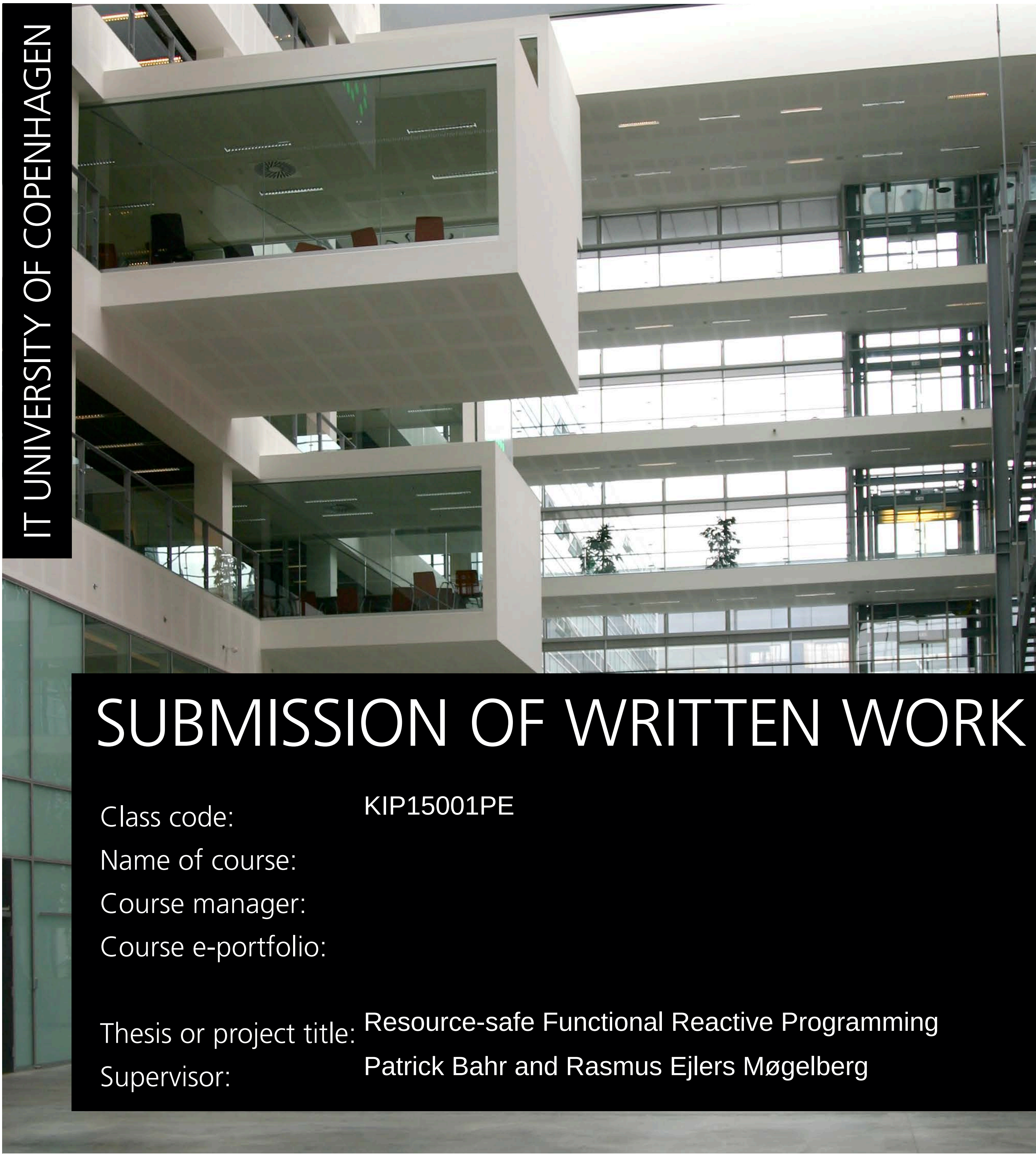
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code: KIP15001PE

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title: Resource-safe Functional Reactive Programming

Supervisor: Patrick Bahr and Rasmus Ejlers Møgelberg

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Adam Bjørn Schønemann	03/04-1991	adsc@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Contents

