing creating values based on a random condition. 10 runs using the anonymous function and 10 runs using the commented out ternary expression

	Anonymous functions with if statement	Ternary expresion
	1386	1389
	1233	1317
	1222	1268
	1224	1262
	1225	1262
	1229	1264
	1230	1264
	1234	1264
Average time	1244.2	1281.7

Figure A.2: Results of my investigations, time in milliseconds

B Full source code of programs

B.0.1 Stopwatch

```
type State = {
   secs :: Int,
   millis :: Int,
   paused :: Bool
   }

type Event = Tick | TogglePause | Reset

@subscribe
   onTimePassed 10 Tick ?<< not state.paused

init :: State
init = 0 0 False

update :: State -> Event -> State
update s e = match e with
   Tick -> cond
      | s.millis == 99 -> {s | secs = s.secs + 1, millis = 0}
      | otherwise -> {s | millis = s.millis + 1}
   TogglePause -> {s | paused = not s.paused}
   Reset -> init

view :: State -> Widget
view s = Column []<<
   Container #{bgColor = Grey} $ Text $ append (toString s.secs) (toString s.millis)
   Row []<<
      Button (Text "Pause") TogglePause
      Button (Text "Reset") Reset
```

B.0.2 Calculator

```
alias Operator = Int -> Int -> Int

type State =
  {
  display :: String,
  operator :: Operator,
  operand1 :: Int
  }

type Event = Clear | Eq | OpPress Operator | Digit Int

--Use plus as a "dummy" opterator
init :: State
init = State "" (+) 0

update :: State -> Event -> State
update s e = match e with
  Clear -> init
  Eq -> State (calc s) (+) 0
  OpPress op -> State "" op $ parseInt s.display
  Digit i -> {s | display = append s.display (toString i)}
  where
    calc st = toString (st.operator st.operand1 (parseInt st.display))

view :: State -> Widget
view s = Column []<<
  Container #{dim = (275,35), bgColor = Grey} $ Text s.display
  Row $ (map mkDigitButton [7..9]) ++ [Button (Text "X") (OpPress (*))]
  Row $ (map mkDigitButton [4..6]) ++ [Button (Text "/") (OpPress (/))]
  Row $ (map mkDigitButton [1..3]) ++ [Button (Text "-") (OpPress (-))]
  Row [mkDigitButton 0, Button (Text "AC") Clear, Button (Text "=") Eq,
    Button (Text "+") (OpPress (+))]
  where
    mkDigitButton i = Button (Text $ toString i) (Digit i)
```

B.1 Paint

```
--Static literal constants do not need a type signature
weight = 3

--The strokes field must always have at least one stroke in it which points are added to
type State = {
   dragging :: Bool,
   color :: Color, --The currently selected color
   strokes :: [Shape]
   }

type Event
   = PixelEntered Int Int
   | MousePressed
   | MouseReleased
   | ColorChanged Color
   | Dummy
   | Clear
   | Undo

emptyStroke :: Color -> Shape
emptyStroke c = Stroke c weight []

init :: State
init = State False Black [emptyStroke Black] []

update :: State -> Event -> State
update s e = match e with
   PixelEntered y x -> {s | strokes = addPointToStrokes y x s.strokes}
   MousePressed -> {s | dragging = True}
   MouseReleased -> {s | dragging = False, strokes = emptyStroke s.color:s.strokes}
   ColorChanged c -> {s | strokes = updateColor c s.strokes, color = c}
   Clear -> init
   Undo -> {s | strokes = safeDrop s.strokes}
   Dummy -> s
   where
      addPointToStrokes y x ((Stroke c weight tups):strokes) =
         ((Stroke c weight ((y,x):tups)):strokes)
      updateColor c ((Stroke _ weight tups):strokes) = Stroke c weight tups:strokes
      safeDrop list = match list with
         [empty] -> [empty] --always ensure there is at least one empty stroke
         empty:_:rest -> empty:rest

view :: State -> Widget
view s = Column []<<
   Row $ (++)
      map mkColorButton [Black, Red, Purple, Blue, Green, Yellow, Pink, Orange, White]
      [txtButton "Clear" Clear, txtButton "Undo" Undo]
   Canvas $<<
      #{onMouseMoved = if s.dragging then PixelEntered else \_ _ -> Dummy,
         onMousePressed = \_ _ -> MousePressed, onMouseReleased = \_ _ -> MouseReleased}
      700
      500
      s.strokes
   where
      mkColorButton c = Button #{bgColor = c, dim = (50,50)} DummyWidget (ColorChanged c)
      txtButton text event = Button #{dim = (100,50)} (Text text) event
```

