Pablo Lanza Serrano

Reacting à la Mode: An Intuitive Functional Reactive Programming Language for Graphical Applications

Computer Science Tripos – Part II

Pembroke College

19th of January, 2023

Declaration of originality

I, Pablo Lanza Serrano of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT.

I am content for my dissertation to be made available to the students and staff of the University.

Signed *Pablo Lanza Serrano*

Date *11th of May, 2023*

Proforma

Candidate Number:	**2364F**
Project Title:	**Reacting à la Mode: An Intuitive Functional Reactive Programming Language for Graphical Applications**
Examination:	**Computer Science Tripos – Part II, 2023**
Word Count:	**11998**[1]
Code Line Count:	**8491**[2]
Project Originator:	**The Dissertation Author**
Supervisor:	**Dr. Jeremy Yallop**

Original Aims of the Project

Functional Reactive Programming (FRP) was originally offered in 1997 as an alternative for Graphical User Interface (GUI) programming over established frameworks. In the over 25 years since, the programming paradigm has failed to get a foothold on the GUI development landscape.

A recent paper proposes a novel conception for an FRP language using modal types and with a focus on convenience. This project aims to implement and extend this language to provide an intuitive and appealing FRP language for a widespread userbase. A user study evaluation will then asses the language's viability as an alternative for GUI programming.

Work Completed

The project was successful having satisfied all success criteria. Oters, a Functional Reactive Programming language based on a modal type system, was designed and implemented. The language ensured causality, productivity, and an absence of space leaks, and was further extended with a foreign function interface and a graphics API.

Oters proved to be robust enough to develop several complex applications, keeping to a concise and intuitive programming style. Moreover, it was found through a user study, that Oters' streams abstraction offered a natural approach for describing reactive behaviors. However, some functional limitations in the design of the language were identified.

[1] This word count was computed by `texcount (https://app.uio.no/ifi/texcount/)`

[2] This line count was computed by `cloc` (`https://github.com/AlDanial/cloc`)

Special Difficulties

None

Contents

Chapter 1

Introduction

The main way people interact with computers is through Graphical User Interfaces (GUIs). Their functions are manifold: displaying information to the user, capturing and reacting to their input, changing program state and providing corresponding feedback to the user. These are all components a successful GUI must implement, thus resulting in a complex system. Unfortunately, this complexity is also reflected in their development.

GUI programming is aided by the use of toolkits and frameworks, yet these suffer from myriad issues. For example, take the Model-view-viewmodel (MVVM) architectural pattern which defines a clear separation between the development of the back-end logic, the GUI and the communication between the two. The MVVM pattern has seen widespread adoption, especially within a web setting with frameworks like React, Vue.js and Svelte. While the separation between logic and graphics allows for changes in the view without having to adjust the model, it makes any changes beyond simple cosmetic ones tedious [26]. Implementations of this pattern also require substantial boilerplate code and for the user to have deep knowledge of the underlying system as the interactions between the GUI and logic are hidden from the programmer [17].

MVVM frameworks are an example of event-driven programming where user input is captured through event listeners and handled in a main loop. Event-driven programming has become the dominant paradigm used in GUI applications [16]. The use of higher-order functions that this paradigm necessitates is poorly implemented in many of the GUI toolkits written in imperative languages like C or Java [13]. Moreover, when building event-driven applications with interconnected components, this style of programming results in a complex network of shared mutable state and event callbacks [6]. Consequently, the individual components' behaviors are hard to reason about, further hindering the development and maintenance of the GUI.

1.1 Functional Reactive Programming

Having identified these issues, many people have proposed functional programming as a solution. The declarative style of this programming paradigm makes it easier to reason about the behavior of a complex GUI, while also providing a natural higher-order programming interface that can bridge the gap between graphics and logic. Functional Reactive

Programming (FRP) was thus conceived by Elliott and Hudak [9] to provide a natural way of programming GUIs, limiting the issues outlined above. Oters, the programming language designed and developed in this paper, follows this paradigm.

However, functional programming introduces a dilemma: On the one hand GUIs are seemingly composed of mutable state as the screen's contents change with user input. On the other hand, functional programming eschews mutable state. To remedy this inconsistency, FRP introduces its core data type: the *stream*. Streams seek to capture user input behaviors, as well as ensuing messages between system components, as time varying values. These can be represented as the following recursive, polymorphic type,

$$\mathsf{Stream}\ A \triangleq A \ll \mathsf{Stream}\ A$$

Streams are composed recursively using the ($\ll$) operator, analogously to lists in OCaml, where we have a head of type A and a recursive tail of type $\mathsf{Stream}\ A$. However, in contrast to lists, streams capture a notion of time, where the data in the tail will be produced at a later time to the data in the head.

These streams are first-class values that can be constructed into new streams through the rich set of operators defined in the FRP canon. A collection of such streams make up an FRP program, with the underlying system automatically updating the streams as the program runs.

The following is a simple program showing how streams are created and combined:

```
1 let from = fn n -> n << from (n+1)
2 let times_two = fn (x << xs) -> x*2 << times_two xs
3 let evens = times_two (from 0)
```

This programming style should be familiar to those with experience in functional programming. Line 1 recursively constructs a stream of increasing integers. Line 2 performs pattern matching on an input stream to double each element. Line 3 composes the two functions to create a stream of even numbers. Moreover, the language runtime has a notion of time, such that the streams `(from 0)` and `evens` above are updated alongside each other. As we will see, streams are a powerful way capturing the complex behavior of GUIs.

1.2 Previous Work

The original conception of FRP was defined over 25 years ago in the language Fran [9], which introduced the concept of streams (albeit under the names of *Behaviors* and *Events*) and how they can be used to define animated reactive behavior. Since Fran, FRP has branched and specialized into many different implementations [5]. Improvements made in subsequent iterations aim to formalize a more rigorous type calculus to ensure the following properties [25] [13]:

1. **Causality:** Definitions of temporal data should only depend past data and contain no reference to future data. In other words, the n^{th} output should depend only on the first $n - 1^{\text{th}}$ inputs.

2. **Productivity:** When relying on a stream as input, the stream should always produce an output in finite time. So, for all n time-steps we want our streams to output something.

3. **No Space Leaks:** Programs should not leak memory even though streams constantly produce new data. We want to ensure programs take up constant space after reaching a stable state unless explicitly specified by the programmer.

To combat these issues, arrowized FRP was conceived, where the expressivity of the FRP system was reduced to make 'leaky' programs harder to write. The embedded language Yampa [21] was the result of this research which no longer has streams as first-class values, but instead limits streams to be manipulated exclusively using stream-processing functions. Yampa removes implicit space leaks, at the cost of the simplicity and flexibility of the original FRP model [3].

Aiming to reformulate FRP, and keeping with the traditional model of having streams as first-class values, Krishnaswami employed a modal type system to ensure the three properties outlined above [13] [12]. Finally, Bahr refined this type system with a focus on simplicity, practicality, and expressiveness [4]. This was achieved by adapting the modal type system to use a Fitch-style type system that extends typing contexts with tokens to avoid overheads introduced in Krishnaswami's type system [3].

This outline of a narrow branch of FRP research illustrates the great interest in seeing this programming paradigm succeed. Nevertheless, FRP still seems to be restricted to a niche corner of GUI development, having yet to catch on in the mainstream development community.

1.3 Aims

The overarching aim of this dissertation is to provide an intuitive and appealing Functional Reactive Programming language for a widespread userbase. I aim to:

1. Define and implement a programming language with a domain focus on GUI programming, based on Rattus' type system defined by Bahr [3].

2. Determine the language's **safety** through memory profiling, **expressiveness** by using it to create complex applications, and **usability** via a user study.

3. Evaluate these results to ascertain the effectiveness of Oters and the FRP paradigm for GUI programming.

Chapter 2

Preparation

We begin by comparing a naive implementation of streams to Bahr's type system for his language Rattus [3] which Oters extends. This analysis shows how Rattus avoids the pitfalls described in Section 1.2 of causality, productivity, and space leaks [12].

2.1 Naive Streams

Take the definition for a polymorphic stream type used above:

$$\mathsf{Stream}\ A \triangleq A \ll \mathsf{Stream}\ A$$

Here, we us a notion of time that is discretized into individual time-steps such that the head of type A is produced one time-step before the tail of type $\mathsf{Stream}\ A$. This model implies streams defined in a program are all updated synchronously with each time-step.

2.1.1 Violating Causality

Using this definition, we can define a head or tail function for a stream using pattern matching:

```
let tail = fn (x << xs) -> xs
```

This definition takes a stream as input at some time-step t, and returns its tail `xs`. `xs`, however, holds the stream's value at time-step $t + 1$. So by applying this function to a stream, we bring the value from $t + 1$ forward to time t, violating causality.

Now, say `mouse_pos` is a stream describing the user's mouse position. At each time-step, `mouse_pos` is updated such that its stream's head holds the current position of the mouse. By applying `tail` to `mouse_pos`, we receive the user's future mouse inputs. Clearly, we want to disallow this program as we cannot use a value that is not yet available.

2.1.2 Violating Productivity

A running FRP program constantly re-evaluates each stream every time-step. Therefore, for our program to be reactive, we want to ensure that this re-evaluation happens in finite

time. In other words, so that our program stays responsive, any computation done within a stream must not result in an infinite loop.

We therefore want to prevent defining the following stream, which will block execution as it loops endlessly within one time-step without producing any output.

```
let loop = fn x -> loop x
let blocking = loop () << blocking
```

2.1.3 Introducing Space Leaks

A space leak occurs when a program's memory use increases as time progresses. In particular, we are concerned with implicit space leaks, i.e., those unintended by the programmer and caused by the underlying FRP system retaining stale data.

In the following program, the function `leak` will produce a space leak when applied to a stream.

```
let from = fn n -> n << from (n+1)
let nats = from 0
let leak = fn xs -> head xs << leak xs
```

The space leaks produced become apparent when we evaluate `leak nats` over multiple time-steps. Below we see how each recursive call to `leak` retains all the n values computed by `nats`, leaking memory:

Time	Computation of `leak nats`

0	`leak nats = head nats << leak nats`
	`= head (from 0) << leak(from 0)`
	`= head (0<<from 1) << leak (0<<from 1)`
	`= 0 << leak (0<<from 1)`

1	`leak nats = leak (0<<from 1)`
	`= head (0<<1<<from 2) << leak (0<<1<<from 2)`
	`= 0 << leak (0<<1<<from 2)`

⋮	⋮
n	`leak nats = 0 << leak (0<<...<<n<<from (n+1))`

Table 2.1: Example of a Computation with Space Leaks

If we can ensure that our streams uphold these three properties, then we know any program is implemented correctly (all streams are causal), executed properly (all streams are productive), and uses memory correctly (no space-time leaks).

2.2 Rattus

Krishnaswami developed a type system which gives the desired causality, productivity and space-leak guarantees [13] by extending the Simply-Typed Lambda Calculus with a

modal type system, and a guarded fixed point operator. Bahr adapted this type system to use of a Fitch-style typing context, to provide a simpler but equally powerful calculus [4]. Oters extends Bahr's third iteration of a FRP calculus, called Rattus [3].

2.2.1 Modal Types

In Rattus streams are defined as follows:

$$\begin{aligned}\mathsf{Stream}\ A &\triangleq A \ll \bigcirc(\mathsf{Stream}\ A)\\ &\simeq A \ll \bigcirc(A \ll \bigcirc(A \ll \bigcirc(\cdots)))\end{aligned}$$

The difference with this definition here is the inclusion of the modal operator $\bigcirc$, which captures explicitly the temporal aspect of streams. A type $\bigcirc A$ signifies that some data of type A will arrive in the next time-step. So in our new definition for streams, the tail will be computed in the next time-step, introducing a temporal gap of one time-step between consecutive elements.

To introduce the $\bigcirc$ type modality we use the Delay operator. Elements behind Delay must be evaluated in the next time-step. So our definition of `from` from earlier becomes:

$$from = \lambda n.n \ll \mathsf{Delay}(from\ (n+1))$$

The other type modality defined is $\square$, which marks a type as stable [12]. A stable type is any type that is time-independent. Primitive types are stable and this property is inherited by product and sum types entirely composed of stable types. Crucially however, streams and functions are not stable. For streams, the use of $\bigcirc$ introduces a dependence on time, whereas functions can capture time-dependent data in their closures, so we cannot be sure they are always time-independent. However, we can add the aforementioned $\square$ modality to turn unstable types into stable types (e.g. $\square(A \to B)$).

The $\square$ modality can be introduced with the Box operator, and is important when using functions as first-class values. To define a map function over a stream, we want to be able to apply its functional argument throughout the duration of the stream's lifetime. This requires our function to be stable, or else risk leaking memory as it carries temporal data in its closure into future time-steps. We must therefore apply Box to the functional argument.

2.2.2 Variable Contexts and Typing Rules

To implement the desired semantics of this modal type system, Rattus uses a Fitch-style typing context which aside from the standard variable-type mappings, also includes tokens that introduce the time modality [8].

$$\text{Term Variable Contexts} \qquad \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \checkmark$$

The $\checkmark$ in these contexts is used to delimit variables that are defined behind a $\bigcirc$ and are therefore only available in the next time-step. Only one $\checkmark$ is allowed within a context at a time.

Using this typing context we give typing rules for the Delay and Box operators.

$$\frac{|\Gamma|, \checkmark \vdash e : T}{\Gamma \vdash \mathsf{Delay}\ e : \bigcirc T}\textsc{Delay} \qquad \frac{\Gamma^{\square} \vdash e : T}{\Gamma \vdash \mathsf{Box}\ e : \square T}\textsc{Box}$$

Where the context transformations $\Gamma^{\square}$ and $|\Gamma|$ are defined as follows:

$$\cdot^{\square} = \cdot \qquad (\Gamma, x : T)^{\square} = \begin{cases} \Gamma^{\square}, x : T & \text{if } T \text{ stable} \\ \Gamma^{\square} & \text{otherwise} \end{cases} \qquad (\Gamma, \checkmark)^{\square} = \Gamma^{\square}$$

$$|\Gamma| = \Gamma \quad \text{if } \Gamma \text{ is tick-free} \qquad |\Gamma, \checkmark, \Gamma'| = \Gamma^{\square}, \Gamma'$$

The Delay rule introduces a ✓ saying that all variables defined inside a Delay are restricted to being used in the next time-step. The $|\Gamma|$ transformation turns Γ into a tick-free context to ensure only one ✓ is present for nested Delays. $|\Gamma|$ also prevents temporal values from, say, time-step $n+2$ to be available alongside temporal values from time-step $n+1$ in the same context, which would violate causality.

The Box rule restricts all variables inside a Box to be stable. This has the effect of leaving any unstable free variables out of scope in a stable function's body.

We must also restrict the rules for functions and their variables.

$$\frac{A \text{ stable} \vee \Gamma' \text{ tick-free}}{\Gamma, x : A, \Gamma' \vdash x : A}\textsc{Hyp} \qquad \frac{|\Gamma|, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}{\to}\mathrm{I}$$

Unless a variable is stable, the Hyp rule restricts it to only being accessed in the time-step it is declared in, since it must be to the right of a ✓ as Γ' is tick-free. The $\to$I rule removes any unstable variables from previous time-steps from the context of a function's body. This ensures that any functions declared inside a Delay expression only contain time-dependent variables from this later time-step in their closure. The purpose of such restrictive typing rules is to disallow space leaks.

2.2.3 Adv Expressions

We want streams to be first-class values to arbitrarily manipulate and transform them in functions. This can be done by applying a transformation recursively to the head of a stream. Consider the following example that doubles all the elements of an int stream in this manner:

$$double = \lambda(x \ll xs).x \times 2 \ll \mathsf{Delay}(double\ xs)$$

Here, $(x \ll xs)$ is syntactic sugar to unwrap a stream argument into its head and tail.

This function fails to type check since *double* has type int Stream $\to$ int Stream whereas xs has type $\bigcirc$(int Stream), and so the types do not match when applying *double* xs. Hence, Bahr et al. introduce the Adv operator [4] which eliminates the $\bigcirc$ modality:

$$double = \lambda(x \ll xs).x \times 2 \ll \mathsf{Delay}(double\ \mathsf{Adv}(xs))$$

We must be careful with the typing rule for Adv expressions to ensure causality. The function

$$from\_the\_future = \lambda(x \ll xs).\mathsf{Adv}(xs)$$

violates causality in the same manner as the naive implementation of `tail` from 2.1.1. We remedy this by ensuring that all Adv expressions happen inside a Delay. This corresponds to the following type rule which restricts Adv expressions to only occur in the presence of a ✓:

$$\frac{\Gamma \vdash e : \bigcirc T}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{Adv}\ e : T}\textsc{Adv}$$

Moreover, we can think of the Adv operator as grabbing a reference to a value from the current time-step that will only become available later. This advances it to be used in the next time-step once it becomes available. Since Adv permits us to go back to the current time-step from inside a Delay, any expression inside the Adv must lose access to all future variables. Thus, the typing rule Adv removes the variables to the right of the ✓ (e.g. Γ') from e's scope, as these were introduced in the next time-step.

2.2.4 Guarded Recursion

The typing rules presented above ensure causality and prevent space-leaks. To ensure productivity, Rattus employs *guarded recursion*, which restricts any recursive function calls to occur inside a Delay. This means that looping within the same time-step is disallowed, by definition of Delay. Thus, we ensure productivity, since infinite loops are thereby also eliminated.

Guarded recursion must be implemented along with guarded recursive types [20]. These types are formed using the guarded fix point combinator: $\mathsf{Fix}\ \phi.A$. To ensure productivity, all recursive types must be guarded recursive, including streams. This results in the following stream type representation:

$$\begin{aligned}\mathsf{Stream}\ A &\triangleq \mathsf{Fix}\ \phi.\ A \times \phi \\ &\simeq A \times \bigcirc(\mathsf{Fix}\ \phi.\ A \times \phi)\end{aligned}$$

This shows that when unfolding a guarded recursive type, we implicitly insert the ◯ modality. When a stream is constructed recursively, its type ensures that the tail will be computed in the next time-step through the implicit insertion of ◯.