1 Introduction 2

2 Motivation and Related Research 2

3 Theory 3

3.1 The principle of ModalFRP . . . 4

3.2 Types and Terms . . . 4

3.2.1 Typing rules . . . 4

3.3 Semantics . . . 5

3.4 Examples . . . 6

4 Implementation 6

4.1 Overview and Design . . . 6

4.2 Abstract Syntax . . . 7

4.3 Parsing and concrete syntax . . . 9

4.4 Type-checking and Inference . . . 10

4.4.1 Algorithm W . . . 10

4.4.2 Constraint generation . . . 12

4.4.3 Constraint solving . . . 13

4.4.4 Modifications to Algorithm W . . . 15

4.5 Semantics . . . 17

4.6 Haskell Integration . . . 20

4.6.1 Type-reflection . . . 20

4.6.2 QuasiQuoters . . . 21

4.7 Evaluation with Haskell inputs . . . 22

4.8 Examples . . . 24

5 Empirical evaluation 25

6 Reflection and Future Perspectives 27

Resource-safe Functional Reactive Programming
An Embedded Domain-Specific Language In Haskell

Adam Schønemann, adsc@itu.dk

1 Introduction

This report will elaborate on the design and implementation of the language for safe higher-order functional reactive programming described in [1]. The language is implemented as a domain-specific language in Haskell, using `TemplateHaskell` and `QuasiQuotes`. The report will first briefly describe the language and the motivation behind it, before elaborating on the implementation, which includes an adaptation of the Hindley-Milner algorithm for type inference and a method for reflecting the types of the embedded language into the host language to facilitate type-safe interaction between them.

A note: Since this report is so closely connected to [1] we shall sometimes just write "the paper" when referring to [1].

2 Motivation and Related Research

Developing interactive programs pose a lot of new challenges compared to traditional batch programs. There is a continuous feedback-loop of user-input and program-output instead of well-defined start- and end-points and thus the programs do not terminate by design, except when the user signals termination. Traditional programming languages do not map naturally to this domain, as they are built around a sequential approach to communication, ie. function calls and control flow primitives.

Instead, to construct interactive programs, it is necessary for the different parts of the program to communicate with each-other using callbacks, shared mutable state and concurrency. These are all things that are very complicated to get right, especially as the interactive programs we wish to develop grow in scale and complexity. Furthermore, attempting to verify such programs using formal methods is incredibly hard.

One proposal to fix this is a ”new” paradigm of programming called *Functional Reactive Programming*, first proposed in 1997 [2]. It has gained some popularity in recent years [3–5], but is still far from mainstream.

FRP models inputs and outputs as streams, or signals, of data. Fundamentally, a stream represents a value that varies over time. Signals can then be processed and manipulated using traditional constructs from functional programming. Interactive programs then become ”internally pure” and interact with the outside world at clearly defined boundaries. However, traditional FRP programming comes with several pitfalls:

1. *Causality* - Traditional "naïve" FRP does not enforce causality, i.e. that output at time n only depends on input at and before time n [1]. Since time is modeled as a stream, nothing is stopping you from looking ”ahead” into the stream, which is not semantically well-founded. For example `noncausal (x :: y :: xs) = y :: noncausal(xs)` will always depend on a future value before producing any output, and thus cannot produce output in the current time-step.
2. *Productivity* - Since FRP programs are inherently non-terminating, structural recursion is not useful. However, recursive definitions must still produce output in finite time (productivity), even if they are non-terminating. A program such as `zeros = cons(0, zeros)` is well-defined, yet another program `xs = xs` is not. How do we tell the difference?

3. *Resource usage* - Since FRP programs abstract away resource usage, yet captures time-dependencies between values implicitly, it is easy to inadvertently define a function that depends on the entire past, thus accumulating the entire history of a stream, resulting in a memory leak. For example, `const xs = cons(xs, const xs)` will accumulate the entire history of the `xs` stream, and will at some point run out of memory. However, if `xs` was a non-time dependent value, like a number, this would be totally fine.