B.2 Minesweeper

```
asset flag = "./lib/GUI/assets/flag.svg"
asset mine = "./lib/GUI/assets/mine.png"
asset explosion = "./lib/GUI/assets/explosion.png"

type State = {
  grid :: [[CellState]],
  bombCoords :: [(Int,Int)],
  gameState :: GameState,
  gen :: RandomGen
  }

type CellState = Hidden | Revealed Int | Flagged | Exploded
type GameState = Playing | Won | Lost | Loading
type Event = LeftClick Int Int | RightClick Int Int | Reset | SeedRecieved Int | Dummy

{-
Start with a random generator from the seed 0, and then replace it with a new random
generator with the seed recieved from IO to make it truly random
-}
init :: State
init = State (replicate2D 10 Hidden) [] Loading (mkRandomGen 0)


type Event
  = LeftClick Int Int
  | RightClick Int Int
  | Reset
  | SeedRecieved Int
  | Dummy

initEffects :: [Effect]
initEffects = [requestSeed SeedRecieved]

update :: State -> Event -> State
update s e = match e with
  LeftClick y x -> match index2D s.grid y x with
    Hidden -> match bombOrCount s.bombCoords (y,x) with
      Nothing -> {s | grid = replace2D s.grid y x Exploded, gameState = Lost}
      Just count -> let
        newGrid = replace2D s.grid y x $ Revealed count
        in
        cond
          | checkWon -> {s | grid = newGrid, gameState = Won}
          | otherwise -> {s | grid = newGrid}
    Revealed _ -> s
    Flagged -> s
  RightClick y x -> match index2D s.grid y x with
    Hidden -> {s | grid = replace2D s.grid y x Flagged}
    Flagged -> {s | grid = replace2D s.grid y x Hidden}
    Revealed _ -> s
  Dummy -> s
  Reset -> let
    (newBombCoords, newGen) = genBombCoords s.gen 10
    newGrid = replicate2D 10 Hidden
    in
    {s | gen = newGen, bombCoords = newBombCoords, gameState = Playing, grid = newGrid}
  SeedRecieved seed -> let
    initGen = mkRandomGen seed
    (rndmCoords, gen1) = genBombCoords initGen 10
    in
    {s | gen = gen1, bombCoords = rndmCoords, gameState = Playing}
  where
    checkWon = pipeline
```

```
        s.grid
        filter2D $ \cellState -> match cellState with
          Revealed _ -> True
          _ -> False
        concat
        length
        \result -> result == 90
      genBombCoords gen count = genBombCoordsRec gen count []

genBombCoordsRec :: RandomGen -> Int -> [(Int, Int)] -> ([(Int,Int)], RandomGen)
genBombCoordsRec gen count acc
  | count == 0 = (acc,gen) --We have built up the number of bomb coordinates needed so return
  | otherwise = let
      (y,gen1) = randomGenBetween 0 9 gen
      (x,gen2) = randomGenBetween 0 9 gen1
      ranCoord = (y,x)
      in
      cond
        | containsCoord acc ranCoord -> genBombCoordsRec gen2 count acc
        | otherwise -> genBombCoordsRec gen2 (count - 1) (ranCoord:acc)


containsCoord :: [(Int,Int)] -> (Int,Int) -> Bool
containsCoord [] _ = False
containsCoord ((y1,x1):coords) (y2,x2)
  | y1 == y2 && x1 == x2 = True
  | otherwise = containsCoord coords (y2,x2)

bombOrCount :: [(Int,Int)] -> (Int, Int) -> Maybe Int
bombOrCount [] _ = Just 0 --Looked through all bomb coordinates and none match so 0 surrounding bombs
bombOrCount ((y1,x1):coords) (y2,x2)
  | diffY == 0 && diffX == 0 = Nothing --0 difference means the coordinate is a bomb!
  | diffY <= 1 && diffX <= 1 = plusMaybe 1 $ bombOrCount coords (y2,x2)
  | otherwise = bombOrCount coords (y2,x2)
  where
    diffY = abs $ y1 - y2
    diffX = abs $ x1 - x2
    plusMaybe x maybe = match maybe with
      Nothing -> Nothing
      Just y -> Just $ x + y

view :: State -> Widget
view s = Column $ (:) $<<
  Row [Button (Text "Reset") Reset]
  map (\y -> Row $ makeCellList s.gameState y 0 (s.grid !! y) s.bombCoords) [0..9]

makeCellList :: GameState -> Int -> Int -> [CellState] -> [(Int,Int)] -> [Widget]
makeCellList _ _ _ [] _ = [] --No more cell states to look at so we have created all buttons required
makeCellList gameState y x (cellState:cellStates) bombCoords = (:) $<<
  match gameState with
    Loading -> Panel #{dim = (50,50), bgColor = Blue} --Graphic TimerGraphic
    Won -> cond
      | containsCoord bombCoords (y,x) -> Container #{bgColor = Green} $ mine #{dim = (50,50)}
      | otherwise -> Panel #{dim = (50,50), bgColor = Green}
    Lost -> cond
      | isExploded cellState -> Container #{bgColor = White} $ explosion #{dim = (50,50)}
      | containsCoord bombCoords (y,x) -> Container #{bgColor = White} $ mine #{dim = (50,50)}
      | otherwise -> Panel #{dim = (50,50), bgColor = Red}
    Playing -> match cellState with --Don't need an explode case as game will be lost as soon as
         explosion occurs
      Hidden -> Button $<<
        #{rounding = 0, onRightClick = RightClick y x, dim = (50,50), bgColor = DarkGrey}
        DummyWidget
        LeftClick y x
      Flagged -> Button $<<
```

```
            #{rounding = 0, onRightClick = RightClick y x, dim = (50,50)}
            Container $ flag #{dim = (40,40)}
            Dummy
        Revealed count -> Container #{dim = (50,50), bgColor = LightGrey} $ Text $ toString count
makeCellList gameState y (x + 1) cellStates bombCoords
where
    isExploded cellSt = match cellSt with
        Exploded -> True
        _ -> False
```