Earlier we constructed streams using the operator ($\ll$). We now define $x \ll xs$ as syntactic sugar for $\mathsf{Into}\ \langle x, xs\rangle$. The $\langle\cdot,\cdot\rangle$ simply constructs a pair, matching the product type in the definition above. The into operator can be seen as rolling an expression into a Fix type, and has the following typing rule:

$$\frac{\Gamma \vdash e : [\bigcirc(\mathsf{Fix}\ \phi.\ T)/\phi]T}{\Gamma \vdash \mathsf{Into}\ e : \mathsf{Fix}\ \phi.\ T}\textsc{Into}$$

The substitution in the premise's type shows how e has the type of an unrolled fixed point type, where ϕ has been replaced with the full type behind the ◯ implicitly inserted. Thus, $\mathsf{Into}\ \langle x, xs\rangle$, takes the expression $\langle x, xs\rangle$ of type $T \times \bigcirc(\mathsf{Fix}\ \phi.\ T \times \phi)$ and converts it to just $\mathsf{Fix}\ \phi.\ T \times \phi$ — a stream.

The expression used to construct guarded recursive types is the guarded fixed point operator $\mathsf{fix}\ r.\ e$. This operator introduces a recursion variable r which holds a copy of the guarded fixed point expression $\mathsf{fix}\ r.\ e$. This variable can then be used inside of e to perform the recursion.

The precise semantics of the fix operator is captured in its typing rule:

$$\frac{\Gamma^{\square}, r : \square(\bigcirc T) \vdash e : T}{\Gamma \vdash \mathsf{fix}\ r.\ e : T}\textsc{Fix}$$

We first notice that the type assigned to r is $\square(\bigcirc T)$, as opposed to the fixed point expression's type: T. The $\bigcirc$ modality is added to ensure that any recursive computations occur in the next time-step. We also add the $\square$ modality so that the recursion variable is available at any time-step. The body of the fixed point expression e will be copied into future time-steps. To prevent capturing time-dependent variables and risk leaking old data we restrict the context to $\Gamma^{\square}$.

All recursive expressions have this guarded fixed point operator implicitly inserted to convert them into guarded recursive types and therefore ensure productivity. This aggressive strategy of always inserting the fix operator is necessary since it is undecidable to tell if a given recursive function terminates. For example, the $from$ function we defined earlier, would be converted to:

$$from = \mathsf{fix}\ r.\lambda.n \ll \mathsf{Delay}(\mathsf{Adv}(\mathsf{Unbox}(r))(n+1))$$

When we insert the fixed point operator, we must also replace any instance of a recursive call with the recursion variable. However, looking back at the typing rule Fix, we see that r has type $\square(\bigcirc(\mathsf{int} \to \mathsf{int\ Stream}))$ and so we cannot directly apply the argument $(n+1)$. Thus, we must unwrap the modalities hiding the underlying function: Adv removes the $\bigcirc$ modality, and the Unbox operator which simply unwraps the $\square$ modality in the obvious manner.

2.2.5 Abstract Machine

The type system rejects programs that may introduce implicit space leaks through the use of modal types. A proper implementation, formalized as an abstract machine, is further needed to run accepted programs in a manner that does not leak memory. It is in the operational semantics that we eliminate space leaks by deleting all data from the previous time-step.

The abstract machine operates as follows: To segregate the old data, we use a store separated into two heaps: the "*now*" heap and the "*later*" heap. These two heaps match the notion of time described by the $\bigcirc$ operator. When we write some expression Delay e, we store e in the *later* heap to be evaluated in the next time-step. Then, once we advance a time-step, the *later* heap, becomes the *now* heap. The e we stored earlier can now be retrieved from the now heap using Adv e expression. Moreover, when the heap switching is performed, the previous *now* heap is destroyed, deleting all data from the previous time-step, and thus preventing any implicit space leaks.

The operational semantic rules show how streams are updated by this abstract machine. The rules fall into two categories: the *evaluation semantics* which reduce an expression within a single time-step, and the *step semantics* which describe how an expression progresses from one time-step to the next [3].

The evaluation semantics for the Delay and Adv expressions which formally define their behavior described above are presented here:

$$\frac{l \notin \mathrm{dom}(\eta_L)}{\langle \mathsf{Delay}\ e; \eta_N \checkmark \eta_L \rangle \Downarrow \langle l; \eta_N \checkmark \eta_L, l : e \rangle}\textsc{DelayEv}$$

$$\frac{\langle e; \eta_N \rangle \Downarrow \langle l; \eta'_N \rangle \qquad l : e' \in \eta'_N \qquad \langle e'; \eta'_N \checkmark \eta_N \rangle \Downarrow \langle v; \eta''_N \checkmark \eta'_L \rangle}{\langle \mathsf{Adv}\ e; \eta_N \checkmark \eta_L \rangle \Downarrow \langle v; \eta''_N \checkmark \eta'_L \rangle}\textsc{AdvEv}$$

In the evaluation semantics we thread the store $\eta_N \checkmark \eta_L$, where η_N is the *now* heap and η_L the *later* heap. Each heap is a map from locations to expressions. The DelayEv rule specifies that we allocate an expression e to the *later* heap by appending a mapping from some fresh location l to the expression e.

The AdvEv rule retrieves the contents e' behind location l, reducing this expression to some value v. These are the only two evaluation rules that access and transform the store. Thus, we only write to the later heap, and only read from the now heap.

The step semantics are similarly defined with inductive rules, but are only used to update expressions of type Stream A and advance them in time with the following rule:

$$\frac{\langle e; \eta \checkmark \rangle \Downarrow \langle v \ll l; \eta_N \checkmark \eta_L \rangle}{\langle e; \eta \rangle \overset{v}{\Longrightarrow} \langle \mathsf{Adv}\ l; \eta_L \rangle}\textsc{StrStep}$$

StrStep takes an expression e and a store η solely comprised of a *now* heap. Before evaluating e, we add a $\checkmark$ to the store so that any new value allocations are added to the *later* heap. e is then evaluated to a stream $v \ll l$, where $\cdot \vdash v : A$ and $l : e' \in \eta_L$ and $\cdot \vdash e' : \mathsf{Stream}\ A$. Then when we move forward a time-step (indicated by $\Longrightarrow$), we replace the stream with the expression Adv l and emit the stream's head v. Additionally, the *now* heap η_N is destroyed leaving the *later* heap η_L behind. In the following time-step, when we evaluate Adv l, by AdvEv, we retrieve and evaluate the computation describing the previous stream's tail from location l located in the new *now* heap η_L.

The example below, shows how the store does not grow with time. In fact, there is a one-to-one correspondence between the store locations across time-steps.

$$\begin{aligned}
&\mathit{from} = \mathsf{fix}\ r.\lambda n.n \ll \mathsf{Delay}(\mathsf{Adv}(\mathsf{Unbox}\ r)(n+1)) \\
&\mathit{double} = \mathsf{fix}\ r.\lambda(x \ll xs).x \times 2 \ll \mathsf{Delay}(\mathsf{Adv}(\mathsf{Unbox}\ r)\ \mathsf{Adv}(xs)) \\
&\mathit{evens} = \mathit{double}\ (\mathit{from}\ 0)
\end{aligned}$$

$$
\begin{aligned}
&\langle evens; \emptyset\rangle \rightsquigarrow \langle double\ (from\ 0); \checkmark\rangle \\
&\qquad \hookrightarrow \langle (from\ 0);\ \checkmark\rangle \Downarrow \langle 0 \ll l_2^{(1)};\ \checkmark,\ l_1^{(1)} : from,\ l_2^{(1)} :\ \mathsf{Adv}(l_1^{(1)})\ (0+1)\rangle \\
&\Downarrow \langle double\ (0 \ll l_2^{(1)});\ \checkmark,\ l_1^{(1)} : from,\ l_2^{(1)} :\ \mathsf{Adv}(l_1^{(1)})\ (0+1)\rangle \\
&\Downarrow \langle 0 \ll l_4^{(1)};\ \checkmark,\ l_1^{(1)} : from,\ l_2^{(1)} :\ \mathsf{Adv}(l_1^{(1)})\ (0+1),\ l_3^{(1)} : double, \\
&\qquad\qquad l_4^{(1)} :\ \mathsf{Adv}(l_3^{(1)})\ \mathsf{Adv}(l_2^{(1)})\rangle \\
&\overset{0}{\Longrightarrow} \langle \mathsf{Adv}\ l_4^{(1)};\ l_1^{(1)} : from,\ l_2^{(1)} :\ \mathsf{Adv}(l_1^{(1)})\ (0+1),\ l_3^{(1)} : double, \\
&\qquad\qquad l_4^{(1)} :\ \mathsf{Adv}(l_3^{(1)})\ \mathsf{Adv}(l_2^{(1)})\rangle \\
&\Downarrow \langle \mathsf{Adv}\ l_4^{(1)};\ \eta_N\checkmark\rangle \\
&\Downarrow \langle double\ (from\ (0+1));\ \eta_N\checkmark\rangle \\
&\Downarrow \langle 2 \ll l_4^{(2)};\ \eta_N\checkmark,\ l_1^{(2)} : from,\ l_2^{(2)} :\ \mathsf{Adv}(l_1^{(2)})\ (0+1),\ l_3^{(2)} : double, \\
&\qquad\qquad l_4^{(2)} :\ \mathsf{Adv}(l_3^{(2)})\ \mathsf{Adv}(l_2^{(2)})\rangle \\
&\overset{2}{\Longrightarrow} \langle \mathsf{Adv}\ l_4^{(2)};\ l_1^{(2)} : from,\ l_2^{(2)} :\ \mathsf{Adv}(l_1^{(2)})\ (0+1),\ l_3^{(2)} : double, \\
&\qquad\qquad l_4^{(2)} :\ \mathsf{Adv}(l_3^{(2)})\ \mathsf{Adv}(l_2^{(2)})\rangle \\
&\overset{4}{\Longrightarrow} \langle \mathsf{Adv}\ l_4^{(3)};\ l_1^{(3)} : from,\ l_2^{(3)} :\ \mathsf{Adv}(l_1^{(3)})\ (0+1),\ l_3^{(3)} : double, \\
&\qquad\qquad l_4^{(3)} :\ \mathsf{Adv}(l_3^{(3)})\ \mathsf{Adv}(l_2^{(3)})\rangle
\end{aligned}
$$

2.3 Requirements Analysis

The requirements of the project were dictated by the project proposal's success criteria (Appendix D). The core of the implementation involved building the compiler front-end and an interpreter for a FRP language. This language was then extended with a Rust foreign function interface (FFI), and GUI programming functionality.

Development mainly followed a linear path of dependencies, with the main components outlined in Figure 2.1.

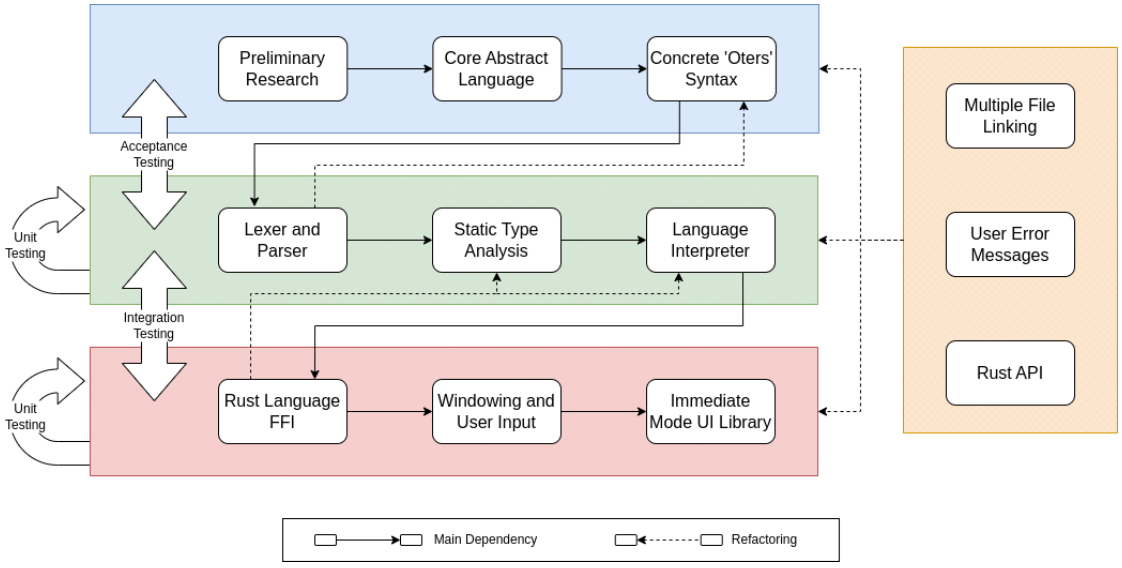


Figure 2.1: The Implementation's Development Work Flow

This diagram includes backwards dependencies to illustrate how later components needed to be integrated with previous ones, requiring refactoring. Keeping this in mind, the initial development of, say, the interpreter, was done with extensibility in mind to accommodate the Rust FFI and file linking.

The testing required is also illustrated here. Specifically, I show how the language front-end and interpreter needed acceptance testing such that the causality, productivity and implicit space leak properties held. Integration testing was further required between the bottom group of components and the language implementation to ensure full compatibility between the language, and the FFI and graphical libraries.

At the start of each component, I analyzed the work to be completed, and subdivided the component into tasks, each assigned a priority and a risk. This process is shown in Table 2.2:

Task	Priority	Risk
Fitch-Style variable context	High	Low
Type check expressions	High	Low
Unit testing	High	Low
Polymorphic types	High	Medium
Well-formedness checks	Medium	Low
User defined types	Medium	Medium
Type check linked files	Low	High
Full type inference	Low	High
Type trait for *Stable* typed function arguments	Low	Medium

Table 2.2: Task division for the Static Type Analysis Module

2.4 Software Engineering

2.4.1 Development Model

Due to the modular nature of the project lends the Scrum software development methodology was used. The linear dependencies in the project's structure fit the iterative and incremental nature of the scrum framework.

Figure 2.1's colored boxes illustrate the components assigned to each 3 to 4 week long sprint. Each sprint began by identifying subtasks as shown in Table 2.2, and work proceeded in order of priority. Unforeseen challenges were added to a backlog. These and the subtasks identified at the beginning of each sprint, were tracked using a Kanban board to visualize the project's status. At the end of each sprint, the work done was reviewed so that future additions could be integrated seamlessly.

2.4.2 Software Tools

The project was implemented with Rust. This language was chose due to my prior proficiency, its current popularity [2] and its emerging ecosystem of GUI framework libraries.

It is also emphasizes performance with memory safety properties.

Rust's builder and package manager Cargo [1] was also used to compile, publish and distribute Oters. Cargo's additional tools such as a test command used to run and analyze unit tests, and *tarpaulin* [28], a code-coverage tool were also used. The project was programmed using the following Rust libraries:

Libraries	Purpose
`lalrpop`	Lexer, Parser and AST generator
`anyhow, thiserror`	Implementing and handling errors
`lazy-static`	Lazily initialize static data at runtime
`macroquad`	Graphics library
`quote, syn`	Rust tokenization for procedural macro creation
`chrono`	Date and time
`daggy`	Directed Acyclic Graphs

Table 2.3: Third-party crates used in the implementation

GitHub was also used for version control, with new branches being created whenever any tentative features were added. GitHub's Actions feature was used to automatically build and run all tests after each commit was made. GitHub served as a backup along side weekly backups of all relevant documents to Google Drive.

2.4.3 Software License

The code is freely available on GitHub under an MIT license since this license is most permissive and the project's intention is to be a proof-of-concept language. It is also compatible with the above libraries' licenses. This will allow anyone to extend Oters however they wish.

2.5 Starting Point

I had some experience with a compiler's front-end, having implemented some projects with OCaml's lexer and parser. Otherwise, I had no experience with building compilers, aside from the Compiler Construction Part IB course. Content from the Semantics and Further HCI courses was also relevant.

While researching for the project proposal, I read numerous research papers on Functional Reactive Programming, comparing the various interpretations of the paradigm [5].

Patrick Bahr made publicly available an implementation of the Rattus language implemented in Haskell. However, as it is directly embedded in Haskell's type system as part of the compiler toolchain it was not useful in the development of this project.

Chapter 3

Implementation

3.1 The Oters Language

My aim in designing Oters was to extend Bahr's Rattus, with two main focuses in mind:

1. **Convenience**: To allow users to write extensive programs without requiring a lot of boilerplate code nor excessive verbosity, while also not relying on the FFI with Rust to perform complex operations.

2. **Familiarity**: To provide a programming style that captures the intuitive reactive behaviors of FRP, while remaining familiar to programmers who have not previously come across this paradigm.

Oters implements the following types:

$$
\begin{aligned}
\text{Types} \quad & T ::= \alpha^s \mid \phi \mid 1 \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{string} \mid \mathsf{bool} \mid T_1 \times \cdots \times T_n \mid [T] \mid T_1 \to T_2 \mid \\
& \{k_1 : T_1 \times \cdots \times k_n : T_n\} \mid \{c_1(T_1) + \cdots + c_n(T_n)\} \mid \Box T \mid \bigcirc T \mid \\
& \mathsf{Fix}\ \phi.T \\
\text{Stable Types} \quad & S ::= \alpha^\Box \mid 1 \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{string} \mid \mathsf{bool} \mid S_1 \times \cdots \times S_n \mid [S] \mid \\
& \{k_1 : S_1 \times \cdots \times k_n : S_n\} \mid \{c_1(S_1) + \cdots + c_n(S_n)\} \mid \Box T
\end{aligned}
$$

The type system extends the Simply Typed Lambda Calculus with the modal types $\Box T$ and $\bigcirc T$, as well as guarded recursive types $\mathsf{Fix}\ \phi.T$. The type system was further extended to include a variety of data types that most programmers expect (integers, floating points, strings and booleans). Product types are unrestricted to an indefinite number of members. There is also a list type $[T]$. Finally, there are the record types, which map keys k_i to values of some type, and variant types which describe values that can take on different types indicated by a constructor c_i. These types are meant to resemble Rust's `struct` and `enum` types respectively, and from now on they will be referred to by these names.