Several approaches to solving these problems have been suggested, among them *event-driven FRP* [6] and *arrowized FRP* [7]. Common to all of these approaches is that they limit the expressiveness of classical FRP in order to prevent some or all of these pitfalls. This is typically done by demoting streams from being first-class values, and then exposing some pre-defined combinators in order to construct functions that work on streams. Typically, this entails prohibiting changing the structure of the signal graph at runtime, and thus also all higher-order functions (e.g. of type stream-of-streams). However, these limitations are often to such a degree that they significantly reduce the expressiveness of the FRP paradigm. To regain dynamic behaviour, like switching between two streams, they expose primitive combinators. However, these combinators re-introduce the possiblity of memory issues and non-productivity, and furthermore have an ad-hoc flavour to them [1].

However, there are also approaches that attempt to exploit the type-system to statically ensure causality and productivity, using a technique called *Guarded Recursion*.

Guarded recursion was originally suggested by Nakano [8] to address the challenge of working with coinductive data in proof assistants such as Coq [5] and Agda [9]. Guarded recursion introduces a new modality $\triangleright$ pronounced "later" which makes it possible to distinguish between values we can use now, and values we can only use later. Coincidentally, this distinction is exactly what is needed in order to ensure causality, by constructing a type-system that prohibits using "later" values now. In the same way, guarded recursion ensures productivity, by requiring that definitions return something "now" if they are to be used "now".

Several approaches for using guarded recursion for FRP programming have been proposed [10,11]. There is also currently being researched how to integrate guarded recursion (for coinductive data) with normal structural recursion (for inductive data) [12–16]. However, as long as the domain is restricted to FRP programs, integration with inductive types is less interesting, and shall not be explored further in this section.

The third pitfall is still unadressed – resource usage. Krishnaswami et al proposed using linear types in order to fix this [17], but found that the resulting types became too complicated and inflexible to be usable in practice, as the exact size of the data-flow graph is reflected in the type of a program. In [1], Krishnaswami presented a new programming language that by using guarded recursion, a specific operational semantics, and a modified, more relaxed approach to restricting memory allocation, set out to solve all three of the above mentioned pitfalls. It is the implementation of this programming language that this report deals with.

Rather than implementing the language as a standalone language, we set out to integrate it tightly with Haskell. Using Haskell's `QuasiQuote` mechanism, one can embed an entirely separate language within Haskell. A type-safe integration with Haskell can then be achieved by using Haskell's constructs for type-level programming. Integrating a custom language with Haskell provides several benefits, for example one can choose to implement selected parts of one's application in a language that fits the domain model (FRP in our case) and then piggy-back on Haskell to provide extra libraries and input/output.

3 Theory

This section *briefly* presents the theory from [1]. Since it revolves around the design of a programming language, it would be prudent to give this language a name. As Krishnaswami does not name it in his paper, we shall name it ModalFRP due to its usage of modalities from temporal logic such as "now" , "later" and "always".

3.1 The principle of ModalFRP

The principle of ModalFRP lies in its operational semantics: Programs are evaluated over time in so called "ticks". During one tick, everything is evaluated using call-by-value, so no thunks are generated or built up. However, the programmer can *explicitly* choose to delay a computation to the next time step. Such a delayed computation is allocated in a store. After a tick has completed, all values that are allocated on the store which were available at the tick are simply deleted, and can thus never be used again. All expressions that were delayed on the store are advanced, and are therefore available for the next tick. The evaluation of the program advances by shifting between executing the program in a store σ, ticking the store to σ' and then evaluating the program in the new store. This combination of call-by-value (eager) and call-by-ned (lazy) evaluation ensures that neither time-leaks or space-leaks are possible. However, a new typing discipline is required to ensure that programs do not reference deleted values on the store.

3.2 Types and Terms

ModalFRP closely resembles the simply-typed lambda calculus, but extends it with the modal types $\Box$ ("stable") and $\bullet$ ("later" or "next"). These modalities denote subsets of types whose values are always available, or not availble until the text time-step, respectively. Values of a type without any modality are only available now. Furthermore, the temporal recursive type $\hat{\mu}\alpha.\ A$ is introduced, which defines types that are recursive through time, and a fixpoint for guarded recursion in the style of Nakano [8] is also provided to populate the $\hat{\mu}$ types.