B.3 Text editor

```
type State = {
   saveField :: String,
   loadField :: String,
   contents :: String
   }

type Event
   = LoadPressed
   | SavePressed
   | FileRead String
   | LoadUpdated String
   | SaveUpdated String
   | EditorTyped String

init :: State
init = State "" "" ""

effect :: State -> Event -> Effect
effect s e = match e with
   LoadPressed -> readFile s.loadField FileRead
   SavePressed -> writeFile s.saveField s.contents
   _ -> NoEffect

update :: State -> Event -> State
update s e = match e with
   EditorTyped str -> {s | contents = str}
   LoadUpdated str -> {s | loadField = str}
   SaveUpdated str -> {s | saveField = str}
   FileRead str -> {s | contents = str}
   _ -> s

view :: State -> Widget
view s = Column #{gap = 10} []<<
   Row [TextField "Load directory" LoadUpdated, Button (Text "load from path") LoadPressed]
   Row [TextField "Save directory" SaveUpdated, Button (Text "save to path") SavePressed]
   TextEditor s.contents 30 40 EditorTyped
```

B.4 Pong

```
--Static constants by convention go at the top of the file
canvasWidth = 800
canvasHeight = 500

ballDiam = 20
paddleWidth = 40
paddleHeight = 200

--Just how many pixels they move by on each tick
ballSpeed = 1
paddleSpeed = 2

--The millisecond interval of tick messages
tickFreq = 5

type State = {
   leftPaddleTop :: Int,
   rightPaddleTop :: Int,
   leftPaddleState :: PaddleState,
   rightPaddleState :: PaddleState,
   leftScore :: Int,
   rightScore :: Int,
   ballY :: Int,
   ballX :: Int,
   ballDir :: Dir,
   state :: GameState,
   gen :: RandomGen
   }

type PaddleState = Stationary | GoingUp | GoingDown
type Dir = NE | SE | SW | NW
--As soon as either score gets to 10 then that player wins
type GameState = Playing | Paused | Loading
type Wall = Top | Bottom | Lft | Rght
type Player = LeftPlayer | RightPlayer

type Event
   = LeftPlayerUp
   | LeftPlayerDown
   | LeftPlayerRelease
   | RightPlayerUp
   | RightPlayerDown
   | RightPlayerRelease
   | Pause
   | Tick

@subscribe
   onTimePassed tickFreq Tick ?<< True
   onKeyPressed UpArrow RightPlayerUp ?<< True
   onKeyPressed DownArrow RightPlayerDown ?<< True
   onKeyReleased UpArrow RightPlayerRelease ?<< True
   onKeyReleased DownArrow RightPlayerRelease ?<< True
   onKeyPressed (KeyChar 'w') LeftPlayerUp ?<< True
   onKeyPressed (KeyChar 's') LeftPlayerDown ?<< True
   onKeyReleased (KeyChar 'w') LeftPlayerRelease ?<< True
   onKeyReleased (KeyChar 's') LeftPlayerRelease ?<< True

init :: State
init = State 0 0 Stationary Stationary 0 0 400 400 NW Playing (mkRandomGen 0)

update :: State -> Event -> State
update s e = match e with
   Tick -> handleTick s
```

```
  LeftPlayerUp -> {s | leftPaddleState = GoingUp}
  LeftPlayerDown -> {s | leftPaddleState = GoingDown}
  LeftPlayerRelease -> {s | leftPaddleState = Stationary}
  RightPlayerUp -> {s | rightPaddleState = GoingUp}
  RightPlayerDown -> {s | rightPaddleState = GoingDown}
  RightPlayerRelease -> {s | rightPaddleState = Stationary}
  Pause -> {s | state = Paused}

--For now just handle the paddles going up and down, implement ball stuff later
handleTick :: State -> State
handleTick s = let
  goUp val = cond
    | val - paddleSpeed < 0 -> 0
    | otherwise -> val - paddleSpeed
  goDown val = cond
    | val + paddleSpeed + paddleHeight > canvasHeight -> canvasHeight - paddleHeight
    | otherwise -> val + paddleSpeed
  calcNewTop paddleState top = match paddleState with
    GoingUp -> goUp top
    GoingDown -> goDown top
    Stationary -> top
  leftTop = calcNewTop s.leftPaddleState s.leftPaddleTop
  rightTop = calcNewTop s.rightPaddleState s.rightPaddleTop
  (uncheckedY, uncheckedX) = match s.ballDir with
    NE -> (s.ballY - 1, s.ballX + 1)
    SE -> (s.ballY + 1, s.ballX + 1)
    SW -> (s.ballY + 1, s.ballX - 1)
    NW -> (s.ballY - 1, s.ballX - 1)
  leftBottom = leftTop + paddleHeight
  rightBottom = rightTop + paddleHeight
  bounce wall dir = match (wall,dir) with
    (Top, NE) -> (SE,0,uncheckedX)
    (Top, NW) -> (SW,0,uncheckedX)
    (Bottom, SE) -> (NE,canvasHeight - ballDiam,uncheckedX)
    (Bottom, SW) -> (NW,canvasHeight - ballDiam,uncheckedX)
    (Lft, NW) -> (NE,uncheckedY,paddleWidth)
    (Lft, SW) -> (SE,uncheckedY,paddleWidth)
    (Rght, NE) -> (NW,uncheckedY,canvasWidth - ballDiam - paddleWidth)
    (Rght, SE) -> (SW,uncheckedY,canvasWidth - ballDiam - paddleWidth)
  ballTop = s.ballY
  ballBottom = s.ballY + ballDiam
  ballLeft = s.ballX
  ballRight = s.ballX + ballDiam
  between val lowerBound upperBound = val < upperBound && val > lowerBound
  iscLeftPad = ballLeft < paddleWidth &&
    (between ballTop leftTop leftBottom || between ballBottom leftTop leftBottom)
  iscRightPad = ballRight > canvasWidth - paddleWidth &&
    (between ballTop rightTop rightBottom || between ballBottom rightTop rightBottom)
  iscTop = ballTop < 0
  iscBot = ballBottom > canvasHeight
  iscLeftWall = ballLeft < 0
  iscRightWall = ballRight > canvasWidth
  scoreOrNewBallInfo = cond
    | iscLeftPad -> Right (bounce Lft s.ballDir)
    | iscRightPad -> Right (bounce Rght s.ballDir)
    | iscTop -> Right (bounce Top s.ballDir)
    | iscBot -> Right (bounce Bottom s.ballDir)
    | iscLeftWall -> Left RightPlayer
    | iscRightWall -> Left LeftPlayer
    | otherwise -> Right (s.ballDir, uncheckedY,uncheckedX)
  opposite dir = match dir with
    NW -> SE
    SE -> NW
    NE -> SW
    SW -> NE
  in
  match scoreOrNewBallInfo with
    Right (newDir,newY,newX) ->
      {s | ballDir = newDir, ballY = newY, ballX = newX, leftPaddleTop = leftTop, rightPaddleTop =
          rightTop}
    Left RightPlayer -> {s | rightScore = s.rightScore + 1,
      ballY = 400, ballX = 400, ballDir = opposite s.ballDir}
    Left LeftPlayer -> {s | leftScore = s.leftScore + 1,
      ballY = 400, ballX = 400, ballDir = opposite s.ballDir}

view :: State -> Widget
view s = Column #{bgColor = Black} []<<
  Row [Text $ toString s.leftScore, Text $ toString s.rightScore]
  Canvas #{bgColor = Black} canvasWidth canvasHeight []<<
    Oval White s.ballY s.ballX ballDiam ballDiam --ball will (should) be frozen in time for now :)
    Rectangle White s.leftPaddleTop 0 paddleWidth paddleHeight
    Rectangle White s.rightPaddleTop (canvasWidth - paddleWidth) paddleWidth paddleHeight
```