Additionally, for the language to be truly expressive, type polymorphism is required. As I came to design the language syntax and implement the type checker, it became apparent that type inference was necessary (see Section 3.3). This means that the polymorphism would have to be restricted to "let-polymorphism" since type inference for an

unrestricted polymorphic type system is undecidable. Thus, we also include type schemes:

$$\forall \alpha_1^{s_1}, \ldots, \alpha_n^{s_n}.T$$

Type schemes are quantified over type variables $\alpha_i^{s_i}$, which can be substituted for concrete types once the polymorphic type is instantiated. Each type variable has a stability s_i, indicating whether the type that substitutes it must be stable or not. I write $\alpha^{\square}$ and α for type variables that respectively have and do not have the stable trait. The rule above also specifies that the body of a scheme is a type, and critically, not another scheme. For example, I use the `fold` function, which takes an accumulator function and an initial value, and repeatedly applies the accumulator to all the elements of a stream:

$$\texttt{fold} : \forall \alpha, \beta^{\square}.(\beta \to \alpha \to \beta) \to \beta \to \mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta$$

Here, the accumulator type β must be stable since the accumulator value is carried to be used in the next time-step. As soon as we apply some function to `fold`, the type variables are substituted for the appropriate concrete types:

$$\texttt{fold}\ (\lambda acc.\lambda x.acc + x) : \mathsf{int} \to \mathsf{Stream\ int} \to \mathsf{Stream\ int}$$

3.1.1 Expressions

Next, we define the full set of language expressions.

Operators	$op ::= + \mid - \mid \times \mid \div \mid \mathsf{mod} \mid :: \mid = \mid < \mid > \mid \wedge \mid \vee \mid \neg$
Expressions	$e ::= x \mid \langle\rangle \mid n \mid f \mid s \mid b \mid op\ e \mid e_1\ op\ e_2 \mid \langle e_1, \ldots, e_n \rangle \mid [e_1, \ldots, e_n] \mid \lambda x.e \mid$ $e_1\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1;\ e_2 \mid \{k_1 : e_1, \ldots, e_n : e_n\} \mid \pi_k(e) \mid$ $c(e) \mid \mathsf{match}\ e\ \mathsf{with}\ \{p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\} \mid \mathsf{let}\ x = e\ \mathsf{in}\ e_2 \mid$ $\mathsf{Delay}\ e \mid \mathsf{Adv}\ e \mid \mathsf{Box}\ e \mid \mathsf{Unbox}\ e \mid \mathsf{fix}\ x.e \mid \mathsf{Into}\ e \mid \mathsf{Out}\ e \mid l$
Patterns	$p ::= \_ \mid x \mid \langle p_1, \ldots, p_n \rangle \mid c(p) \mid p_1 \vee p_2$

First, the unary and binary operator expressions are not defined for all the operators. For example $e_1 \neg\ e_2$ is not defined and neither is $\div e$. Moreover, the arithmetic and comparative operators are overloaded for ints and floats. Further, there is no tuple projection expression. Type inference requires knowledge of tuple length which is not provided, and since Oters does not implement subtyping there is no type safe way of projecting tuples. Tuple terms must therefore be extracted by pattern-matching.

We also see here the inclusion of the modal operators Delay, Box, Adv and Unbox from Section 2.2. Similarly, we have the expression fix $x.e$ that introduces guarded recursion, as well as the into and out operators that fold and unfold respectively expressions of guarded recursive types. While these latter expressions are included in the language, they should rarely be used by the end programmer. Similarly, note that fix $x.e$ expressions are not exposed to the user, rather they are implicitly inserted with recursive definitions. Lastly, l expressions represent the locations left behind after evaluating a Delay e expression as described in the previous chapter. These are never written by the programmer but form part of the core abstract language that the interpreter implements.

Patterns are needed to unwrap tuples and variant types. These patterns are used in match expressions to bind variables either to the individual elements of a tuple, or to the underlying value of the corresponding variant via the $c(p)$ pattern. For example, in the following example, x gets bound in both patterns to the string `"Hello"`, and y gets bound to the string `"World!"` in the second pattern:

$$\begin{aligned}&\texttt{match } \langle \texttt{"Hello "},\ \texttt{Some("World")} \rangle \texttt{ with } \{\\ &\quad \langle x, \texttt{None} \rangle \Rightarrow \mathit{print}\ x,\\ &\quad \langle x, \texttt{Some}(y) \rangle \Rightarrow \mathit{print}\ x; \mathit{print}\ y\\ &\}\end{aligned}$$

3.2 Syntax and Parsing

The first stage of a standard compiler pipeline is the lexer and parser. These were implemented with the Rust library *Lalrpop* [14], a Look-Ahead LR parser generator framework.

The main challenge faced here was avoiding parsing conflicts, multiple instances of which informed the final design of the syntax. The following are parsing conflicts encountered, each of which was resolved in a different manner:

- **Subtraction and Negation**: Given an expression $e - e$, we get a reduce/reduce conflict where the parser has the following two options: either directly reduce to a subtraction expression, or reduce to a function application expression, followed by a negation expression. This was fixed by changing the negation operator from (`-`) to (`~`) like in Standard ML.

- **Nested Match Statements**: The original OCaml-style match statement syntax used was ambiguous with nested patterns: to which match does `p3` belong to below? This was resolved by using Rust's design of brace-delimiting match statements.

```
match e1 with
  | p1 => match e2 with
            | p2 => e3
        | p3 => e4
```

(a): OCaml Match Statement Syntax

```
match e1 {
  p1 => match e2 {
    p2 => e3
   },
  p3 => e4
}
```

(b): Rust Match Statement Syntax

- **Function and Variant Application**: Take the expression `f None 2` which applies a function `f` to two arguments: an option variant `None` and an integer `2`. The second argument `2` introduces some ambiguity as it could potentially be bound to the variant instead like so: `f (Some 2)`. This conflict was resolved by making variant expressions have lower precedence than function application.

The parser adds code location spans when constructing the Abstract Syntax Tree (AST). Each parsed expression, is paired with a span denoting the byte indexes that bind it.

```
Code:     !#f x << @(map f !@xs)
          |   |    |            |
Indexes: 10   14   19           31
```

AST	Span
Binop(	[10,31]
Apply(...),	[10,14]
StreamOp,	
Delay(...),	[10,14]
)	

Figure 3.1: AST and Span generation when parsing

These spans are used to provide code locations in error messages.

A correct implementation of the parser was verified using unit tests. The tests were implemented by nesting Oters code strings to iteratively generate full expressions which were then parsed and compared to the expected AST.

3.3 Type Checking

The parser converts the user's code into an Abstract Syntax Tree. However, the user Oters code and the abstract language do not correspond one-to-one. The AST first needs to be "desugared" and transformed into the calculus defined in Section 3.1, so that we can perform a static type check. For example we perform the following transformations:

- Function definitions which can take multiple arguments, such as `fn x y -> e` are transformed into $\lambda x.\lambda y.e$.
- A let expression `{ let x = e1; e2 }` is transformed into $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$.
- The stream operator `x << xs` is desugared as $\mathsf{Into}\ \langle x, xs \rangle$.

After this transformation, type checking is done on the result. Type-checking is done in Oters linearly down the file. This means that top-level definitions must come in order of dependencies. For example if function `g` calls function `f`, then `f` must be defined before `g` in the file. The type checker thus runs through each top-level expression linearly and adds definitions into a map from names to their type. The linear progression of the type checker extends to file dependencies as shown in Figure 3.2. We define next the top-level expressions in Oters.

First, there is the `use` expression which brings functions and types into scope from other files or the standard library. Moreover, the standard library has a tree structure with two root modules `std` and `gui`, and further children modules such as `std::stream` and `gui::widget`. When type checking a file, a node is added to a Directed Acyclic Graph (DAG), where each node holds the file's definition maps. These definition DAGs are illustrated in Figure 3.3. If a programmer writes `use std::stream::map`, then the type checker first traverses the DAG from the root `std` to its child `stream`. Next, the `map` entry is retrieved from the value definitions map corresponding to this file.

The next set of top-level expressions are type aliases, struct and enum definitions. All of these insert an entry in the type definitions map of the current file being type checked. The parsed AST is converted to a type representation. For a polymorphic type such

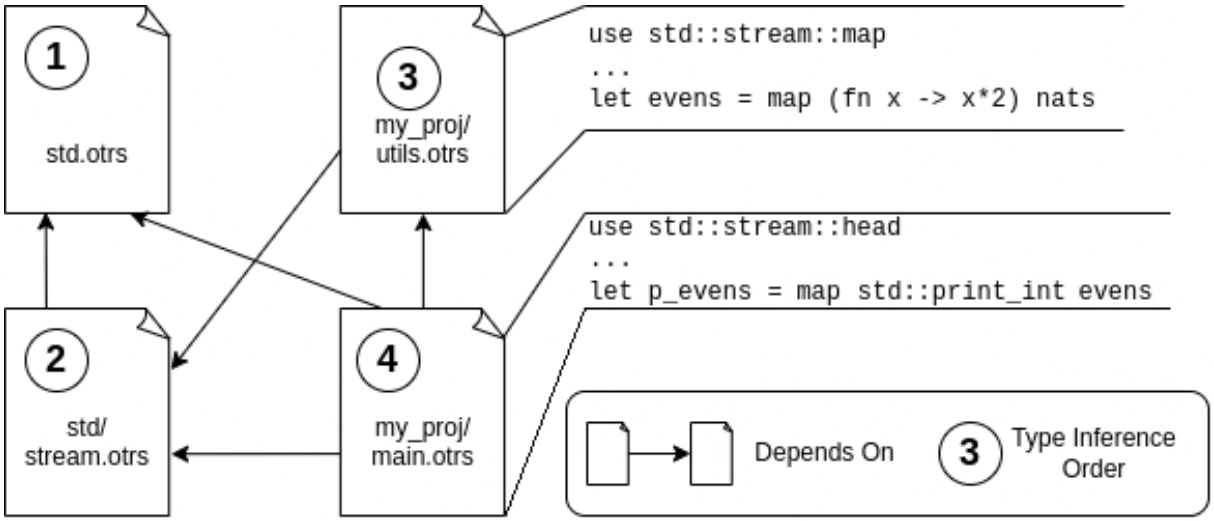


Figure 3.2: Order and Dependencies for Type Checking Files

as `enum Option<A>`, this includes introducing a type scheme. For a guarded recursive type, we also insert Fix ϕ at the front and substitute any recursive uses behind a $\bigcirc$ with the fix point variable ϕ. So the `type Stream<A> = (A, @Stream<A>)` is converted to $\forall A.\mathsf{Fix}\ \phi.A \times \phi$. The final type is then run through a well-formedness check to ensure that it does not contain any unbounded type variables.

Finally, consider the top-level `let` expressions which define global variables. The type checker first desugars the expression, and transforms any recursive definitions into guarded fix point expressions. This requires substituting recursive uses of the global variable with a fresh recursion variable Adv (Unbox r). This recursion variable is added to the expression's variable context with which the expression's type will be inferred. The type returned by the inference is then analyzed for any free type variables. If any are found, we convert the type into a scheme quantified over all the free variables. This final type is then inserted to the current file's value definitions map.

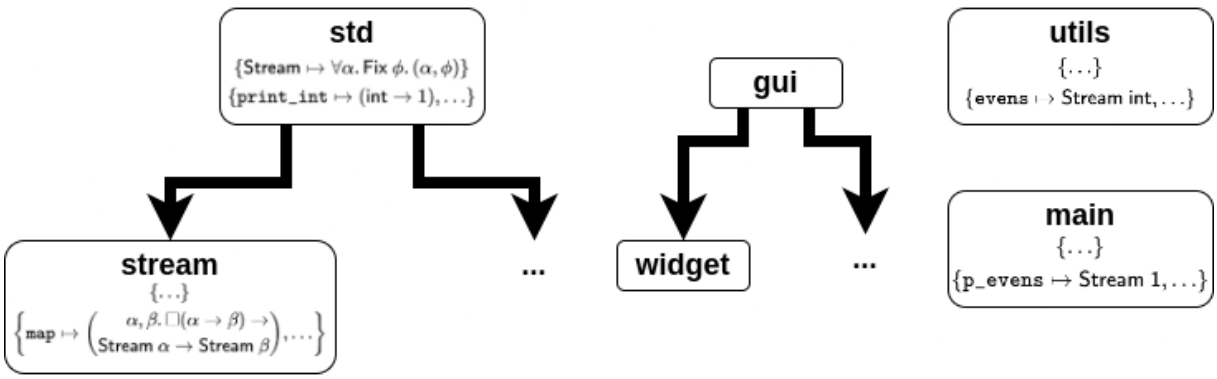


Figure 3.3: Module Definitions DAGs

In addition to the standard `let` binding, I also provide a top-level expression for defining mutually recursive streams. A different expression is needed as the linear type checking strategy disallows mutually recursive streams as no cycles are allowed between value dependencies. Mutually recursive streams pose a chicken and egg dilemma. To

solve this, an order of evaluation for the two mutually recursive streams must be chosen, and an initial value must be given for the first stream. These requirements are captured in the expression `let x = e1 and y = e2 with e3` which has the following type rule:

$$\frac{\Gamma \vdash e_3 : T_1 \qquad \Gamma, x : \mathsf{Stream}\ T_1 \vdash e_2 : \mathsf{Stream}\ T_2 \qquad \Gamma, y : \mathsf{Stream}\ T_2 \vdash e_1 : \mathsf{Stream}\ T_1}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{and}\ y = e_2\ \mathsf{with}\ e_3 : 1}$$

This rule's three premises appear in the order in which they are checked. First, e_3 (the initial value for the stream e_2) has its type inferred. This type is then used to include the variable $x : \mathsf{Stream}\ T_1$ in the variable context of e_2 which has its type subsequently inferred. The resulting type is then bound to $y : \mathsf{Stream}\ T_2$ when inferring e_1, whose type should be a stream of e_3's type.

3.3.1 Type Inference

Motivation

My initial plan assumed only static type *checking* would be necessary. However, simple type checking turned out to be at adds with my language design goals. The original intent was to restrict explicit types to solely function arguments. For example, the `map` function would look something like `let map = fn f: #(A -> B), xs: (Stream A) => ...`. The return type would then be statically found by checking the type of the function's return expression.

The issue with this approach lies with type checking recursive functions. Let f be a recursive function for which we know its argument's type, say A, but not its return type, giving $\Gamma \vdash f : A \to ?$. Often, the return value for a recursive function is produced from a recursive call. So given some value x of type A, the return value's expression might be typed as:

$$\frac{\Gamma \vdash f : A \to ? \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : ?} \to \mathrm{E}$$

This gives us no additional information on f's return type, and so we must infer it from elsewhere in the typing proof tree. Thus, to keep return types implicit I would have to employ type *inference*.

The Inference Algorithm

The type inference algorithm follows Hindley and Milner's Algorithm J [19]. I further extended the algorithm to include a notion of a stable variable trait. For example, the function that creates a constant stream: `let const = fn x -> x << @(const x)`, requires the variable `x` to be stable. This allows us to retain polymorphism while restricting argument types to be stable when needed.

Algorithm J works by taking an expression e and a variable context Γ and recursively applying the appropriate type rule (Appendix A.2). We can therefore formalize it as a function $\mathcal{J}(\Gamma, e)$. A type is inferred from a particular expression case-by-case depending on the typing rule [29]. This algorithm is partly defined below:

Variable:

$$\frac{A \text{ stable} \vee \Gamma' \text{ tick-free}}{\Gamma, x : A, \Gamma' \vdash x : A}\text{Hyp}$$

The result of the algorithm when it encounters a variable will depend on the latter's type. First, if the variable's type is stable, then we can immediately return it:

$$\mathcal{J}((\Gamma, x : A, \Gamma'), x) = A \quad \text{if } A \text{ Stable}$$

Otherwise, if Γ' tick-free, then we must instantiate any type schemes bound to x:

$$\mathcal{J}((\Gamma, x : \forall\alpha.A, \Gamma'), x) = [\alpha/\beta]A \quad \text{if } \Gamma \text{ tick-free with } \beta \text{ fresh}$$

Lastly, if neither of these hold such that x is bound to an unstable type A and is located behind a ✓ in the context, then we will infer the type variables in A to have the stable trait. If this substitution results in a stable type then we return it.

$$\mathcal{J}((\Gamma, x : A, \Gamma'), x) = [\vec{\alpha}/\vec{\alpha}^{\square}]A \quad \text{if } \Gamma \text{ not tick-free and } [\vec{\alpha}/\vec{\alpha}^{\square}]A \text{ Stable}$$

Lambda Abstraction:

$$\frac{|\Gamma|, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}{\to}\text{I}$$

The type A is not derived from any of the rule's premises. To type check e, we must therefore instantiate A using a fresh type variable.

$$\mathcal{J}(\Gamma, \lambda x.e) = \alpha \to B \quad \text{where } B = \mathcal{J}((|\Gamma|, x : \alpha), e) \text{ and } \alpha \text{ is fresh}$$

Application:

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}{\to}\text{E}$$

This rule attempts to unify the type variable previously introduced for a lambda abstraction as it now matches e''s type. Moreover, we introduce a new return type variable β which we use to unify the lambda's type C:

$$\mathcal{J}(\Gamma, \lambda x.e) = \beta \quad \text{where } C = \mathcal{J}(\Gamma, e),\ A = \mathcal{J}(\Gamma, e') \text{ and unify}(C \equiv A \to \beta) \text{ with } \beta \text{ fresh}$$

Let Binding:

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A \vdash e' : B}{\Gamma \vdash \text{let } x = e \text{ in } e' : B}\text{Let}$$

We introduce type schemes with let-bindings. Here, we first infer the type of e to be A. Then, we identify any free type variables in A that aren't found in our context Γ. Those free type variables are the ones we quantify over in our type scheme.

$$\mathcal{J}(\Gamma, \text{let } x = e \text{ in } e') = B \quad \text{where } A = \mathcal{J}(\Gamma, e), \quad B = \mathcal{J}(\Gamma, x : \forall\vec{\alpha}.A, e') \text{ and } \vec{\alpha} = \text{free}(A) \text{ / free}(\Gamma)$$