We shall not re-produce the full grammar, semantics and typing rules here. Instead we shall focus on a few of the interesting parts that separate the language from the standard lambda calculus (figure 1).

$$
\begin{array}{lll}
\text{Types} & A & ::= \ \ldots \mid \bullet A \mid \Box A \mid \hat{\mu}\alpha.\ A \mid S\ A \mid \texttt{alloc} \\[1ex]
\text{Terms} & e & ::= \ \ldots \mid \delta_{e'}(e) \mid \texttt{let}\ \delta(x) = e\ \texttt{in}\ e' \mid \texttt{stable}(e) \mid \texttt{let stable}(x) = e\ \texttt{in}\ e' \mid \\
 & & \ \ \texttt{into}\ e \mid \texttt{out}\ e \mid \texttt{cons}(e, e') \mid \texttt{let cons}(x, xs) = e\ \texttt{in}\ e' \mid \texttt{promote}(e) \\
 & & \ \ \texttt{l} \mid \texttt{!l} \mid \diamond \\[1ex]
\text{Values} & v & ::= \ \ldots \mid \texttt{l} \mid \diamond \mid \texttt{stable}(v) \mid \texttt{cons}(v, v') \\[1ex]
\text{Stores} & \sigma & ::= \ \cdot \mid \sigma, \texttt{l} : v\ \texttt{now} \mid \sigma, \texttt{l} : e\ \texttt{later} \mid \sigma, \texttt{l} : \texttt{null}
\end{array}
$$

Figure 1: Syntax extensions to the simply-typed lambda calculus.

As mentioned above, the standard types are augmented with the $\bullet$ and $\Box$ modalities, as well as the temporal recursive type. Added to this is a primitive type of streams S and the type of allocation tokens `alloc`. Functions that have an allocation token in scope can delay values into the next time-step by allocating memory on the store.

Terms include an introduction and elimination form of the $\bullet$ and $\Box$ modalities (δ and `stable` respectively), as well as temporal recursive types (`into` and `out`). Primitives for constructing and deconstructing streams are also added. `promote` takes a term, and if it is of an inherently stable type, like $\mathbb{N}$, we can "promote" it to be `stable`. There is syntax for pointers, represented as `l` for a pointer label and `!l` for a pointer dereference. However, pointers are not surface syntax, and as such can never appear in a user-program. Instead, pointers are used by the evaluator to suspend computations to a later time. Finally, the $\diamond$ represents an allocation token, which is also not part of the surface syntax – only the runtime can create allocation tokens.

3.2.1 Typing rules

The typing context Γ does not simply map names to types, but names to a type paired with a temporal qualifier (figure 2).

$$\begin{array}{llll} \text{Qualifiers} & q & ::= & \texttt{now} \mid \texttt{stable} \mid \texttt{later} \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A\ q \end{array}$$

Figure 2: Qualifiers and typing contexts.

These qualifiers encode when a value can be typed. For example, to delay a value into the next time step, we must type-check it in a context that only retains stable values and values that are also delayed. This can be seen when type-checking $\delta_{e'}(e)$ and `let` $\delta(x) = e$ `in` e' (figure 3). There are similar rules for the `stable` forms, which further restricts the context to only contain values qualified with `stable`. These typing rules serve to only type programs that respect causality, as in "expressions executed in the future can only depend on expressions in the future" and "expressions that are time-independent can only rely on other time-independent expressions".

$$\frac{\Gamma \vdash e : A\ \texttt{later} \qquad \Gamma \vdash e' : \texttt{alloc}\ \texttt{now}}{\Gamma \vdash \delta_{e'}(e) : \bullet A\ \texttt{now}} \bullet\mathrm{I} \qquad \frac{\Gamma \vdash e : A\ \texttt{now} \qquad \Gamma, x : A\ \texttt{later} \vdash e' : C\ \texttt{now}}{\texttt{let}\ \delta(x) = e\ \texttt{in}\ e' : C\ \texttt{now}} \bullet\mathrm{E}$$

$$\frac{\Gamma^{\bullet} \vdash e : A\ \texttt{now}}{\Gamma \vdash e : A\ \texttt{later}}\textsc{TLater} \qquad \begin{array}{lcl} (\cdot)^{\bullet} & = & \cdot \\ (\Gamma, x : A\ \texttt{later})^{\bullet} & = & \Gamma^{\bullet}, x : A\ \texttt{now} \\ (\Gamma, x : A\ \texttt{stable})^{\bullet} & = & \Gamma^{\bullet}, x : A\ \texttt{stable} \\ (\Gamma, x : A\ \texttt{now})^{\bullet} & = & \Gamma^{\bullet} \end{array}$$

Figure 3: Rules and context-operations for delaying values.