C Parser

C.1 Whitespace and comments

```
comment = L.skipLineComment "--"
commentMl = L.skipBlockComment "{-" "-}"

sc = hidden $ do
    let normalSc = L.space (void $ char ' ') comment commentMl
    lf <- asks inLineFold
    if lf
       then normalSc *> (option () lineFoldSpace) <* normalSc
       else normalSc

lineFoldSpace = void $ do
   --Elements in a linefold must have one level of indentation greater
   lvl <- asks tabLvl
   try (char '\n' >> count (lvl + 1) (char '\t') <* notFollowedBy (char
       '\t'))
```

(a) Space consumers with comment parsers

```
lexeme p = L.lexeme sc p

lChar chr = lexeme $ char chr
```

(b) Lexeme definition and example "lexeme-d" parser

Figure C.1: Whitespace and comment handling parsers

Megaparsec includes the `space` function which takes a whitespace consuming parser, a single line comment parser and a block comment parser and returns a new whitespace consuming parser that behaves the same as the one given as an argument, except it now also consumes all occurrences of single line and multi-line comments specified by the parser arguments, Megaparsec also provides predefined parsers for these comments as seen in Figure C.1. A line-fold is simply some syntactic structure that can be written across multiple lines, as-long as all lines after the first line have one level of indentation greater than the first line. Once we have defined a space consumer, we can define *lexemes* which are parsers that handle consuming their trailing whitespace. In Figure C.1b, the Megaparsec `lexeme` function takes a parser argument and produces a new parser that consumes its trailing whitespace with the space consumer, `lChar` is the lexeme version of a char parser. Most parsers I use are lexemes, as this handles consumption of all whitespace and comments in the file (apart from of course the initial leading whitespace in the file before the first syntactic structure).

Manual logic again comes into play here for handling indentation, the environment value of `inLineFold` is queried, if true then allow for a line fold space (newline followed by 1 + the current level of indentation of tabs), if false then don't allow for a line fold space. This value is important as we want to explicitly disallow linefolds in some syntactic structures such as in match expressions, where the list of patterns should not be parsed in a linefold environment, only each of the subexpressions after the "->" should be allowed to be in a linefold *** EXPLAIN THIS *** .

C.2 Choice of syntax

```
exprs <- option [] $ (sepBy (char ',') pExpr)
   <|> (opStr >>= \str -> return [Var str])
```

(a) Monadic bind syntax

```
str <- (:)<$> lowerChar <*> many alphaNumChar
```

(b) Applicative syntax

```
pAlias :: Parser Dfn
pAlias = inLf $ fmap (unPos . sourceLine) getSourcePos >>=
     \lineNum -> try $ pKeyword "alias" >> pCapsIdentifier >>=
     \name -> lChar '=' >> pType >>= \ty -> greedySc >>
     (return $ Alias name lineNum ty)
```

(c) pAlias definition without do notation

Figure C.2: Variety of syntax

As `ParsecT` is an instance of both an applicative (understanding what an applicative is is unnecessary, think of it as a weaker version of a monad) and a monad, we have the option of using the monadic bind operators >> and >>=, do notation and also applicative syntax (mainly the <*> and <$> operators). All 3 of these are convenient in different situations, and combining them in various ways allows for clear and readable parser code. For example in the excerpt of the parenthesised expression parser in Figure C.2a, by using >>= we can access the result of the `opStr` parser directly inside a parser expression in an alternative chain without needing to use do notation which is much more concise whilst still remaining readable (Think of >>= as an operator that feeds the output of the left hand side into the right hand side). Applicative syntax is the most practical choice in a few situations such as the excerpt of the variable identifier parser (Figure C.2b), by lifting cons into the parser context through the use of <$>, we can simply think of the result of the parser as being a parsed lowercase character "cons'd" to a list of alphanumeric characters. Do notation is mostly used for the top-level parser function definitions to "tie everything together", as they are generally more complex so it clearly shows where all the different sub-parts forming the final parsed data type are coming from and allows the overall code to be reasoned about in an almost "imperative" manor. Consider if no do notation was used for the aforementioned `pAlias` function from Figure 5.7a, the result is Figure C.2c which is much less readable.