Match:

$$\frac{\Gamma \vdash e : T \qquad p_i :_P T \text{ for } i \in [1,n] \qquad \Gamma, \text{bind}(p_i, T) \vdash e_i : T' \text{ for } i \in [1,n]}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\} : T'}\textsc{Match}$$

The bind function here returns the appropriate variable bindings given an expression and a pattern. Type inference then requires that e's type and all the patterns' types are unified. Moreover, the return type of all the matching expressions $e_1, \ldots, e_n$ must also be unified.

$$\begin{aligned}\mathcal{J}(\Gamma, \mathsf{match}\ e\ \mathsf{with}\ \{p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\}) = \ \beta \quad & \text{where } A = \mathcal{J}(\Gamma, e),\\ & p_1 \ :_P A_1 \text{ for } i \in [1,n]\\ & \text{unify}(\{A \equiv A_1, \ldots, A \equiv A_n\}),\\ & B_i = \mathcal{J}((\Gamma, \text{bind}(p_i, A_i)), e_1) \text{ for } i \in [1,n]\\ & \text{and unify}(\{\beta \equiv B_1, \ldots, \beta \equiv B_n\}), \text{ with } \beta \text{ fresh}\end{aligned}$$

Figure 3.4: Algorithm J for Oters

Figure 3.4 shows how some of the cases require the unification of pairs of variables. The 'unify' function takes a set of constraints as inputs and returns a substitution. The inference algorithm must therefore additionally operate using a global substitution which is applied to the variable context after each expression is inferred. Substitutions returned by unification are added to this global substitution. A partial definition of the algorithm is given here:

$$\begin{aligned}\text{unify}(\emptyset) &= []\\ \text{unify}(\{A \equiv A\} \cup C) &= \text{unify}(C)\\ \text{unify}(\{A \equiv \alpha\} \cup C) &= \text{unify}([A/\alpha]C) \circ [A/\alpha] \text{ if } \alpha^{\square} \notin \text{free}(A)\\ \text{unify}(\{A \equiv \alpha^{\square}\} \cup C) &= \text{unify}([A/\alpha]C) \circ [A/\alpha^{\square}] \text{ if } A \text{ Stable and } \alpha^{\square} \notin \text{free}(A)\\ \text{unify}(\{A \equiv \alpha^{\square}\} \cup C) &= \text{FAIL if } A \text{ not Stable}\\ \text{unify}(\{\langle A, B\rangle \equiv \langle A', B'\rangle\} \cup C) &= \text{unify}(\{A \equiv A', B \equiv B'\} \cup C)\\ \text{unify}(\{\bigcirc A \equiv \bigcirc A'\} \cup C) &= \text{unify}(\{A \equiv A'\} \cup C)\\ &\vdots\\ \text{unify}(\{A \equiv B\} \cup C) &= \text{FAIL}\end{aligned}$$

Figure 3.5: Unification Algorithm

Above, we denote $\sigma \circ \sigma'$ to mean the composition of substitutions, i.e. first applying σ' and then σ. Moreover, when unifying free type variables α we check that they do not occur in A to prevent recursive types. When unifying type variables $\alpha^{\square}$ with the stable

trait, we require that the type being unified with is also stable. The algorithm unifies composite types such as $\langle A, B \rangle$ and $\bigcirc A$ by adding constraints unifying the underlying types.

Finally, I mention the overloaded arithmetic operators for both int and float. Say we have a function $\lambda x.\lambda y.x+y$. As the $(+)$ operator is overloaded for both number types, the type inference algorithm gives this function the type $\forall\alpha.\alpha \to \alpha \to \alpha$. This unfortunately means that we could apply values of, say, type string to this function, for which the operator is undefined. I have therefore chosen the greater familiarity and convenience afforded by overloaded operators at the risk of runtime errors.

3.4 The Interpreter

The interpreter's purpose is to implement the operational semantics of the language. Expressions are evaluated over a store whose structure was defined in Subsection 2.2.5. This was implemented as two maps (for the *now* and *later* heaps) from locations to expressions. Locations are further implemented as unsigned integers. A naive strategy for allocating store locations is to simply count up from 0. However, given that a typical program has many streams constantly being updated adding new locations, there is a risk of integer overflow. Instead I implement a location allocator using the following strategy:

1. **Initialize**: Initialize the allocator with an initially empty priority queue (implemented as a binary heap) and a maximum location counter starting at 0.

2. **Allocate**: If the priority queue is empty, increase the maximum location counter and return the old value. Otherwise, pop the top value off the queue corresponding to the minimum available location and return it. This has time complexity $\mathcal{O}(1)$.

3. **Deallocate**: Push the location being deallocated onto the priority queue. A location is deallocated when stepping into a new time-step and the location is in the *now* heap. This has average time complexity $\mathcal{O}(1)$.

Since the language ensures no implicit space leaks, this strategy guarantees that the maximum location stays constant after the program reaches a stable state. Moreover, since the number of locations in the allocator's queue will stay constant after the program reaches a stable state, it has space complexity $\mathcal{O}(1)$.

Of these globals, the interpreter identifies which ones are streams and must therefore be re-evaluated each time-step. Streams are then evaluated in their order of dependency. Non-stream values are immutable and therefore do not ever need to be re-evaluated. In terms of our modal type system, global variables are considered to be stable.

Furthermore, all the variables brought into the local scope by `use` expressions are also copied in the global variables map according to the module path where the import takes place. When evaluating an expression, we keep track of its current module path so that variables from the same module can be correctly indexed when the path is not specified.

Another interpreter optimization is that after we evaluate a stream at a particular time-step, we update its location in the *now* heap with the new value. This prevents

the stream from being re-evaluated when another variable accesses it. Moreover, while no Oters expression can have any side-effects, functions imported from Rust through the FFI may. Thus, this single evaluation of a stream also prevents inconsistent streams when they have side-effects.

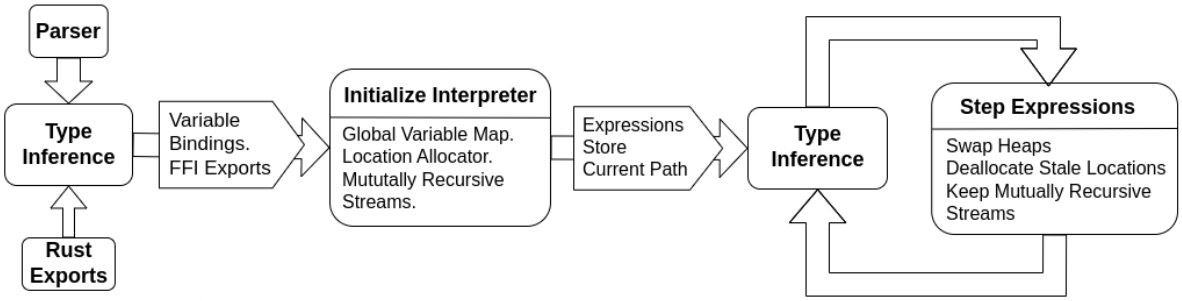


Figure 3.6: A Sketch of the Interpreter Pipeline

There is a further difficulty when evaluating pairs of mutually recursive streams. The first of each pair that is evaluated uses the other's value from the previous time-step. However, when we step forward in time we delete the *now* heap, and discard the values of all streams to avoid space leaks. If we attempt to access the second stream's value in the store, we end up in an endless loop as each stream calls for the other to be evaluated. To remedy this situation, we keep a record of all the mutually recursive streams. When we step forward in time, we retain all mutually recursive streams' values for a single time-step so that the first of each pair has a value available.

3.5 Foreign Function Interface

The core of the Oters language is now finished and we can start running programs. However, the functionality of the language is greatly limited as there is no way to provide user I/O. To remedy this and give Oters access to Rust's mature library ecosystem I develop the foreign function interface (FFI).

The aim of the FFI is to provide a seamless way of calling Rust functions from Oters. This was achieved through metaprogramming, where a procedural macro converts Rust expressions to be used by Oters. Procedural macros in Rust are functions that take as input a stream of syntax tokens of an expression and replaces it with another token stream. The Rust compiler type checks the generated code to make sure it is error free.

The library *syn* [27] is used to parse the code token stream input, returning its corresponding AST. The exporting macro analyzes this AST and determines the argument and return types of the input function. These types are then homogenized into a single `Value` type, that encodes the different types of Oters values as unique variants. Maps in Rust require all elements to be of the same type. Thus, the homogenization allowed all the exported functions to be collected into a single map from function names to function pointers of type `Vec<Value> -> Value`. In the code generation stage, the body of the function is prefixed with a statement unwrapping the arguments back to their original

Rust types. Finally, the return statement is wrapped into a `Value` type so that it matches our homogenized return type.

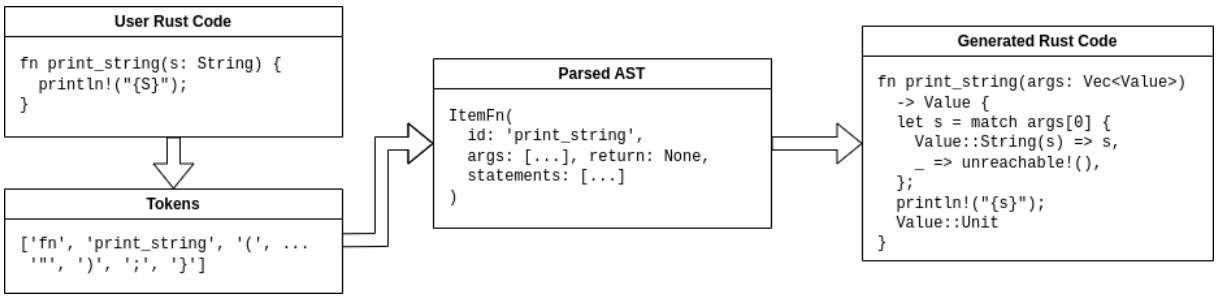


Figure 3.7: Exporting a Rust Function

We also want to be able to export structs and enums. These don't require any code to be generated, but we still need a procedural macro to analyze the types of any structs' fields and enums' variants. This analysis is performed in the same manner as for exported functions.

Finally, we expose a function to the user that compiles the exported expressions into three maps: 1. exported functions, 2. structs, 3. enums. The function map contains the function pointers generated above, allowing the interpreter to call the exported function when required. It also contains information on the function's type for the type checker. The other two maps collect the type information of the exported structs' fields, and enums' variants, also for the type checker. This information will prevent runtime type errors as the type checker will catch any wrong types passed onto the exported functions as their arguments, or as content for the structs or enums.

3.6 Adding Graphics

All graphics and user input was handled using the *macroquad* [15] Rust library. *Macroquad* provides built-in compatibility between operating systems and even WebAssembly which would allow compiling Oters to other platforms as a future extension. It is also well-tested, with good documentation, and is used in many projects over a range of domains. However, the main reason it was chosen was for how well it integrated with programming using streams.

A standard program using *macroquad* operates on a run loop that draws a frame on a window each iteration. Therefore, to draw a sustained rectangle, we must call the `draw_rectangle` function every iteration of the loop. This is analogous to Oters, which similarly operates on a run loop that evaluates and updates streams. Therefore, integrating *macroquad* into Oters is a near trivial matter with the FFI. In Oters, to draw the sustained rectangle we can simply create a stream that calls `draw_rectangle` every time-step.

Similarly, we can easily create streams that provide user input. For example, the mouse's position is provided as a stream of x and y coordinates. This is done by exporting the appropriate *macroquad* function from Rust and calling it in a stream in Oters:

```
let mouse_stream = (mouse_pos ()) << @mouse_stream
```

We can then map this stream to a function that draws a circle at the mouse's position.

```
let cursor = map (fn coords -> draw_circle coords) mouse_stream
```

This example showcases how powerful functional reactive programming can be, as we were able to write this program in just two lines, retaining a clean programming style.

Moreover, Oters is designed with two modes of drawing graphics that take advantage of FRPs streams. The first I term *direct mode* and works like the example above. To draw shapes, text and images we must create streams that redrawn them every time-step.

Immediate mode implements the UI widget library. This mode follows the Immediate Mode GUI design pattern where drawn objects do not persist across time, rather they are all redrawn each frame. This gives us a functional view of GUIs since the interface's state is never stored as it is constantly being drawn [22]. Immediate mode GUIs further require a background system that calculates the layout of the widgets automatically each frame. No event handler is used as all output is immediately returned.

This pattern, however, does come with some limitations. To handle user interaction, we must know the layout of each widget before they are drawn. But also, the layout of each widget is calculated based on its size. Thus, we must keep the size of the widget elements constant or else risk suffer a mismatch between the visual position of each widget, and the logical position that the user interacts with.

While *macroquad* provides an immediate mode UI library, its design is closely coupled with a Rust API and would not fit FRP's programming style. Instead, the design of the widget library was redone such that widgets are implemented as streams.

A basic UI widget is a button. At its most basic, buttons takes as input, a size, a position and some content text. As output, they produce a boolean value indicating if it has been clicked. Since we are implementing it in an immediate mode library, the position is automatically calculated and the size must stay constant. Thus, we represent a button as a function from a stream of content strings to a stream of booleans indicating whether it has been clicked since the last frame.

Widget	FRP Type
Button	Stream string→Stream bool
Label	Stream string→Stream 1
Checkbox	Stream bool
Textbox	Stream string
Group	Stream [Widget]→Stream 1

Table 3.1: Oters Widgets

These widgets need a system to position them. Oters' immediate mode library includes *frames* onto which widgets are attached. Frames provide an anchor point used as a reference to position a UI tree. This tree is constructed with either vertical or horizontal groups which are themselves functions of streams, allowing for dynamic composition of UIs. A root widget is specified for the frame's UI tree, which we traverse each time-step, calculating the position and subsequently drawing all the widgets. The layout is then calculated based on the type of grouping (vertical or horizontal), such that each widget given in the group's input widget-list-stream is placed along the line.

3.7 The End Product

Oters is distributed as a Rust package through the Cargo package manager. To use Oters one simply adds it as a project dependency using Cargo. The Rust API only consists of a single `run` function, which takes a list of Oters files to link and run, and a window configuration which specifies properties of the GUI window, such as size and title. (Appendix B) Additionally, the user may export their own Rust code which can be done using a single macro call.

A rich standard library was written that implements a canon set of stream operators, as well as utility functions and graphical functionality. I further provided documentation for Oters and its standard library that contains everything a user may need to reference. Wherever Oters' standard library is insufficient, the user may draw from Rust's mature package ecosystem and export any functions needed.

3.8 Repository Overview

The repository structure segregates the main language and the graphical functionality into separate modules. This separation allowed for the language interpreter's components to be tested in isolation from any external components. Then, the FFI and the graphical capabilities were added alongside integration tests, such that any future bugs could be more easily confined to each module.

Furthermore, Cargo partly dictated the repository layout. Cargo requires that macro definitions be given as separate libraries, hence why `oters_macro/` is a top-level folder. Moreover, Cargo's test command gives clearest output when tests are inside the modules they test. This meant that no dedicated testing folders were created.

Folder	Subfolder	Description
`oters_lang/`		Oters language definitions and interpreter
	`src/exports`	Convert exported Rust to Oters
	`src/exprs`	Expressions definitions and testing
	`src/interpret`	Interpreter implementation and testing
	`src/parser`	Parse Oters to AST and testing
	`src/std`	Standard library for Oters
	`src/types`	Type definitions, checking and testing
	`src/lib.rs`	Module specification and Rust user API
	`src/oters.lalrpop`	Oters parsing grammar
`oters_macro/`		Macro definitions for FFI
`oters_gui/`		Graphical functionality for Oters
	`src/gui`	Oters gui library code
	`*.rs`	Graphical function bindings into Oters
`examples/`		Example programs in Oters

Table 3.2: Repository Overview

Chapter 4

Evaluation

In this chapter, I present the qualitative and quantitative evaluations performed on the Oters language. The evaluation was done with the core focus on answering whether Oters is a viable alternative for GUI programming, and what limitations would impede its mainstream adoption. Three criteria were used to evaluate the Oters language:

1. **Correctness**: We test the interpreter to see if it correctly rejects programs that do not comply with causality, productivity, nor allow implicit space leaks. We further confirm that there are indeed no implicit space leaks, through memory profiling.

2. **Expressiveness**: We develop develop two complex sample programs in Oters to gauge its viability for GUI programming. We subsequently judge the programming experience and the performance of the applications to identify deficiencies in the language's design and implementation.

3. **Usability**: We conduct a user study asking participants to create a small GUI application using Oters. We then analyze participants' performance to asses the effectiveness of Oters for GUI programming.

4.1 Correctness

4.1.1 Type System Safety

We first show how the type checker indeed ensures causality and productivity by testing potentially problematic programs.

In Section 2.1,the `tail` function, defined as `let tail = fn (x << xs) -> xs`, violated causality. In Oters, such a function is valid, but instead of violating causality, `tail` returns a value of type $\bigcirc(\mathsf{Stream}\ A)$. We could try to violate causality and change `tail` such that it unwraps the $\bigcirc$ modality like so:

```
let tail = fn (x << xs) -> !@xs
```

However, this correctly raises the following type error:

```
Adv expressions must always be inside Delay expressions on line 1
let tail = fn (x << xs) -> !@xs
           ^^^^^^^^^^^^^^^^^^^^
```

Next, we can try to violate productivity with our earlier naive example:

```
let loop = fn x -> loop x
let blocking = loop () << blocking
```

Here, our implementation of guarded recursive types catches the error. We recall that recursive functions have a fixed point implicitly inserted such that `loop` gets transformed into fix $r.\lambda x.\mathsf{Adv}(\mathsf{Unbox}\ r)\ x$. Thus, we get a similar error as above, although in this example it does not describe the actual cause:

```
Adv expressions must always be inside Delay expressions on line 1
let loop = fn x -> loop x
           ^^^^^^^^^^^^^^
```

The last property our type checker must ensure is a lack of implicit space leaks. Passing the following program to the Oters type checker should also raise an error since we are trying to pass the full, non-stable stream, into the next time-step:

```
let leaky = fn xs -> head xs << @(leaky xs)
```

And indeed, we get the following error:

```
The variable xs, cannot be accessed in the current context on line 2
let leaky = fn xs -> head xs << @(leaky xs)
                                        ^^
```

A more subtle attack for implicit space leaks would be to write the following function:

```
let leaky_map = fn f -> {
  let aux = fn (x << xs) -> f x << @(leaky_map f !@xs);
  aux
}
```

However, this function also fails as the fix point operator that is implicitly introduced restricts the variable context to only stable types. Since `f` is not stable, Oters correctly returns this error:

```
The variable f, cannot be accessed in the current context on line 2
let aux = fn (x << xs) -> f x << @(leaky_map f !@xs);
                                  ^
```

4.1.2 Memory Safety

We have shown how the type checker correctly identifies and invalidates programs that do not hold up to our three safety properties. However, to truly ensure a lack of implicit space leaks, the interpreter must also be implemented correctly since Oters only fully ensures a lack of implicit space leaks through its operational semantics.