Notice that $\bullet$I requires an allocation token to delay a term. This makes sure that all definitions that delay values, and thus extend the store, are marked by having `alloc` as a parameter in their type. As such, `alloc` represents a form of capability. Furthermore, the allocation tokens serve to prevent construction of values of type $\square \bullet A$ (e.g. `stable`($\delta_u(42)$)) – since values of $\bullet A$ are stored on the heap, they are deleted in 2 time-steps, and thus should not be allowed to be marked as stable.

$$\frac{\Gamma \vdash e : [\bullet(\hat{\mu}\alpha.\ A)/\alpha]A\ \texttt{now}}{\Gamma \vdash \texttt{into}\ e : \hat{\mu}\alpha.\ A\ \texttt{now}}\mu\mathrm{I} \qquad \frac{\Gamma \vdash \hat{\mu}\alpha.\ A\ \texttt{now}}{\Gamma \vdash \texttt{out}\ e : [\bullet(\hat{\mu}\alpha.\ A)/\alpha]A\ \texttt{now}}\mu\mathrm{E} \qquad \frac{\Gamma^{\square}, x : A\ \texttt{later} \vdash e : A\ \texttt{now}}{\Gamma \vdash \texttt{fix}\ x.\ e : A\ \texttt{now}}\textsc{Fix}$$

Figure 4: Typing rules for temporal recursive types and fixpoints.

As figure 4 illustrates, the (un)folding of a temporal recursive type mirrors a normal recursive type, except that the substituted type-variable is always guarded in a delay modality. This ensures that only one "step" of the recursive value is unfolded at each tick. Similarly, the judgment for the fixpoint ensures that the recursive call is always guarded, since a value of type A : `later` can only be used in a delayed expression. As a fixpoint will be evaluated at multiple time-steps, it cannot capture any time-dependent expressions in its closure – the $\Gamma^{\square}$ context ensures this.

3.3 Semantics

The semantics are for the most part fairly standard call-by-value semantics. However, expressions can be delayed to the next tick, by using the introduction form of the $\bullet$ modality ($\delta_{e'}(e)$). Semantically, this is modelled by allocating a new expression on the store and returning a pointer to the expression in the store. Figure 5 show the semantics involving delayed expressions.

$$\frac{\langle \sigma, e' \rangle \Downarrow \langle \sigma', \diamond \rangle \qquad \mathtt{l} \notin \mathrm{dom}(\sigma')}{\langle \sigma, \delta_{e'}(e) \rangle \Downarrow \langle \sigma', \mathtt{l} : e\ \mathtt{later}\ ; \mathtt{l} \rangle} \qquad \frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; \mathtt{l} \rangle \qquad \langle \sigma; [!\mathtt{l}/x]e' \rangle \Downarrow \langle \sigma''; v \rangle}{\langle \sigma; \mathtt{let}\ \delta(x) = e\ \mathtt{in}\ e' \rangle \Downarrow \langle \sigma''; v \rangle} \qquad \frac{\mathtt{l} : v\ \mathtt{now} \in \sigma}{\langle \sigma; !\mathtt{l} \rangle \Downarrow \langle \sigma; v \rangle}$$

Figure 5: Semantics for delayed expressions.

3.4 Examples

```
const : S alloc → ℕ → S ℕ
const us n =
  let cons(u,δ(us')) = us in
  let stable(x) = promote(n) in
  cons(x, δᵤ (const us' x))
```

Figure 6: The constant function that repeats a natural number.

```
scary_const : S alloc → S ℕ → S (S ℕ)
scary_const us ns =
  let cons(u,δ(us')) = us in
  let stable(xs) = promote(ns) in -- TYPE ERROR
  cons(xs, δᵤ (scary_const us' xs))
```

Figure 7: The const function that cannot be typed since it causes a memory leak. Note that the only difference lies in the type signature – we cannot constantly repeat time-varying values without causing memory leaks.