C.3 Binary operators

```
opTable :: [[Operator Parser Expr]]
opTable = [
   -- $ and : are right associative as in Haskell, all other operators are left associative
   -- Dollar operator will only ever be called with a function call as LHS arg, so
   --simply add RHS arg to the LHS funcall list of args
   [binary "*", binary "^", binary "/", binary "%"],
   [binary "+", binary "-"],
   [binary "==", binary "!="],
   [binary ">", binary ">=", binary "<", binary "<="],
   [binary "&&"],
   [binary "||"],
   [binary "++"],
   [InfixR $ handleDollar <$ (op "$" <?> "a binary operator")],
   [InfixR $ (\lhs rhs -> App (App (Var ":") lhs) rhs) <$ (op ":" <?> "a binary operator")]
```

Figure C.3: Binary operator precedence table

For handling binary operators, Megaparsec provides the `makeExprParser` function which takes as argument the "function application" expression parser without binary operators and an operator table argument (Figure C.3), allowing for binary operators to be written inbetween these function application expressions so that things like `fn0 arg00 + fn1 arg01` can be parsed as one entire expression. We simply declare a 2D list where the operators are listed in order of descending precedence (how "tightly" they bind), and declare them to be right associative, left associative or have no associativity with `InfixL`, `InfixR` and `InfixN`. Each inner parser takes as argument the "LHS" and "RHS", parses the binary operator and returns the resulting overall expression (`<$` can be read as "run the parser on the right, if it succeeds then lift the thing on the left into the parser context and return it"). `binary` is a helper function that does the same as the cons operator case but uses `InfixL` for left associativity instead. As you can see in the cons operator parse case, Syphon binary operators as in Haskell are treated as a curried expression application to the "LHS" and "RHS" expressions.

D Type system

D.1 Inference (continued)

D.1.1 Patterns

Inferring patterns is slightly trickier because not only does inferring a pattern return the type of the pattern and a set of constraints generated from inferring it, patterns also have the property of being able to insert new type bindings into the environment. The `inferPattern` function in the Haskell code builds up a map of type/variable bindings extracted from the patterns during inference and then the part of the normal `infer` function for expressions which called the `inferPattern` function (It will become clear in later sections where patterns are used in expressions) modifies the environment by replacing the old simple section of the environment (a map of type/variable bindings) with with the map of bindings found from inferring the pattern unioned with the old simple section.

With this added complexity of pattern type inference rules additionally returning maps of bindings, the notation must be extended slightly. The bindings are written in angular brackets after the set of constraints, similarly to constraints the pattern expression may generate it's own map of bindings ($B_{ptrnexpr}$), and may contain any number of sub--patterns that also generate their own maps of bindings ($B_0, ..., B_n$):

$$\frac{\Gamma \vdash subptrn_0 :: t_0 \dashv C_0, \langle B_0 \rangle \qquad ... \qquad \Gamma \vdash subptrn_n :: t_n, \dashv C_n, \langle B_n \rangle}{\Gamma \vdash ptrnexpr :: t, \dashv C_{ptrnexpr} \cup C_0 \cup ... \cup C_n, \langle B_{ptrnexpr} \cup B_0 \cup ... \cup B_n \rangle}$$

I will demonstrate this abit more clearly with another example, below is the pattern inference rule for the cons pattern:

TyConsPtrn

$$\frac{\Gamma \vdash p_0 :: t_0 \dashv C_0, \langle B_0 \rangle \qquad ... \qquad \Gamma \vdash p_n :: t_n \dashv C_n, \langle B_n \rangle \qquad \Gamma \vdash ps :: t_{n+1} \dashv C_{n+1}, \langle B_{n+1} \rangle}{\Gamma \vdash (p_0 : ... : p_n : ps) :: [\alpha_0] \dashv \{t_i \doteq \alpha_0, i \in \{0...n\}\} \cup \{t_{n+1} \doteq [\alpha_0]\} \cup C_0 \cup ... \cup C_n \cup C_{n+1}, \langle B_0 \cup ... \cup B_n \cup B_{n+1} \rangle}$$

Note that $p_0, ..., p_n, ps$ do not represent variable patterns but rather pattern expressions. In this rule it is not the root cons pattern expression that creates any bindings itself, it aggregates the bindings obtained from inferring each of the sub patterns $p_0, ..., p_n, ps$. I hope that the intuition of the cons pattern constraints is clear, all elements except the last element in the cons pattern must be of type α_0, and the last element in the cons pattern must be a list of type $[\alpha_0]$, which precisely reflects the behaviour of the cons function type signature: `a -> [a] -> [a]`. As discussed previously, patterns are a combination of both predicates and bindings hence some or all of the maps $B_0, ..., B_n$ may be empty; for example suppose p_n is a literal int pattern with the value 7, clearly this is a predicate (p_n must have the value 7) and does not contain a binding so in this case $B_n = \emptyset$. The variable pattern is in fact the only pattern that generates a variable/type binding; the wildcard pattern is similar, except the intuition of a wildcard is that its value needn't be referenced anywhere in the expression tree so therefore we do not generate an environment binding:

TyVarPtrn
$$\Gamma \vdash x :: \alpha_0 \dashv \emptyset, \langle \{x :: \alpha_0\} \rangle$$

TyWildcardPtrn
$$\Gamma \vdash \_ :: \alpha_0 \dashv \emptyset, \langle \emptyset \rangle$$

Inferring literal value patterns is again trivial as you might imagine, the rules are identical to the literal value expression rules except we now need to state that the pattern generates an empty set of bindings:

$$\frac{}{\Gamma \vdash i :: \texttt{Int} \dashv \emptyset, \langle\emptyset\rangle}\,\text{TyIntPtrn} \qquad \frac{}{\Gamma \vdash d :: \texttt{Double} \dashv \emptyset, \langle\emptyset\rangle}\,\text{TyDoublePtrn} \qquad \frac{}{\Gamma \vdash b :: \texttt{Bool} \dashv \emptyset, \langle\emptyset\rangle}\,\text{TyBoolPtrn} \qquad \frac{}{\Gamma \vdash c :: \texttt{Char} \dashv \emptyset, \langle\emptyset\rangle}\,\text{TyCharPtrn}$$

$$\frac{}{\Gamma \vdash s :: \texttt{String} \dashv \emptyset, \langle\emptyset\rangle}\,\text{TyStringPtrn}$$

Inferring a constructor pattern is actually very closely related to the function expression rule, due to the fact that constructors are actually just functions. We know that once a function is applied to all its arguments it will evaluate to some return type which is our assumption α_0. Clearly the type signature stored in the environment for the constructor function `Constr` will be equivalent to an arrow type with arguments of all the inferred pattern types (these are the arguments), returning our assumption α_0. By generating this constraint, this will allow the unification algorithm to solve for the value of α_0 by comparing with the function signature from the environment lookup with our generated function signature, breaking them down into their atomic types.

$$\text{TyConstrPtrn}\quad \frac{\Gamma \vdash p_0 :: t_0 \dashv C_0, \langle B_0\rangle \quad \ldots \quad \Gamma \vdash p_n :: t_n \vdash C_n, \langle B_n\rangle \quad t_{Constr} = \texttt{lookup}^!(\Gamma, \texttt{Constr})}{\Gamma \vdash \texttt{Constr}\ p_0\ \ldots\ p_n :: \alpha_0 \dashv \{t_{Constr} \doteq t_0 \to \ldots \to t_n \to \alpha_0\} \cup C_0 \cup \ldots \cup C_n, \langle B_0 \cup \ldots \cup B_n\rangle}$$

Inferring the type of a tuple pattern is trivial and very similar to inferring the type of a tuple expression, we simply infer the type of each of the sub-patterns and wrap these in a structured tuple type:

$$\text{TyTupPtrn}\quad \frac{\Gamma \vdash p_0 :: t_0 \dashv C_0, \langle B_0\rangle \quad \ldots \quad \Gamma \vdash p_n :: t_n \dashv C_n, \langle B_n\rangle}{\Gamma \vdash (p_0, \ldots, p_n) :: (t_0, \ldots, t_n) \dashv C_0 \cup \ldots \cup C_n, \langle B_0 \cup \ldots \cup B_n\rangle}$$

D.1.2 Match expressions

The most obvious example where patterns are used is inside a match expression. To infer patterns within expressions the notation must again be extended, this is straightforward though: let `inferPattern` be the function that infers the type of the pattern within the environment Γ and returns the inferred type, the set of constraints generated by the pattern and all its sub patterns and the set of bindings generated by the pattern and all its sub patterns, these bound variables will be written atop the inference definition of the corresponding subexpression that uses the bindings generated by the pattern. Once a pattern has been inferred we must inject the bindings generated into the simple part of the environment and within this modified environment infer the corresponding expression (i.e. in match expressions e_k is inferred in an environment that contains the bindings generated by p_k). Let $B \gg \Gamma$ denote injecting the set of bindings B into the simple part of Γ resulting in a new environment, such that the simple part of the new environment is the simple part of the old environment unioned with B (recall that the simple part of the environment holds type/variable bindings). All patterns clearly must have the same type as the expression that is being matched (e_m), and we generate a fresh variable as our assumption of the type of the entire match expression, stating that each sub expression must have this type.

$$\text{TyMatch}\quad \frac{\Gamma \vdash e_m :: t_m \dashv C_m \qquad \begin{array}{c}\mathrm{t}_{a0}, C_{a0}, B_0 = \texttt{inferPattern}(\Gamma, p_0)\\ \mathrm{B}_0 \gg \Gamma \vdash e_0 :: t_{b0} \dashv C_{b0}\end{array} \qquad \ldots \qquad \begin{array}{c}\mathrm{t}_{an}, C_{an}, B_n = \texttt{inferPattern}(\Gamma, p_n)\\ \mathrm{B}_n \gg \Gamma \vdash e_n :: t_{bn} \dashv C_{bn}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{match}\ e_m\ \texttt{with}\ p_0 \to e_0\ \ldots\ p_n \to e_n :: \alpha_0 \dashv \{t_{ai} \doteq t_m, i \in \{0 \ldots n\}\} \cup \{t_{bi} \doteq \alpha_0, i \in \{0 \ldots n\}\}\\ \cup C_m \cup C_{a0} \cup C_{b0} \cup \ldots \cup C_{an} \cup C_{bn}\end{array}}$$