To investigate any potential memory leaksthe memory profiling tool *bytehound* [11] was used, which attaches itself to a program binary, and logs memory allocation information as the program runs. It is important to note that *bytehound* introduces a significant performance overhead. For instance, the same program on the same machine ran at an

average of 16 frames per second without *bytehound*, but at only 2 frames per second with the profiler activated. Nevertheless, this probe effect does not influence the data collected, as it only impacts CPU performance and not memory usage.

Firstly, we used *bytehound* to analyze and Oters program containing a single constant stream of strings. The string being streamed was purposefully long (8KiB) to amplify any memory leak effects.

```
let long_text = "..."
let const_text = long_text << @const_text
```

Bytehound further provides a scripting language for analyzing the collected data and creating plots. We use it to single out the memory allocations done by the interpreter. This is done by grouping allocations by their stack trace and then filtering them to select only those that pass through one of the interpreter's methods.

The resulting graph clearly shows constant memory usage (∼400KB) by the interpreter after initialization. This confirms that the interpreter only requires a constant amount of memory for a stream and that it does not introduce any space leaks in this program.

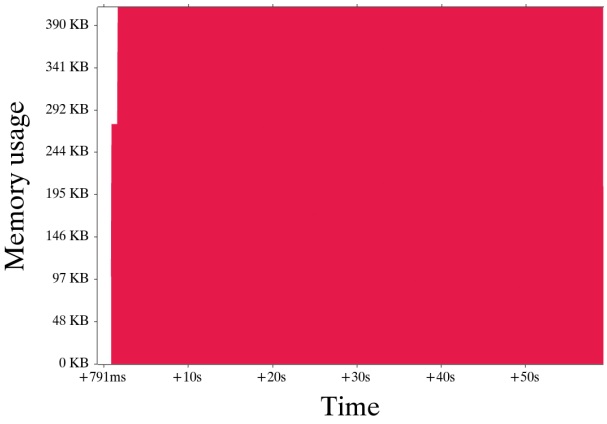


Figure 4.1: Memory Usage for a Single-Stream Program

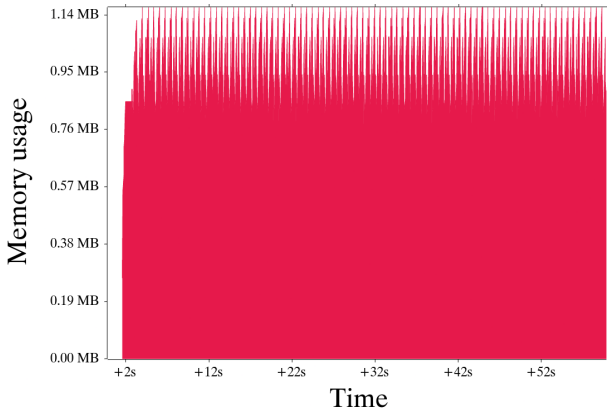


Figure 4.2: Memory Usage for a GUI Counter Program

Next, we scale up this test, and profile a more complicated program. This program implements a counter using a GUI, which is the program that participants of the user study are asked to code.

Here, we similarly see constant memory usage throughout the duration of the program's lifespan, albeit at a greater magnitude (∼1.15MB). However, we also see a constant sequence of spikes of memory allocations. Each spike corresponds to one evaluation cycle of the interpreter, translating to a frame rate of less than 1fps. This is unsatisfactory for reactive programs where we desire a frame rate of at least 20fps. Nevertheless, this example shows that the interpreter does not introduce any space leaks in larger Oters programs.

Lastly, we compare these two results with a program that contains an explicit space leak. This program stores a history of all the values emitted by a stream. In other words, it constructs a list of copies of the text string:

```
let long_text = "..."
let const_text = long_text << @const_text
let buffer_text = fold #(fn xs x -> x:xs) [] const_text
```

In this program `buffer_text` is a stream that returns an incrementing list of copies of `long_text`. We therefore expect memory usage to grow over time.

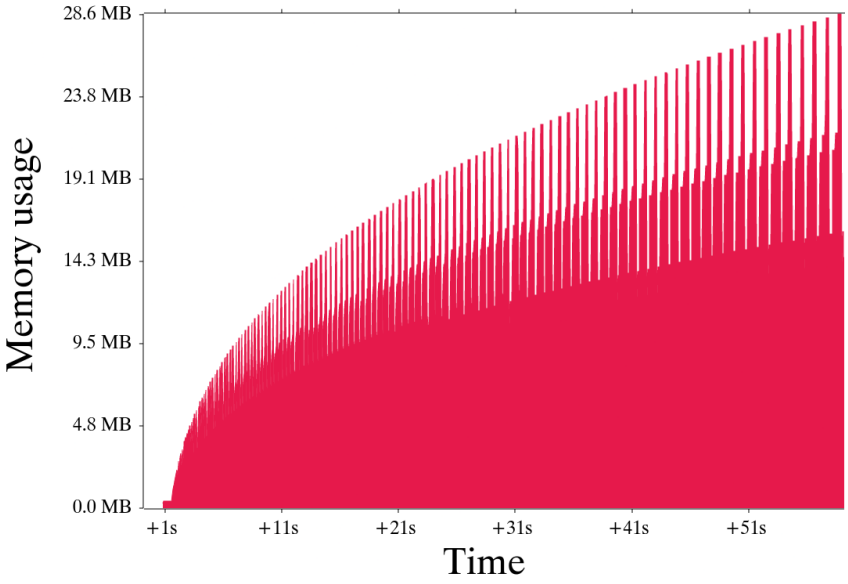


Figure 4.3: Memory Usage for an Explicitly Leaky Program

Indeed, the graph shows a clear upwards trend in memory usage over time. While the trend bears resemblance to a logarithmic function, the increase in memory usage is actually constant between evaluation cycles. However, as time passes, each evaluation cycle takes longer, and so consecutive peaks appear more spread out. Thus, the actual issue with the interpreter is not one of space leaks, but rather one of *time leaks* with the program's frame rate diminishing over time.

We have confirmed the correct implementation of the type checker and interpreter, ensuring causality and productivity, while avoiding implicit space leaks. However, we have uncovered an important flaw: the interpreter's propensity for introducing time-leaks.

4.2 Expressiveness

To evaluate Oters' expressiveness, two applications were developed: A simple paint app, and the classic snake game. Programming the applications was intuitive, and both programs were less that 100 lines of code long. The FRP style worked well, with streams being used equally for describing the application's graphics and logic.

Oters' standard library was useful, with the variety of stream operators (e.g. map, fold, zip) composing the interlaced behavior between the different components of the programs. Both graphics modes in the GUI library complemented each other well. The immediate mode graphics provided a concise style of specifying UI behavior, and the direct mode graphics allowed for more precise graphical elements. For example, in the

paint application, the tool menu was entirely composed of immediate mode graphics, but the painting itself was done in direct mode.

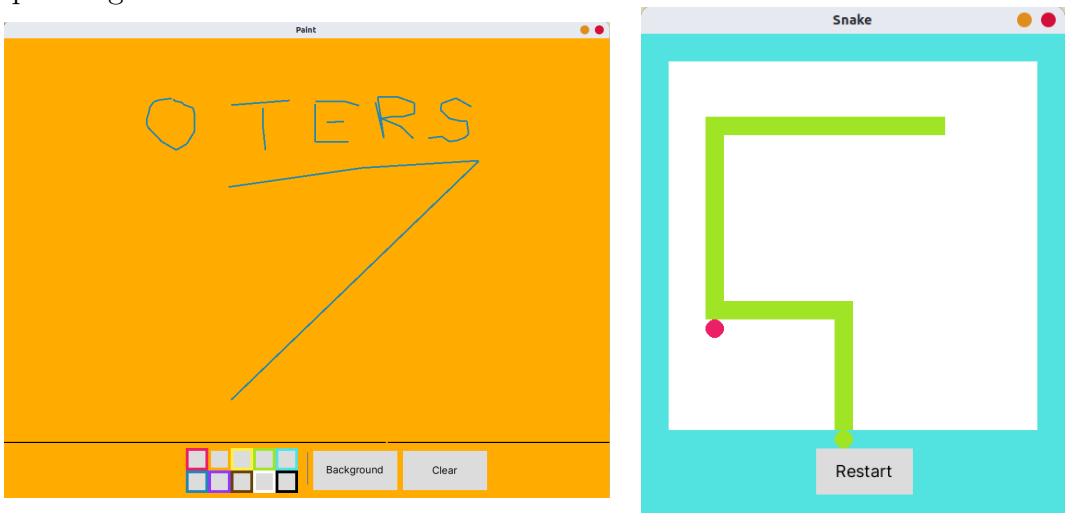


Figure 4.4: Paint and Snake Programs Written in Oters

The error messages provided also aided programming. Using the spans of the parsed tokens as described in Section 3.2, errors were mostly successful in pointing out the precise position of their source. Admittedly, familiarity with the type system and underlying implementation was sometimes needed to interpret the errors well. For example, a common error encountered was a "variable not found in the current context" error. The source of this could be any of the following: 1. The variable was not defined, 2. access to a non-stable variable within a stable context was attempted, 3. a variable went out of scope inside a Delay expression.

However, Oters is unfortunately afflicted with some limitations, namely restrictions imposed by guarded recursion and its interplay with lists. All looping over lists needs to be done in Rust since Oters does not allow unbounded recursion. This limits any list processing to one item per frame when often times we want to traverse an entire list in this time frame. The issue is that translating from Oters to Rust and back again is inefficient. As we saw in the previous section growing lists induce time leaks, and this effect is further exacerbated by this translation. This manifests in a dropping frame rate, such that the snake gets slower as it grows in length; and in the paint program as more marks are added to the canvas.

While writing the programs, I also had to deal with the strictness that the language imposes on mutually recursive streams. Using the snake game as example, we get the following dependencies:

- The position of the snake's head, depends on the state of the game (either in playing state, or in game over).
- The tail of the snake is dependent on the head's position since it must follow it.
- The game state is dependent on the position of the head (game over when it is out of bounds), **and** on the snake's tail (game over when the snake has "eaten itself").

Such complex dependency cycles are not allowed in Oters, where mutual recursion

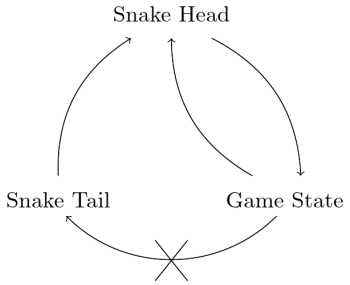


Figure 4.5: Dependency Graph for the Streams in Snake

between larger number of streams is disallowed. This meant that the game was simplified such that the game does not end if the snake "eats" itself.

4.3 Usability

4.3.1 Background

In the previous section, we verified how streams successfully capture the core of reactive behaviors when programming GUI applications. Yet despite these strengths, FRP has failed to gain mainstream popularity. We therefore want to test whether Oters' programming model is actually accessible to a broader user base. Thus, we evaluate Oters' usability through a user study.

A comprehensive evaluation of a programming language would require another dissertation. Instead, we limit the aspects of Oters to focus on by drawing on Mijailović and Milićev's research, in which they identify seven main GUI programming concerns [30]. Rather than considering all seven concerns, we focus on two:

- **Layout**: Manipulation of the position and size of GUI widgets, and other graphical elements. This aspect will assess the design of the GUI library provided by Oters.
- **Inter-Behavior**: The way in which the graphical elements and the application logic interact. This aspect will assess the suitability of Oters' FRP model for GUI programming.

4.3.2 The Study

The user study asked participants to create a simple graphical application in Oters (Appendix C). The full assignment was broken down into six tasks that guided the creation of a graphical counter. A time limit of 30 minutes was set and the full documentation of the language was provided. The structure of the tasks alternate between being Layout focused and Inter-Behavior focused. After participants completed the programming tasks or the time limit ran out, they were asked to complete a questionnaire whose questions took the form of a Likert Scale.

Participants' performance was then measured on three metrics:

1. The time it took to complete each task.

2. The number of tasks completed.

3. The number of times the application was built and run. This is meant as a proxy for the number of errors made.

The data was then collected through a screen recording.

Initially, a pilot study was conducted which was helpful in finding out what instructions were unclear. The original time limit of 20 minutes was extended to 30, after the pilot showed that more time was needed.

Special care was taken to control the programming environment for each participant. All of the user studies were conducted using the same hardware and peripheral devices, including two monitors, a mouse and a keyboard. Furthermore, all data collected was anonymized and made private.

The design of the user study was also informed using Green's Cognitive Dimensions of Notations framework [10]. Cognitive Dimensions (CDs) are described as a vocabulary used when discussing usability aspects of Human-Computer Interactive (HCI) systems. When designing HCI systems we make choices that result in trade offs between the CDs [7]. Fourteen cognitive dimensions were originally identified by Green, but in this user study, the following three were considered most relevant:

- **Error Proneness**: Do aspects of the language invite mistakes? Specifically, FRP's unfamiliarity may risk more user errors. In the user study, the number of times the program was built and how users reacted to the error messages provided gives a good metric of Oters' error proneness.

- **Hard Mental Operations**: Does programming in Oters exert the user's cognitive resources? In particular, are users able to mentally visualize how streams are used in Oters, and how they correspond with the output? This dimension was considered in the user study's design, especially when requiring the participant to compose graphical and logical elements using streams.

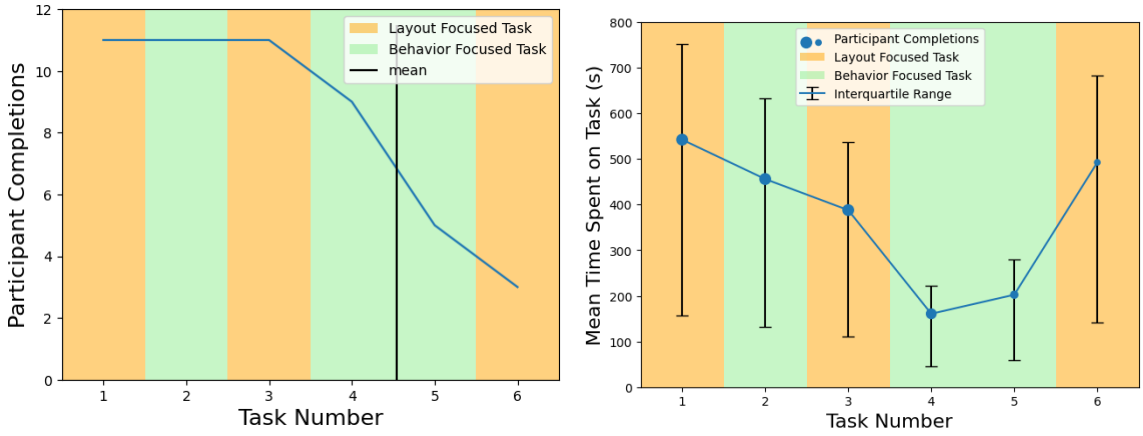
- **Hidden Dependencies**: Do invisible links exist between entities in the language's implementation? These are notably present between Oters' interpreter and the Immediate Mode graphics API. To complete the tasks, participants will need to become familiar with these hidden dependencies.

4.3.3 Results

Quantitative Analysis

A total of 11 participants performed the user study with varying degrees of success. Out of these participants, only 3 managed to complete the full set of 6 tasks. Moreover, the minimum number of tasks completed was 3 with a mean 4.54 tasks.

We also look at the time taken to complete each task shown on graph (b) above. The confidence intervals were found by assuming the time to complete each task is modeled by an exponential distribution, which models the time it takes for an event to occur. Each



(a) Number of Participants that Completed Each Task

(b) Time Taken to Complete Each Task

Figure 4.6: User Sudy

task is modeled by a different distribution. Note that a limitation of this model is the underlying assumption that the completion time between tasks is independent.

The data in this graph exhibits an initial downwards trend in the task completion times for the first 4 tasks. This can be explained by the participants becoming better acquainted with Oters. However, this trend reverses at task 4, signifying that at that point, participants have fully familiarized themselves with the basics of Oters.

Next, we notice how the behavior oriented tasks have a lower completion time than layout oriented tasks. To verify that this difference is meaningful, we perform a significance test. Specifically, we test the hypothesis that the completion times for tasks 4 and 5 follow an exponential distribution whose mean is less than that of task 6's distribution. So, let $X \sim \text{Exp}(\lambda)$ model the completion times for tasks 4 and 5, and let $Y \sim \text{Exp}(\lambda - d)$ model the completion times for task 6. Therefore, we have the null hypothesis $H_0 : d = 0$.

We now generate many synthetic datasets under the assumption of the null hypothesis. We plot the distribution of the test statistic $t = 1/\mathbb{E}(X) - 1/\mathbb{E}(Y)$ applied to these generated sets, and mark the observed value of t. From this, we calculate the p value: the probability that the generated t values are less than the observed t value. This corresponds to the probability that $\mathbb{E}(Y) > \mathbb{E}(X)$, our one-sided test. Our results find that $p = 0.042$, giving us a very high likelihood that the completion times for task 6 are on average greater than those for tasks 4 and 5.

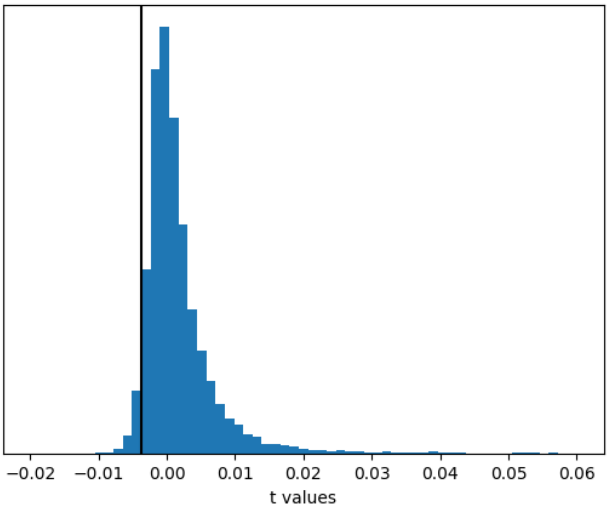


Figure 4.7: Generated and Observed t Values

Lastly, we look at the number of times each participant built and ran the application

for each task.

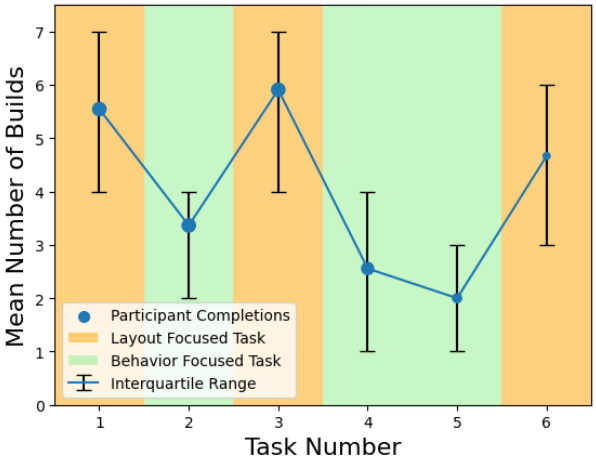


Figure 4.8: Number of Application Builds Executed for Each Task

For this graph, we found the confidence intervals by assuming the number of builds is modeled by a Poisson distribution. This distribution expresses the probability for an event to occur (e.g. build the application) a given number of times under a time interval. Once again, a limitation of this model is the assumption that the number of builds between tasks is independent.

As before, we see how the number of builds executed is lower for the tasks focused on implementing behaviors, than for the tasks focused on laying out graphical elements. We perform significance testing again, this time modeling the number of builds executed for behavior-oriented tasks with $X \sim \text{Po}(\mu)$, and for layout-oriented tasks with $Y \sim \text{Po}(\mu+d)$.

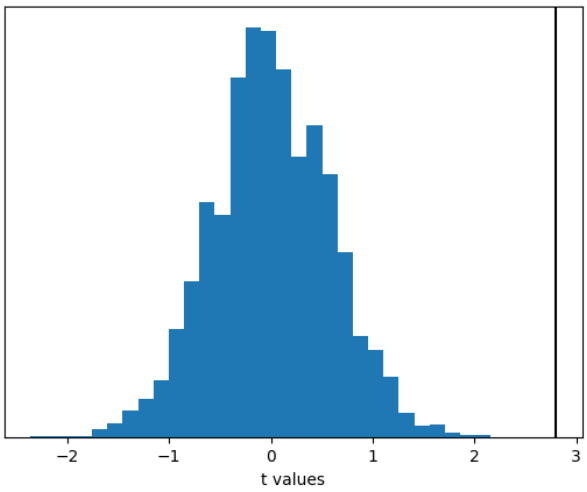


Figure 4.9: Generated and Observed t Values

Using these distributions, a null hypothesis $H_0 : d = 0$, and test statistic $t = \mathbb{E}(X) - \mathbb{E}(Y)$, we get the results above. With this data we get that $p = 0.00$ for the probability that the generated t values are greater than the observed t value. In other words, it is virtually impossible for the number of builds in the behavior-focused tasks to be greater than the number of builds in the layout-focused tasks.

We can conclude that Oters does a better job of capturing time-dependent behaviors as streams, than providing a good API for creating GUIs. It seems that while the FRP paradigm is very powerful for describing complex systems of behaviors, Oters' graphical API fails to provide an adequate means of programming GUIs.

In terms of the cognitive dimensions, on the one hand, the graphics API is error-prone. Perhaps this is due to the hidden dependencies between the API implementation, and the actual results on the screen. On the other hand, streams provide a good abstraction and mental model for users, such that they simplify the hard mental operations involved in combining complex time-dependent behaviors.

Qualitative Analysis

The Likert Scale questionnaire that participants completed asked them to rate certain prompts from a score of 1 (strongly disagree) to 5 (strongly agree). The responses from this questionnaire agree with our quantitative results. The stream data type was well received, with participants rating it with an average score of 4.1 as an "intuitive and natural fit for the tasks completed". In contrast, the graphics API was rated to not be "intuitive to use", having been given an average score of 2.5. This was likely made worse by the poor error feedback which participants found to not be "clear and useful", with an average rating of 2.8.

A further qualitative analysis of the participants' video recordings revealed more specific issues with Oters. For example, there seemed to be repeated confusion between attaching root elements and attaching widgets to a frame (Section 3.6). Moreover, the process of building a UI tree appeared unclear.

There was additional confusion among participants between immediate mode and direct mode graphics. Labels were sometimes implemented using direct mode graphics, which was not intended. Moreover, immediate mode's implementation of widgets as stream functions was not readily understood by some participants. Overall, these results show that the hidden dependencies are poorly presented to the user, especially in terms of the graphics API.

4.4 Summary

Our correctness evaluation shows how Oters successfully implements FRP in the style of Bahr's Rattus. Importantly, the type checker strictly adheres to causality and productivity, which along with the interpreter eliminates any space leaks. Unfortunately, the unoptimized interpreter suffers from time leaks.

Nevertheless, Oters has great potential to be used for developing GUI applications and games. Streams provide a concise and intuitive way of describing behaviors that can easily be scaled up. However, an over-reliance on the FFI for recursing over lists introduces further inefficiencies. Oters also fails to capture some of the complex dependencies between behaviors.

Lastly, Oters was largely well-received by users who participated in the user study. The results of this user study confirmed the suitabilty of FRP as a paradigm for programming

reactive applications. Most participants intuitively grasped programming using streams, using them to describe and compose multiple behaviors. A further shortcoming of Oters was determined, namely the inadequate design of the graphics API.

Chapter 5

Conclusions

In this dissertation, we designed and implemented Oters, an FRP language based on Bahr's modal type system. We showed how the type system was correctly implemented by adhering to the three properties: 1. causality, 2. productivity, and 3. absence of space leaks.

We further extended the language with a foreign function interface to endow it with greater usability. Moreover, we provided Oters with a graphics API, granting it functionality in its intended domain of programming GUIs.

The result is a language that proved to be robust enough to develop several complex applications, keeping to a concise and intuitive programming style. Moreover, it was found to be enjoyable by participants in a user study, who found Oters' streams abstraction a natural approach for describing reactive behaviors. However, some functional limitations in the design of the language were identified, along with a graphics API that left room to be desired.

Overall, the project was successful having satisfied all success criteria (Appendix D). Regrettably, the planned extensions were not fulfilled as most were too ambitious or ended up falling outside the project's scope. This was also partly due to an underestimation of the time a user study would take to conduct, and the requirement of unplanned components such as type inference.

5.1 Lessons Learned

One of the first lessons learned was how small design choices can have much broader implications in later components of a project. Many important design trade-offs were made, where simplicity, functionality and ease of use had to weighed against each other. I learned to better appraise the merits and design choices in programming languages.

This project also taught me how to conduct a well thought-out user study. In particular, the prior research conducted helped focus the design of the study, ensuring that useful data was collected. The pilot study was also invaluable in refining the user study to further guarantee its success.

However, while the results of the user study proved to be insightful, with the benefit of hindsight, I would likely restructure the project to be more oriented at providing an

efficient implementation.

5.2 Future Work

Most of the shortcomings Oters suffers from can be addressed with further work. I give here a list of improvements that could be made to Oters' implementation:

- Oters currently disallows recursively iterating through lists in one time-step. To address this limitation, we could introduce a bounded recursive expression for iterating over lists and numbers to avoid using the FFI.
- We can eliminate the linear evaluation strategy through a file to provide greater flexibility for programmers. We could then extend this to allow larger dependency cycles that Oters cannot support presently.
- The graphics API should be simplified. Specifically, a better way of attaching and composing UI elements in the immediate mode graphics framework should be designed. More widgets could also be implemented to allow users to create more complex UIs.
- There remain a few uncaught errors in the inference and interpreter modules that result in Rust errors. These give no information to the user about what went wrong. Moreover, greater precision in error location is also needed as some of the current messages either highlight an entire function's worth of code, or point to an incorrect expression as the source.

Ultimately, with further refinement, Oters has the potential to become a viable alternative for programming GUI applications. It provides an intuitive FRP interface for programmers which very naturally implements reactive applications.

Bibliography

[1] The Cargo Book. `https://doc.rust-lang.org/cargo/`, 2014.

[2] Stack Overflow Developer Survey 2022 — survey.stackoverflow.co. `https://survey.stackoverflow.co/2022/`, 2022.

[3] Patrick Bahr. Modal frp for all: Functional reactive programming without space leaks in haskell. *Journal of Functional Programming*, 32:e15, 2022.

[4] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), aug 2013.

[6] Stephen Blackheath and Anthony Jones. *Functional reactive programming*. Manning Publications Co., 2016.

[7] Alan F Blackwell. *Notational systems - the Cognitive Dimensions of Notations framework*, page 103–134. Morgan Kaufmann, 2003.

[8] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[9] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, page 263–273, New York, NY, USA, 1997. Association for Computing Machinery.

[10] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[11] koute. bytehound: A memory profiler for Linux. — github.com. `https://github.com/koute/bytehound`, 2019.

[12] Neelakantan R. Krishnaswami. Higher-order reactive programming without space-time leaks. In *International Conference on Functional Programming (ICFP)*, September 2013.

[13] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, pages 257–266, Washington, DC, USA, 2011. IEEE Computer Society.

[14] lalrpop. lalrpop: LR(1) parser generator for Rust — github.com. `https://github.com/lalrpop/lalrpop`, 2015.

[15] Fedor Logachev. macroquad: Cross-platform game engine in Rust. — github.com. `https://github.com/not-fl3/macroquad`, 2020.

[16] Aleksi Lukkarinen, Lauri Malmi, and Lassi Haaranen. Event-driven programming in programming education: A mapping review. *ACM Trans. Comput. Educ.*, 21(1), mar 2021.

[17] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–11, 2011.

[18] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1–20, New York, NY, USA, 2009. Association for Computing Machinery.

[19] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[20] H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

[21] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 51–64, New York, NY, USA, 2002. Association for Computing Machinery.

[22] Quinn Radich and Michael Satran. Retained mode versus immediate mode - win32 apps, Aug 2019.

[23] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. Onward! 2014, page 55–68, New York, NY, USA, 2014. Association for Computing Machinery.

[24] Kensuke Sawada and Takuo Watanabe. Emfrp: A functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, page 36–44, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. *SIGPLAN Not.*, 44(9):23–34, aug 2009.

[26] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. Manipulating gui structures declaratively. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, page 63–69, New York, NY, USA, 2020. Association for Computing Machinery.

[27] David Tolnay. syn: Parser for Rust source code — github.com. `https://github.com/dtolnay/syn`, 2016.

[28] xd009642. tarpaulin: A code coverage tool for Rust projects — github.com. `https://github.com/xd009642/tarpaulin`, 2017.

[29] Jeremy Yallop. Type inference. `https://www.cl.cam.ac.uk/teaching/1718/L28/02-type-inference-notes.pdf`, 2018.

[30] Žarko Mijailović and Dragan Milićev. Empirical analysis of gui programming concerns. *International Journal of Human-Computer Studies*, 72(10):757–771, 2014.

Appendix A

Oters Full Type System

In this appendix, I describe in detail Oters' type system.

A.1 Programs and Definitions

Oters' programs can be considered as a sequence of type and variable definitions. Section 3.3 introduces the five types of definitions as top-level expressions. These definitions transform the 'Alias' and 'Global' contexts, which are represented as maps from type aliases and variable names respectively to type schemes. They are formally defined below:

$$
\begin{array}{rrl}
\text{Type Scheme} & & \forall \bar{\alpha}.T \\
\text{Definitions} & D ::= & \mathsf{type}\ \tau = t \mid \\
& & \mathsf{struct}\ \tau\ \{k_1 : t_1, ..., k_n : t_n\} \mid \\
& & \mathsf{enum}\ \tau\ \{c_1(t_1), ..., c_1(t_n)\} \mid \\
& & \mathsf{let}\ \upsilon = e \mid \\
& & \mathsf{let}\ \upsilon_1 = e_1\ \mathsf{and}\ \upsilon_2 = e_2\ \mathsf{with}\ e_3 \\
\text{Program} & P ::= & \cdot \mid D, P \\
\text{Alias Context} & \Delta ::= & \cdot \mid \Delta, \tau \mapsto \forall \bar{\alpha}.T \\
\text{Global Context} & \Upsilon ::= & \cdot \mid \Upsilon, \upsilon \mapsto \forall \bar{\alpha}.T
\end{array}
$$

Types are defined over type expressions t, which we are used above and defined here:

$$
\begin{array}{rl}
\text{Type Expressions} & t ::= 1 \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{string} \mid \mathsf{bool} \mid [t] \mid \langle t_1, \ldots, t_n \rangle \mid t_1 \to t_2 \mid \bigcirc t \mid \\
& \Box t \mid \tau \mid \tau\ (t_1\ \cdots\ t_m) \mid \phi
\end{array}
$$

These largely mirror the types defined in Section 3.1. However, note the absence of guarded fixed point types $\mathsf{Fix}\ \phi.T$ which are inferred and thus automatically introduce recursive type variables ϕ. We also have here the inclusion of an application type expression $\tau\ \bar{t}$ used to instantiate type schemes.

Additionally, we have "local" term variable contexts:

$$
\text{Term Variable Context} \qquad \Gamma ::= \ \cdot \ \mid \Gamma, x : A \mid \Gamma, \checkmark
$$

These definitions are then used in the following relations:

Relation	Description
$\langle \Delta; \Upsilon; P \rangle \rightsquigarrow \langle \Delta'; \Upsilon'; P' \rangle$	Type check programs making sure each type definition is well-formed, and each variable definition is well-typed.
$\Delta \vdash t \Rightarrow T$	Form a type T from the type expression t
$\Delta; \Upsilon; \Gamma \vdash e : T$	The "standard" typing judgment. Give expression e the type T, under term variable context Γ and global contexts Δ, Υ.

Table A.1: Relations Defined in Oters' Type System

We define these relations inductively, beginning with $\langle \Delta; \Upsilon; P \rangle \rightsquigarrow \langle \Delta'; \Upsilon'; P' \rangle$ which we read as saying *Evaluate program P, under alias context Δ and global context Υ, to produce program P' with extended definitions Δ', Υ'.*

$$\frac{[\phi/\bigcirc\tau]t \neq t \qquad \Delta \vdash [\phi/\bigcirc\tau]t \Rightarrow T \qquad \bar{\alpha} = free(T)}{\langle \Delta; \Upsilon; \mathsf{type}\ \tau = t, P \rangle \rightsquigarrow \langle \Delta, \tau \mapsto \forall\bar{\alpha}.\mathsf{Fix}\ \phi.T; \Upsilon; P \rangle}\textsc{RecAliasDef}$$

$$\frac{[\phi/\bigcirc\tau]t = t \qquad \Delta \vdash t \Rightarrow T \qquad \bar{\alpha} = free(T)}{\langle \Delta; \Upsilon; \mathsf{type}\ \tau = t, P \rangle \rightsquigarrow \langle \Delta, \tau \mapsto \forall\bar{\alpha}.T; \Upsilon; P \rangle}\textsc{AliasDef}$$

$$\frac{\Delta \vdash t_1 : T_1 \text{ for } i \in [1,n] \qquad \bar{\alpha} = \bigcup free(T_i)}{\begin{array}{l}\langle \Delta; \Upsilon; \mathsf{struct}\ \tau\ \{k_1 : t_1, \ldots, k_n : t_n\}, P \rangle \rightsquigarrow \\ \quad \langle \Delta, \tau \mapsto \forall\bar{\alpha}.\{k_1 : T_1 \times \cdots \times k_n : T_n\}; \Upsilon; P \rangle\end{array}}\textsc{StructDef}$$

$$\frac{\Delta \vdash t_1 : T_1 \text{ for } i \in [1,n] \qquad \bar{\alpha} = \bigcup free(|_i)}{\begin{array}{l}\langle \Delta; \Upsilon; \mathsf{enum}\ \tau\ \{c_1(t_1), \ldots, c_n(t_n)\}, P \rangle \rightsquigarrow \\ \quad \langle \Delta, \tau \mapsto \forall\bar{\alpha}.\{c_1(T_1) + \cdots + c_n(T_n)\}; \Upsilon; P \rangle\end{array}}\textsc{EnumDef}$$

$$\frac{\Delta; \Upsilon; \cdot \vdash e : T \qquad \bar{\alpha} = free(T)}{\langle \Delta, \Upsilon; \mathsf{let}\ v = e, P \rangle \rightsquigarrow \langle \Delta; \Upsilon, v \mapsto \forall\bar{\alpha}.T; P \rangle}\textsc{LetDef}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_3 : T_2 \\ \Gamma, v_2 : \mathsf{Stream}\ T_2 \vdash e_1 : \mathsf{Stream}\ T_1 \\ \Gamma, v_1 : \mathsf{Stream}\ T_1 \vdash e_2 : \mathsf{Stream}\ T_2 \\ \bar{\alpha}_1 = free(T_1) \qquad \bar{\alpha}_2 = free(T_2)\end{array}}{\begin{array}{l}\langle \Delta, \Upsilon; \mathsf{let}\ v_1 = e_1\ \mathsf{and}\ v_2 = e_2\ \mathsf{with}\ e_3, P \rangle \rightsquigarrow \\ \quad \langle \Delta; \Upsilon, v_1 \mapsto \forall\bar{\alpha}_1.\mathsf{Stream}\ T_1, v_2 \mapsto \forall\bar{\alpha}_2.\mathsf{Stream}\ T_2; P \rangle\end{array}}\textsc{LetAndDef}$$