D.1.3 Lambda expressions

Unlike in the standard Lambda Calculus, Syphon's Lambda expressions like Haskell's makes use of patterns in arguments as opposed to simple variables. This is very useful in many situations for instance if you take a list as an argument but you are only ever interested in using the head of the list, you can write something like:

$$(\backslash(hd : \_)\ \to\ <\texttt{lambda body expression that references}\ hd>)\ [5, 90, 13, 4]$$

So here, *hd* will have a value of 5. I hope this demonstrates how much of a powerful, expressive tool patterns can be. The rule is straightforward, we simply infer each of the argument patterns, and then inject all the bindings generated from these into the simple part of the environment in which the body expression is inferred, and obviously it's type is an arrow type with arguments of the types of all the patterns and returning the type of the body expression:

$$\frac{t_0, C_0, B_0 = \texttt{inferPattern}(\Gamma, p_0) \quad \dots \quad t_n, C_n, B_n = \texttt{inferPattern}(\Gamma, p_n) \qquad (B_0 \cup \dots \cup B_n) \gg \Gamma \vdash e :: t_{Body} \dashv C_{Body}}{\Gamma \vdash \backslash p_0 \ \dots \ p_n \rightarrow e :: t_0 \rightarrow \dots \rightarrow t_n \rightarrow t_{Body} \dashv C_{Body} \cup C_0 \cup \dots \cup C_n}\ \text{TyLambda}$$

D.1.4 Regular functions

Regular module level functions are very closely related to lambda expressions due to the fact that any kind of function in Syphon is a Javascript arrow function under the hood, however Syphon requires that module level definitions *must* provide a type signature, hence we can add the constraint that our inferred type matches the expected type that is in the environment, I will explain the advantage of this below. Regular functions can be thought of as syntactic sugar for module level let bindings with a type signature.

$$\frac{t_0, C_0, B_0 = \texttt{inferPattern}(\Gamma, p_0) \quad \dots \quad t_n, C_n, B_n = \texttt{inferPattern}(\Gamma, p_n) \qquad (B_0 \cup \dots \cup B_n) \gg \Gamma \vdash e :: t_{Body} \dashv C_{Body} \qquad t_{foo} = \texttt{lookup}^{!}(\Gamma, \texttt{foo})}{\Gamma \vdash \texttt{foo}\ p_0 \ \dots \ p_n \ = e :: t_{foo} \dashv \{t_0 \rightarrow \dots \rightarrow t_n \rightarrow t_{Body} \doteq t_{foo}\} \cup C_{Body} \cup C_0 \cup \dots \cup C_n}\ \text{TyRegFn}$$

Syphon's type system does not allow for mutual recursion of local, untyped bindings; for instance something like

```
let odd = \n → if n == 1 then True else not $ even (n − 1) in
  let even = \n → if n == 0 then True else not $ odd (n − 1)  in even 2
```

is not possible as during inference of the nested let expressions, when we are inferring the odd let expression we will come across even in it's body and query the environment for it's type, however this is outside the scope in which it is declared (let definitions are only in scope within the body expression after the `in` keyword). For this reason, Syphon places the constraint that all regular module level functions *must* provide a type signature, the implication of this is that mutual recursion and normal recursion is easily handled as we can be certain that any recursive type binding will exist in the environment. Haskell arguably has the most elegant solution for allowing local untyped mutual recursion ***, however a very advanced type system is outside of the scope of this project; Ocaml, F# and several other languages make use of the `letrec` and `with` keywords to solve this problem where 2 let expressions are explicitly defined by the developer as being mutually recursive, however I want to stay as close to a minimal Haskell-like syntax as possible so want to avoid introducing these. `where` blocks are in fact just syntactic sugar for several nested expressions where the body of the nested let expressions is the function body expression, so an extra static check can be performed of ensuring that no 2 where definitions are mutually recursive i.e. ensure they do not both reference each other in their body expressions.

D.1.5 Generic types

There are actually 2 different kinds of type variables that must be distinguished: regular type variables (`TyVar Int Kind` constructor) whose values the unification algorithm solves for, and *generic* type variables (`GenVar Int` constructor) which represent types that can be substituted for another type, which is the reason why functions like `length` with a type signature of `[a] -> Int` can be used on lists of `Int`s, `Bool`s etc. This process of replacing the occurrences of the generic type variables with normal unique `TyVar` constructors that are compatible with the unification algorithm is known as *instantiation*. The only place where instantiation takes place is when performing an environment lookup; by always replacing generic type variables with fresh, unique type variables, the fresh type variables can "assume" the role of any type they are unified with and be substituted with that type. Instantation is trivial, we simply extract a list of all generic type variables contained within a type and substitute them with freshly generated type variables provided by the stateful context, if the type contains no generic type variables then the instantiation function leaves the original type unchanged and simply returns its argument. Below I give a sort of "storyboard" of what happens during inference when a variable is looked up:

`map`	`(GenVar 0 → GenVar 1) → [GenVar 0] → [GenVar 1]`
1: A variable is detected during inference	2: The resulting type from the environment lookup containing generic type variables

`[GenVar 0, GenVar 1]`	`{GenVar 0 : TyVar 23, GenVar 1 : TyVar 24}`
3: A list of generic type variables that appear are extracted from the type	4: A mapping is generated of the generic type variables to be replaced, and their freshly generated replacement regular `TyVars` (So here, the counter in the stateful context was 23 and is now 25)

`(TyVar 23 → TyVar 24) → [TyVar 23] → [TyVar 24]`, ∅
5: The `infer` function returns the instantiated type and the constraints generated, recall in the variable expression rule that no constraints are generated from inferring a variable

D.1.6 Miscellaneous

The rule for inferring the type of a tuple expression is trivial, we simply infer the type of each of the element expressions and type the expression as a structured tuple type $(t_0, ..., t_i)$ of each of these inferred types. Tuples are structured types in the same way that expression applications are, these can be broken down and solved in unification explained next. List generators only handle Ints so we simply ensure that the "from", "to" and "interval" (if one is specified) expressions are Ints.

TyTuple

$$\frac{\Gamma \vdash e_0 :: t_0 \dashv C_0 \qquad ... \qquad \Gamma \vdash e_n :: t_n \dashv C_n}{\Gamma \vdash (e_0, ..., e_n) :: (t_0, ..., t_n) \dashv C_0 \cup ... \cup C_n}$$

TyListGen

$$\frac{\Gamma \vdash x :: t_0 \dashv C_0 \qquad \Gamma \vdash y :: t_1 \dashv C_1 \qquad \Gamma \vdash interval :: t_2 \dashv C_2}{\Gamma \vdash [x, interval..y] :: [\texttt{Int}] \dashv \{t_0 \doteq \texttt{Int}, t_1 \doteq \texttt{Int}, t_2 \doteq \texttt{Int}\} \cup C_0 \cup C_1 \cup C_2}$$

D.1.7 Records

Records in Syphon are somewhat limited compared to Haskell as each record type can contain only a single data constructor, which simplifies the inference rules greatly. Ideally I would have liked to implement an advanced record type system with support for field overloading as pioneered by Purescript ***, however this is by no means trivial to implement and is outside the scope of this project. Type checking records is straightforward enough, however we require an extended, slightly more complex environment. The environment in fact has several different sections which will be introduced later such as for handling optional structures and traits, and we must handle interaction with each of these sections differently. Let the aforementioned `lookup` function denote a normal environment lookup in the "simple" part of the environment for a variable's type binding, and let `lookupFromFields` be introduced as a function which looks up the type of a record by a list of fields referenced in the section of the environment responsible for storing record information, which returns a pair of the type of the record and the expected types of each field. The record constructor rule states that the inferred type of each expression given for each field must match its expected type; note that fields may be given in any order, hence the constraint set relies on equality of field names rather than order, for which I use the notation $\texttt{field}_x$ == $\texttt{field}_y$. Obviously `lookupFromFields` may fail as in a normal lookup, if so then an error message is returned stating that no record type exists with the given fields.

TyRecConstruct

$$\frac{\begin{array}{c}\Gamma \vdash e_0 :: t_{a0} \dashv C_0 \qquad ... \qquad \Gamma \vdash e_n :: t_{an} \dashv C_n \\ t_{Rec}, \{field_0 :: t_{b0}, ..., field_n :: t_{bn}\} = \texttt{lookupFromFields}^!(\Gamma, [field_0, ..., field_n])\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{Constr}\ \{field_0 = e_0, ..., field_n = e_n\} :: t_{Rec} \dashv \{t_{ax} \doteq t_{by}, field_x == field_y \quad x, y \in \{0...n\}\} \cup \{\texttt{lookup}^!(\Gamma, \texttt{Constr}) \doteq t_{Rec}\} \\ C_0 \cup ... \cup C_n\end{array}}$$

The rule for field replacement syntax is similar, we just need to add a constraint that the record variable which is having it's fields replaced is infact the type of the record:

TyFieldReplace

$$\frac{\begin{array}{c}\Gamma \vdash e_0 :: t_{a0} \dashv C_0 \qquad ... \qquad \Gamma \vdash e_m :: t_{am} \dashv C_m \\ t_{Rec}, \{field_0 :: t_{b0}, ..., field_n :: t_{bn}\} = \texttt{lookupFromFields}^!(\Gamma, [field_0, ..., field_m])\end{array}}{\begin{array}{c}\Gamma \vdash \{x \ \mid \ field_0 = e_0, ..., field_m = e_m\} :: t_{Rec} \dashv \{\texttt{lookup}^!(\Gamma, x) \doteq t_{Rec}\} \cup \{t_{ay} \doteq t_{bz}, field_y == field_z \quad y, z \in \{0...m\}\} \\ \cup C_0 \cup ... \cup C_m\end{array}}$$

Note the different number of fields m and n, more specifically $m \leq n$ due to the fact that any number of the record fields are replaced, perhaps only one. `lookupFromFields` handles searching with a subset of fields just fine as record fields are all unique to only one record type within the same module. The field selection rule is trivial, the rule is very similar to a standard variable lookup except we are searching the record section of the environment, the only additional constraint is ensuring the type of the expression that we are selecting from is the expected record type:

$$\frac{\Gamma \vdash e_0 :: t_0 \dashv C_0 \qquad t_{Rec}, \{\ldots, field_i :: t_i, \ldots\} = \texttt{lookupFromFields}^{!}(\Gamma, [field_i])}{\Gamma \vdash e_0.field_i :: t_i \dashv \{t_0 \doteq t_{Rec}\} \cup C_0}\ \text{TyFieldSelect}$$