Next, we define $\Delta \vdash t \Rightarrow T$ read as *Form a type T from a type expression t under alias context Δ.*

$$\frac{}{\Delta \vdash 1 \Rightarrow 1}\textsc{Form1} \qquad \frac{}{\Delta \vdash \mathsf{int} \Rightarrow \mathsf{int}}\textsc{FormInt} \qquad \frac{}{\Delta \vdash \mathsf{float} \Rightarrow \mathsf{float}}\textsc{FormFloat}$$

$$\frac{}{\Delta \vdash \mathsf{string} \Rightarrow \mathsf{string}}\textsc{FormString} \qquad \frac{}{\Delta \vdash \mathsf{bool} \Rightarrow \mathsf{bool}}\textsc{FormBool}$$

$$\frac{\Delta \vdash t \Rightarrow T}{\Delta \vdash [t] \Rightarrow [T]}\textsc{FormList} \qquad \frac{\Delta \vdash t_i \Rightarrow T_i \text{ for } i \in [1,n]}{\Delta \vdash \langle t_1, \ldots, t_n \rangle \Rightarrow T_1 \times \cdots \times T_n}\textsc{Form}\times$$

$$\frac{\Delta \vdash t_1 \Rightarrow T_1 \qquad \Delta \vdash t_2 \Rightarrow T_2}{\Delta \vdash t_1 \to t_2 \Rightarrow T_1 \to T_2}\textsc{Form}\to \qquad \frac{\Delta \vdash t \Rightarrow T}{\Delta \vdash \bigcirc t \Rightarrow \bigcirc T}\textsc{FormDelay}$$

$$\frac{\Delta \vdash t \Rightarrow T}{\Delta \vdash \Box t \Rightarrow \Box T}\textsc{FormBox} \qquad \frac{\Delta(\tau) = T}{\Delta \vdash \tau \Rightarrow T}\textsc{FormAliasType}$$

$$\frac{\Delta(\tau) = \forall \bar{\alpha}.T}{\Delta \vdash \tau \Rightarrow T}\textsc{FormAliasScheme}$$

$$\frac{\Delta(\tau) = \forall \alpha_1, \ldots, \alpha_n.T \qquad \Delta \vdash t_i \Rightarrow T_i \text{ for } i \in [1,m] \qquad m \leq n}{\Delta \vdash \tau\ (t_1, \cdots, t_m) \Rightarrow [T_i/\alpha_i]T}\textsc{FormApp}$$

$$\frac{}{\Delta \vdash \phi \Rightarrow \phi}\textsc{FormRecVar}$$

Note that in the rule FormApp, the number of type expression arguments $(t_1\ \cdots\ t_m)$ and the number of bound variables in τ's scheme $\alpha_1, \ldots, \alpha_n$ do not necessarily need to match. The leftover type variables are left unbound and added to the top-level type's scheme as described in the $\langle \Delta; \Upsilon; P \rangle \rightsquigarrow \langle \Delta'; \Upsilon'; P' \rangle$ above.

A.2 The Typing Rules

Throughout the dissertation, I have introduced many of the typing rules used for the expressions in Oters' abstract language. In the previous section, I expanded the typing judgment $\Delta; \Upsilon; \Gamma \vdash e : T$ to include the global contexts Δ and Υ. Here I give the full set of typing rules, however, instead of explicitly including the global contexts I leave them as implicit for better legibility. So we define $\Gamma \vdash e : T$ as saying, *the expression e has the type T under the term variable context Γ.*

$$\frac{T \text{ stable} \vee \Gamma' \text{ tick-free}}{\Gamma, x : T, \Gamma' \vdash x : T}\textsc{Hyp} \qquad \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Upsilon(x) = T}{\Gamma \vdash x : T}\textsc{Global}$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1}1\textsc{I} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}}\textsc{IntI} \qquad \frac{}{\Gamma \vdash f : \mathsf{float}}\textsc{FloatI} \qquad \frac{}{\Gamma \vdash b : \mathsf{bool}}\textsc{BoolI}$$

$$\frac{}{\Gamma \vdash s : \mathsf{string}}\textsc{StringI}$$

$$\frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash e + e' : \mathsf{int}}+\textsc{Int} \qquad \frac{\Gamma \vdash e : \mathsf{float} \qquad \Gamma \vdash e' : \mathsf{float}}{\Gamma \vdash e + e' : \mathsf{float}}+\textsc{Float}$$

$$\frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash e = e' : \mathsf{bool}}=\textsc{Int} \qquad \frac{\Gamma \vdash e : \mathsf{float} \qquad \Gamma \vdash e' : \mathsf{float}}{\Gamma \vdash e = e' : \mathsf{bool}}=\textsc{Float}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e' : \mathsf{bool}}{\Gamma \vdash e \wedge e' : \mathsf{bool}}\textsc{And}$$

Note that we also include similar rules to +Int and +Float for the other arithmetic

operators $(-, \times, \div, \text{mod})$. Likewise, for rules $=$Int and $=$Float with the comparison operators $(<, >)$ and rule And with the $(\vee)$ operator.

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash -e : \mathsf{int}}\textsc{NegInt} \qquad \frac{\Gamma \vdash e : \mathsf{float}}{\Gamma \vdash -e : \mathsf{float}}\textsc{NegFloat} \qquad \frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \neg e : \mathsf{int}}\textsc{Not}$$

$$\frac{\Gamma \vdash e_i : T_i \text{ for } i \in [1,n]}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : T_1 \times \cdots \times T_n}\textsc{Prod}$$

$$\frac{\Gamma \vdash e_i : T \text{ for } i \in [1,n]}{\Gamma \vdash [e_1, \ldots, e_n] : [T]}\textsc{List} \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash e' : [T]}{\Gamma \vdash e :: e' : [T]}\textsc{Cons}$$

$$\frac{|\Gamma|, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2}{\to}\text{I} \qquad \frac{\Gamma \vdash e : T_1 \to T_2 \qquad \Gamma \vdash e' : T_1}{\Gamma \vdash e\ e' : T_2}{\to}\text{E}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}\textsc{If} \qquad \frac{\Gamma \vdash e : 1 \qquad \Gamma \vdash e' : T}{\Gamma \vdash e; e' : T}\textsc{Seq}$$

$$\frac{\Gamma \vdash e_i : T_i \text{ for } i \in [1,n] \qquad \{k_1 : T_1, ..., k_n : T_n\} \in \text{codom}(\Delta)}{\Gamma \vdash \{k_1 : e_1, ..., k_n : e_n\} : \{k_1 : T_1, ..., k_n : T_n\}}\textsc{StructI}$$

$$\frac{\Gamma \vdash e : \{k_1 : T_1, ..., k_n : T_n\} \qquad k : T' \in \{k_1 : T_1, ..., k_n : T_n\}}{\Gamma \vdash \pi_k(e) : T'}\textsc{StructE}$$

$$\frac{\Gamma \vdash e : T' \qquad \{c_1(T_1) + ... + c_n(T_n)\} \in \text{codom}(\Delta) \qquad c(T') \in \{c_1(T_1) + ... + c_n(T_n)\}}{\Gamma \vdash c(e) : \{c_1(T_1) + ... + c_n(T_n)\}}\textsc{EnumI}$$

$$\frac{\Gamma \vdash e : T \qquad p_i :_P T \text{ for } i \in [1,n] \qquad \Gamma, \text{bind}(p_i, T) \vdash e_i : T' \text{ for } i \in [1,n]}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\} : T'}\textsc{Match}$$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma, x : T_1 \vdash e' : T_2}{\Gamma \vdash \text{let } x = e \text{ in } e' : T_2}\textsc{Let}$$

$$\frac{|\Gamma|, \checkmark \vdash e : T_1}{\Gamma \vdash \mathsf{Delay}\ e : \bigcirc T_1}\textsc{Delay} \qquad \frac{\Gamma \vdash e : \bigcirc T_1}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{Adv}\ e : T_1}\textsc{Adv}$$

$$\frac{\Gamma^{\Box} \vdash e : T_1}{\Gamma \vdash \mathsf{Box}\ e : \Box T_1}\textsc{Box} \qquad \frac{\Gamma \vdash e : \Box T}{\Gamma \vdash \mathsf{Unbox}\ e : T}\textsc{Unbox}$$

$$\frac{\Gamma^{\Box}, r : \Box(\bigcirc T) \vdash e : T}{\Gamma \vdash \mathsf{fix}\ r.\ e : T}\textsc{Fix} \qquad \frac{\Gamma \vdash e : [\bigcirc(\mathsf{Fix}\ \phi.\ T)/\phi]T}{\Gamma \vdash \mathsf{Into}\ e : \mathsf{Fix}\ \phi.\ T}\textsc{Into}$$

$$\frac{\Gamma \vdash e : \mathsf{Fix}\ \phi.\ T}{\Gamma \vdash \mathsf{out}\ e : [\bigcirc(\mathsf{Fix}\ \phi.\ T)/\phi]T}\textsc{Out}$$

A.2.1 Patterns

In addition to typing rules for expressions, we must also type patterns such that we can safely bind expressions to them in match statements. We therefore introduce the $p :_P T$ judgment read as *Pattern p matches expressions of type T*. Note that the global contexts Υ and Δ are still in scope for this typing judgment and passed implicitly.

$$\frac{}{\_ :_P T}\textsc{BlankPat} \qquad \frac{}{x :_P T}\textsc{VarPat} \qquad \frac{p_i :_P T_i \text{ for } i \in [1,n]}{\langle p_i, \ldots, p_n \rangle :_P T_1 \times \cdots \times T_n}\textsc{ProdPat}$$

$$\frac{p :_P T' \qquad \{c_1(T_1) + ... + c_n(T_n)\} \in \text{codom}(\Delta) \qquad c(T') \in \{c_1(T_1) + ... + c_n(T_n)\}}{c(p) :_P \{c_1(T_1) + ... + c_n(T_n)\}}\textsc{EnumPat}$$

$$\frac{p_1 :_P T \qquad p_2 :_P T}{p_1 \vee p_2 :_P T}\textsc{OrPat}$$

We also define the bind function which takes a pattern p and a type T, and returns a set of variable bindings:

$$\begin{aligned}
\text{bind}(_, T) &= \emptyset \\
\text{bind}(x, T) &= \{x : T\} \\
\text{bind}(\langle p_1, \ldots, p_n \rangle, T_1 \times \cdots \times T_n) &= \bigcup_{i \in [1,n]} \text{bind}(p_i, T_i) \\
\text{bind}(c(p), \{c_1(T_1) + ... + c_n(T_n)\}) &= \text{bind}(p, T') \quad \text{where } c(T') \in \{c_1(T_1) + ... + c_n(T_n)\} \\
\text{bind}(p_1 \vee p_2, T) &= \text{bind}(p_1, T) \cup \text{bind}(p_1, T)
\end{aligned}$$

A.3 Operational Semantics

The type system is not complete without its operational semantics. As described in Section 2.2.5, the operation semantics is split into two categories of rules: 1. the evaluation semantics, 2. the step semantics.

A.3.1 Evaluation Semantics

The evaluation semantics describe how Oters expressions are reduced within a time-step. They are presented using the judgment $\langle e; \sigma \rangle \Downarrow \langle e'; \sigma' \rangle$ which we read as *an expression e with store σ is evaluated to e' with the store transformed to σ'*.

We further need to define values in Oters:

$$\begin{aligned}
\text{Values} \qquad v ::= {} & \langle\rangle \mid n \mid f \mid s \mid b \mid \langle v_1, \ldots, v_n \rangle \mid [v_1, \ldots, v_n] \mid \lambda x.e \mid \\
& \{k_1 : v_1, \ldots, k_n : v_n\} \mid c(v) \mid \mathsf{Box}\ v \mid \mathsf{fix}\ x.e \mid \mathsf{Into}\ e \mid l
\end{aligned}$$

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle}\textsc{ValueEv} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \qquad \langle e';\sigma'\rangle \Downarrow \langle n';\sigma''\rangle \qquad m = n + n'}{\langle e + e';\sigma\rangle \Downarrow \langle m;\sigma''\rangle}+\textsc{Ev}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \langle e';\sigma'\rangle \Downarrow \langle v';\sigma''\rangle \qquad v = v'}{\langle e = e';\sigma\rangle \Downarrow \langle \mathsf{true};\sigma''\rangle}=\textsc{TrueEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \mathsf{false};\sigma'\rangle}{\langle e \wedge e';\sigma\rangle \Downarrow \langle \mathsf{false};\sigma'\rangle}\wedge\textsc{FalseEv} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle \mathsf{true};\sigma'\rangle}{\langle \neg e;\sigma\rangle \Downarrow \langle \mathsf{false};\sigma'\rangle}\neg\textsc{Ev}$$

$$\frac{\langle e_i;\sigma_i\rangle \Downarrow \langle v_i;\sigma_{i+1}\rangle \text{ for } i \in [1,n]}{\langle\langle e_1,\ldots,e_n\rangle;\sigma_1\rangle \Downarrow \langle\langle v_1,\ldots,v_n\rangle;\sigma_{n+1}\rangle}\textsc{ProdEv}$$

$$\frac{\langle e_i;\sigma_i\rangle \Downarrow \langle v_i;\sigma_{i+1}\rangle \text{ for } i \in [1,n]}{\langle[e_1,\ldots,e_n];\sigma_1\rangle \Downarrow \langle[v_1,\ldots,v_n];\sigma_{n+1}\rangle}\textsc{ListEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma\rangle \qquad \langle e';\sigma\rangle \Downarrow \langle[v_1,\ldots,v_n];\sigma\rangle}{\langle e :: e';\sigma\rangle \Downarrow \langle[v,v_1,\ldots,v_n];\sigma\rangle}::\textsc{Ev}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \lambda x.e'';\sigma'\rangle \qquad \langle e';\sigma'\rangle \Downarrow \langle v;\sigma''\rangle \qquad \langle [v/x]e'';\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle e\ e';\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}\textsc{AppEv}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle \mathsf{true};\sigma'\rangle \qquad \langle e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}\textsc{IfTrueEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle\langle\rangle;\sigma'\rangle \qquad \langle e';\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle e;e';\sigma\rangle \Downarrow \langle v;\sigma''\rangle}\textsc{SeqEv}$$

$$\frac{\langle e_i;\sigma_i\rangle \Downarrow \langle v_i;\sigma_{i+1}\rangle \text{ for } i \in [1,n]}{\langle\{k_1 : e_1,\ldots,k_n : e_n\}\rangle;\sigma_1\rangle \Downarrow \langle\{k_1 : v_1,\ldots,k_n : v_n\};\sigma_{n+1}\rangle}\textsc{StructEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle\{k_1 : v_1,\ldots,k_n : v_n\};\sigma'\rangle \qquad k : v \in \{k_1 : v_1,\ldots,k_n : v_n\}}{\langle \pi_k(e);\sigma\rangle \Downarrow \langle v;\sigma'\rangle}\textsc{ProjStructEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle c(e);\sigma\rangle \Downarrow \langle v;\sigma'\rangle}\textsc{VariantEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \langle[\mathrm{subs}(v,p_i)]e_i;\sigma'\rangle \Downarrow \langle v_i;\sigma''\rangle \text{ if } v \text{ matches } p_i}{\langle \mathsf{match}\ e\ \mathsf{with}\ \{p_1 \Rightarrow e_1,\ldots,p_n \Rightarrow e_n\};\sigma\rangle \Downarrow \langle v_i;\sigma''\rangle}\textsc{MatchEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \langle[v/x]e';\sigma'\rangle \Downarrow \langle v';\sigma''\rangle}{\langle \mathsf{let}\ x = e\ \mathsf{in}\ e';\sigma\rangle \Downarrow \langle v';\sigma''\rangle}\textsc{LetEv}$$

$$\frac{l \notin \mathrm{dom}(\eta_L)}{\langle \mathsf{Delay}\ e;\eta_N\checkmark\eta_L\rangle \Downarrow \langle l;\eta_N\checkmark\eta_L, l : e\rangle}\textsc{DelayEv}$$

$$\frac{\langle e;\eta_N\rangle \Downarrow \langle l;\eta'_N\rangle \qquad l : e' \in \eta'_N \qquad \langle e';\eta'_N\checkmark\eta_N\rangle \Downarrow \langle v;\eta''_N\checkmark\eta'_L\rangle}{\langle \mathsf{Adv}\ e;\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\eta''_N\checkmark\eta'_L\rangle}\textsc{AdvEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \mathsf{Box}\ e';\sigma'\rangle \qquad \langle e';\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \mathsf{Unbox}\ e;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}\textsc{UnboxEv}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{Into}\ e;\sigma\rangle \Downarrow \langle \mathsf{Into}\ v;\sigma'\rangle}\textsc{IntoEv} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle \mathsf{Into}\ v;\sigma'\rangle}{\langle \mathsf{Out}\ e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}\textsc{OutEv}$$

$$\frac{\langle [\mathsf{Box}(\mathsf{Delay}(\mathsf{fix}\ r.e))/r]e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{fix}\ r.e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}\textsc{FixEv}$$

A.3.2 Step Semantics

Finally, the step semantics describe how streams in Oters are updated across time-steps. They are presented using the judgment $\langle e;\sigma\rangle \overset{v}{\Longrightarrow} \langle e';\sigma'\rangle$ which we read as *an expression e with store σ is updated after one time-step to e' with the store transformed to σ'.*

$$\frac{\langle e;\eta\checkmark\rangle \Downarrow \langle v \ll l;\eta_N\checkmark\eta_L\rangle}{\langle e;\eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{Adv}\ l;\eta_L\rangle}\textsc{StrStep}$$

Appendix B

Example Oters GUI Application

The Oters code below implements a graphical counter application. This is the same application that participants of the user study are asked to program (See Appendix C).

```
use gui::widget::*
use std::stream::const
use std::stream::map
use std::stream::zip
use std::stream::fold
use gui::shape::*

let on_space_bar =
    gui::input::is_key_down "Space" << @on_space_bar

let back_rect = draw_shape (
      Shape::Rect((180, 190),
                  (250, 210),
                  gui::Color {r: 255, g: 128, b:0, a: 255}
                 )
    ) << @back_rect

let ui = gui::frame (200, 200)

let (upbtn_id, upbtn_stream) = button ui (100, 100) (const "UP")
let (downbtn_id, downbtn_stream) =
    button ui (100, 100) (const "DOWN")

let btn_presses = zip upbtn_stream downbtn_stream
let counter_events = zip btn_presses on_space_bar
let counter_fn = fn acc ((up, down), space) ->
  if up then
    acc + 1
  else if down then
    acc - 1
  else if space then
    0
```

```
  else
    acc

 let counter = fold #counter_fn 0 counter_events

let (lab_id, lab_stream) = label ui (230, 50)
      (map #(fn i -> std::int_to_string i) counter)

let (hgrp_id, hgrp_stream) = hgroup ui (220, 150)
      (const [upbtn_id, downbtn_id]) (Alignment::Top)

let (vgrp_id, vgrp_stream) = vgroup ui (250, 250)
      (const [lab_id, hgrp_id]) (Alignment::Left)

let _ = gui::attach_root (ui, vgrp_id)
```

Listing B.1: `counter.otrs`

Oters programs are run from Rust. An example file that runs the code above is given here:

```
#[oters::export_oters]
fn print_message(s: String) {
    println!("This message is being printed from a Rust function:
        \n{s}");
}

fn main() {
    let config = oters::WindowConfig {
        title: "Counter".to_string(),
        dimensions: (800, 600),
        resizable: true,
        fullscreen: false,
        icon: None,
    };
    oters::run!(vec!["./counter.otrs".to_string()], config);
}
```

Listing B.2: `main.rs`

When run, this program outputs the following:

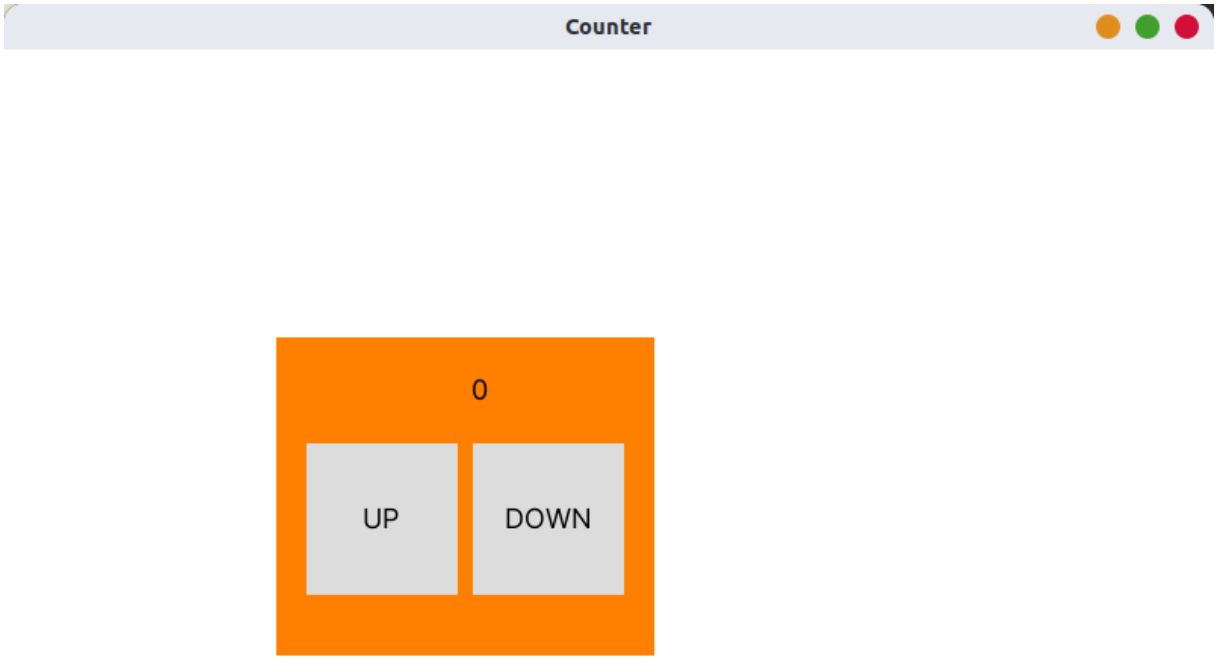


Figure B.1: GUI Counter Application Written in Oters

Appendix C

Condensed User Study Document

This is the document presented to the user study participants, condensed to take up less space, and with images removed.

C.1 Introduction

The user study you will be taking part in seeks to evaluate the effectiveness of the Oters programming language. Oters is a domain specific language intended for writing Graphical User Interfaces (GUIs) in a functional reactive programming style. Essentially, the Oters language aims to provide an intuitive programming framework for creating GUI applications using functional programming. In particular, it makes use of the 'Stream' data type to implement GUI content and behavior. This will be explained in further detail to you shortly.
Participation in this user study is entirely voluntary and as participant, you can withdraw from the project at any time without prejudice, now or in future. If you would like to withdraw, simply let the study conductor know and they will destroy any data collected during the experiment.

C.2 User Study

After a briefing by the conductor of the user study, you will be given six simple programming tasks to complete sequentially using the Oters programming language. These tasks will be performed under a time limit of 30 minutes. Since we are performing user trials for an untested language, we find it important to note that failure to complete any of the tasks will only reflect on the poor design of the language, and not on your abilities. At your disposal you will have the full documentation written by the language's author, and you may additionally reference any other online resources you may wish to use.
It is imperative that you perform the programming tasks in sequential order without peeking at any of the following tasks until you have completed the preceding ones. As each task is printed on a separate page, we therefore ask you not to turn the page until you have completed the task.
Finally, after you have completed the programming tasks, or the time limit runs out

you will be asked to complete a questionnaire regarding your experience programming in Oters. If you have any questions about the user study at any point, do not hesitate to ask the study conductor.
We will be collecting data through a screen recording, from which specific performance measures will be extracted to be later used for the evaluation. To not affect participant performance theses measures will not be divulged until after the study is complete. The screen recordings taken will not be published. Data will further be collected in the form of a questionnaire. Evaluation of the results will be done in an anonymized fashion. No personal data will be published.
You will be programming the application in Visual Studio Code. The coding environment will be set up inside the application's encapsulating Rust project, but you require no knowledge of Rust to complete any of the tasks. A terminal will take up the bottom portion of your screen which you will need to run your program and where you will receive any potential error messages. You will be editing the skeleton code found in the `user_study.otrs` file. To run the code, simply type in the command `cargo run` into the command line and the resulting window will show up automatically. To stop the code running simply close the window.
Once you are ready to begin the study please let the study conductor know to commence

C.3 Programming tasks

1. Edit the skeleton code, to create a simple UI consisting of a label widget and a button widget. These two elements must be arranged vertically, with the label appearing on top of the button. The content of the button must be the word "UP", and the label must display the word "LABEL". Size and position of the elements is left up to your discretion.

   Figure 1 below is an example of what the output should look like. Note that the size and positioning of the UI components is unlikely to be identical and you should not try to make it so. You may also edit the skeleton code to change the layout of the components.

2. Change the UI's behavior such that the top label's text (i.e. the one labelled "LABEL") corresponds to the number of times the button has been clicked. In other words, starting with the label displaying 0, increment the label's text by one each time the button is pressed.

   Hint: The standard library provides functions to convert between types.

3. Add a second button to the right of the first button with a label displaying the word "DOWN". As with the other UI widgets, the size does not matter so long as the label counter remains above the two buttons.

   As before, figure 3 provides an example output, but you should not aim to replicate it.

4. Change the UI's behavior such that each time the "DOWN" button is pressed, the label's counter value is decreased by one.

5. Capture user input such that when the user presses the space-bar the counter resets to zero. That is, when the space-bar is pressed, the label's text should change to display "0". After the counter is reset, the UI should continue operating in the same manner as before, with the buttons incrementing and decrementing the counter appropriately.

   Hint: The space-bar's key name is 'Space'.

6. Draw a rectangle underneath the UI widgets such that it forms a tight bound around them. The precise dimensions around the UI do not matter and the color of the rectangle can be of your choice.

Appendix D

Project Proposal

D.1 Introduction and Description

Functional Reactive Programming (FRP) was introduced by Elliot and Hudak [9] as an alternative paradigm that better fit the needs of programming Graphical User Interfaces (GUI). The imperative programming languages in which most widely used GUI toolkits are written in, implement poorly the higher-order functions needed for reactive behavior. FRP builds programs from Behaviors and Events, which are described as streams of values over continuous time. From this basic model, we can implement reactive behavior by describing functions that are automatically applied to these changing streams of values. For example, take a program that increments and prints a counter whenever the mouse is clicked. This could be written as follows using an FRP toolkit in OCaml-like syntax:

```
let count clk_s acc =
  match clk_s with
  |   $\perp$    :: cs' -> acc :: count cs' acc
  | Click :: cs' -> (acc+1) :: count cs' (acc+1)
  in
let print_count val_s =
  match val_s with
  | v :: vs' -> print v;
                print_count vs'
  in
print_count (count clicks 0)
```

In this example, streams are implemented as lists, with the runtime environment ensuring that these stream functions loop synchronously with the input. $\perp$ is used to describe a non-event.

At first glance, this short example is very succinct and descriptive, illustrating how suitable FRP is for writing reactive programs. This is despite using a familiar syntax not optimized for FRP. There is no need to describe an event loop, nor define any callback behavior. And there are more benefits to using FRP aside from usability. In particular, the declarative nature of FRP allows for easier reasoning about programs. Additionally, the modularity

of stream functions supports idiomatic methods for dynamically changing the structure of a program.

Nevertheless, FRP remains in relative obscurity, used mainly in academia with few widespread practical examples. This leads me to ask: Why is it that FRP hasn't caught on?

In my research into FRP I have identified the following recurring reasons:

1. Computational inefficiency — To match the continuous time semantics of FRP needed for a correct implementation, many FRP libraries, such as *Yampa* [21], require the constant polling of input.

2. Space-time leaky — *Fran* [9], the first FRP language, stored all state belonging to a stream, such that increasing amounts of memory were needed as the program ran.

3. Language of implementation — Functional languages, and in particular Haskell, are used to embed FRP languages like *Reflex* [23]. These "host languages" are far from being widely adopted.

4. Oversaturated market — When they are embedded into more popular languages, like how *Flapjax* [18] is built on Javascript, they compete with already established reactive libraries in their domain.

5. Poor abstraction from mathematics — If the languages implement a research paper's theory like *AdjS* [12], mathematical correctness is prioritized over programming style. As a result, the programmer needs to use qualifiers and expressions with unintuitive mathematical semantics to write a correct program.

6. Lack of documentation — The only documentation for some FRP languages is a research paper describing the theory behind it. This is the case for *Emfrp* [24], making it difficult for new programmers to learn it.

Thus, in this project I aim to reinvigorate FRP, by creating a Domain Specific Language (DSL) that is integrated within the Rust programming language to address the above issues. This DSL will provide an FRP framework with which to intuitively create native graphical applications, with the help of Rust's mature repository of packages.

D.2 Substance and Structure

Out of the FRP approaches described in countless research papers, the one I find provides the most natural programming style, while retaining a rigorous calculus is Bahr et al.'s Simply RaTT [4]. This language borrows from Krishnaswami's [12] modal type system that ensures all programs are causal and don't introduce any space-time leaks. Yet Simply RaTT's specific implementation of modal types, simplifies the type system and makes programs more concise and straightforward.

Thus, the project will initially consist of designing the semantics of the DSL and defining its syntax with a focus on giving users an intuitive way of programming GUIs. Taking Simply RaTT's type system and operational semantics as a base, I will be extending the language in the following ways:

- Introducing new graphical types, such as a canvas or a button.
- Implementing top-level FRP constructions like function lifting and switching expressions.
- Providing a mechanism for initializing and running the graphical application.
- Allowing for the importing of Rust functions, making them compatible with the language's type system. contexts.

The next part of the project will be to construct the compiler for the DSL. This will require building a lexer and a parser, as well as a static analyzer that adheres to the semantics of the modal type system. Aside from the front-end, the compiler will also require translating the abstract syntax tree into Rust code, onto which the imported Rust functions will be added. The integration between Rust and the DSL can be done fairly seamlessly using a build script[1]. Finally, the runtime FRP environment in which the programs will run must be constructed. This runtime will unavoidably need to utilize some graphical library to render the GUI to a window.

Once the DSL is implemented and is fully functional, its efficacy as an alternative GUI framework will need to be evaluated. This evaluation will be three-fold: First, I will write example programs to evaluate the expressiveness of the language, proving its usefulness for writing GUI applications. Second, I will evaluate the performance of the DSL and its runtime system, making sure that there are no space-time leaks. Thirdly, I will conduct a user study to find if using my FRP language actually provides a better programming style over established imperative GUI libraries.

D.3 Starting Point

As of the start of this project, I have some experience with a compiler's front-end as I implemented some small projects with the OCaml lexer and parser. Otherwise, I have no experience with building compilers, aside from what was taught in the Compiler Construction Part IB course. I am also quite proficient with the Rust programming language, having worked on several hobby projects with it in the past. More specifically, I have used a couple native graphic toolkits for Rust GUI development and my familiarity with them will be helpful for this project. Content from the Semantics, Compilers and Further HCI Part IB courses will also be relevant.

[1] `https://doc.rust-lang.org/cargo/reference/build-scripts.html`

Patrick Bahr has made publically available an implementation of the Rattus language[2]—an embedded DSL in Haskell implementing the Simply RaTT type system. While the "host" language is very different to the one I'll be using, it may still be useful as a reference when constructing my own DSL with the same type system.

D.4 Success Criteria

I propose the following criteria to determine if the project is a success:

1. Define the language's type system and semantics, as well as the concrete syntax used by the programmer.
2. Implement the language compiler and companion API to interface with functions written in Rust.
3. Compile and execute GUI applications using the completed language.
4. Evaluate the viability of the DSL as an alternative for writing GUI applications by conducting user studies.

D.5 Extensions

If time permits, the following extensions could be added to project:

1. Evaluate the DSL by benchmarking against other frameworks.
2. Include support for natively interfacing with OpenGL in the DSL.
3. Extend the DSL to import functions from another programming language.
4. Add language constructs for asynchronous behavior, with some streams running at different rates than others.
5. Allow for streams to run concurrently on different threads.
6. Add syntax highlighting for the DSL in a popular editor.

D.6 Plan of Work

14th October – 23rd October

Finalize research and define full language typing rules and operational semantics.

[2] `https://github.com/pa-ba/Rattus`

Consider how to implement GUI components such as images, shapes and windows within the type system.

Define concrete syntax for the language, aiming to abstract Simply RaTT's modal qualifiers and provide higher-order constructs.

Further consider how to integrate Rust functions and potentially data structures into the DSL.

24th October – 6th November

Derive the syntax tokens by building a lexer for the language.

Build a parser to construct an Abstract Syntax Tree (AST) from the tokens derived by the lexer.

Test the ASTs generated and ensure no parsing conflicts exist.

7th November – 20th November

Write the static analyzer and type checker.

This must ensure causality and productivity through the modal types and guarded recursion specified in the semantics.

Test the correctness of the static analyzer with unit tests.

Start constructing the FRP runtime environment for the language.

21st November – 4th December

Implement the main event loop that feeds all the streams with their corresponding new values at each tick.

With a small but functional runtime, translate the AST into Rust instructions that run in the environment.

Implement the heap for the effectful instructions.

Ensure the heap is well typed to prevent space leaks and introduce the garbage collector.

2nd December — End of Michaelmas Term

5th December – 18th December

Continue constructing the runtime environment.

Interface with input libraries giving users access to streams that hold user input.

Add the graphical functionality to the runtime system, interfacing with windowing and GUI libraries.

Translate the rest of the AST that relies on the graphical components.

2nd January – 15th January

Define the Rust API to interact with the DSL.

Write the build script that integrates the DSL code and runtime with the user's imported Rust code.

Ensure that the imported Rust functions correspond to the DSL's modal type system.

16th January – 29th January

Extend the language with typical GUI and debugging functions, as well as more complex FRP constructions such as dynamic switching.

Run unit tests on the language to ensure correct implementation.

17th January — Start of Lent Term

30th January – 12th February

Write the Progress Report and prepare the presentation.

This week will also serve as slack time in the event that the implementation takes longer than expected.

3rd February — Progress Report Deadline

8th - 15th February — Progress Report Presentations

13th February – 26th February

Measure and evaluate the language's performance to test for space-time leaks.

Evaluate the language's expressiveness by using it to write sample programs of increasing complexity.

Plan user study experiments and have them reviewed by someone knowledgeable in the field.

Find people willing to participate in the user study familiar with Rust.

27th February – 12th March

Perform a pilot study on a friend and debug the experiment as needed.

Begin conducting the user study.

13th March – 26th March

Continue the user study if needed before term ends.

Evaluate the results of the user study with appropriate techniques.

Write up the non-substantive sections of the dissertation (e.g. the proforma, bibliography, etc.).

Finish a draft of the Introduction, Preparation and Implementation chapters.

17th March — End of Lent Term

27th March – 9th April

Work on the extensions, time permitting.

Complete a first draft of the Evaluation and Conclusion chapters.

Finish the first draft of the dissertation and hand in for feedback.

10th April – 23rd April

Apply the feedback and improve the dissertation.

Include extension work in the dissertation if applicable.

These two weeks also serve as a time buffer in the case of unforeseen delays.

24th April – 12th May

25th April — Start of Easter Term

Finalize the dissertation.

Ask colleagues to read the dissertation for further feedback and to ensure clarity in writing.

Inevitably cut down on the number of words to meet the word limit.

12th May — Dissertation and Source Code Deadline

D.7 Resource Declaration

To develop this project I will be using my personal laptop computer with the following specifications: Intel Core i7-1195G7 (2.9 GHz), 16GB memory, and 512GB of storage. In the case of computer failure, I will use the University MCS. To safeguard against software failure I will make frequent backups stored on my Google Drive, and also regular backups on an external hard drive. I will also be using GitHub for version control, which will serve as another way of backing up the code repository. *I accept full responsibility for this machine, and I have made contingency plans to protect myself against hardware and/or software failure.* Lastly, all software that I will be needing for the project will be freely available and open-source.