4 Implementation

This section presents the implementation of ModalFRP. If one should wish to use the the implementation e.g. to follow along, please refer to the usage instructions found in the README in the git repository[1].

4.1 Overview and Design

If you clone the git repository you get a directory tree as seen in 4.1 (auxiliary- and test-files are omitted).

[1] https://github.com/adamschoenemann/simple-frp

```
src
 `- FRP
     `- AST
         `- Construct.hs
         `- QuasiQuoter.hs
         `- Reflect.hs
     `- Parser
         `- Construct.hs
         `- Decl.hs
         `- Lang.hs
         `- Program.hs
         `- Term.hs
         `- Type.hs
     `- AST.hs
     `- Pretty.hs
     `- Semantics.hs
     `- TypeChecker.hs
     `- TypeInference.hs
 `- FRP.hs
 `- Utils.hs
```

Listing 1: Structure of the source files.

The structure of the files mirror the module hierarchy in the source. `FRP.hs` is the main "entry point" of the project. It re-exports other modules and defines functions to run ModalFRP programs and definitions. `Utils.hs` simply contains a few un-interesting utility functions. The `FRP` module contains most of the implementation. It has two sub-modules: `AST` and `Parser`. The `AST.hs` file contains definitions for the Abstract Syntax of both terms and types of the language, along with type-class instances and functions to work with them. The submodules of `AST` provide helper functions to `Construct` the AST, `QuasiQuoter`s to construct ASTs at compile-time, and methods to `Reflect` the type of the AST into a Haskell type.

The `Parser` sub-modules simply contains parser definitions and helpers, that allow us to parse an AST from a `String` using combinators provided by the `Parsec` library.

`Pretty.hs` defines a tiny type-class that deals with pretty-printing things. `Semantics.hs` contains a complete interpreter of ModalFRP using the operational semantics defined in the paper. `TypeChecker.hs` contains the first implementation of a type-checker for ModalFRP. While this type-checker rigidly follows the typing judgments described in the paper, it does not perform any inference whatsoever, and as such requires programs to be fully annotated. As this is less than ideal, this file is actually deprecated in favor of `TypeInference.hs`, which implements a Hindley-Milner style inference algorithm for the type-system described in the paper.

In total, the implementation measures two-thousand source-lines of Haskell code.

4.2 Abstract Syntax

A program in ModalFRP is simply a list of declarations. A declaration is a name, a type, and a body that is a term. A type is an algebraic data-type (ADT) with constructors that mirror the grammar given in the paper. A term is likewise represented as an ADT that closely resembles the grammar. The Haskell definitions can be seen in listing 2.

```haskell
data Program a = Program [Decl a]
data Decl    a = Decl { _name :: Name, _type :: Type a, _body :: Term a}

-- |A Type in the FRP language
data Type a
 = TyVar    a Name              -- ^Type variable
 | TyProd   a (Type a) (Type a) -- ^Product type
 | TySum    a (Type a) (Type a) -- ^Sum type
 | TyArr    a (Type a) (Type a) -- ^Arrow (function) type
 | TyLater  a (Type a)          -- ^Later type
 | TyStable a (Type a)          -- ^Stable type
 | TyStream a (Type a)          -- ^Type of streams
 | TyRec    a Name (Type a)     -- ^Recursive types
 | TyAlloc  a                   -- ^Type of allocator tokens
 | TyPrim   a TyPrim            -- ^Primitive types

-- |Terms in the FRP language
data Term a
 = -- standard lambda calculus terms --
 | TmCase a (Term a) (Name, (Term a)) (Name, (Term a)) -- ^Case pattern match
 | TmCons a (Term a) (Term a)                          -- ^Stream Cons
 | TmOut  a (Type a) (Term a)                          -- ^@out@ for recursive type elimination
 | TmInto a (Type a) (Term a)                          -- ^@into@ for recursive type introduction
 | TmStable a (Term a)                                 -- ^Attempt to make a value stable
 | TmDelay  a (Term a) (Term a)                        -- ^Delay a value with an allocation token
 | TmPromote a (Term a)                                -- ^Promote a term of a stable type
 | TmLet a Pattern (Term a) (Term a)                   -- ^A let binding with a a pattern
 | TmPntr a Label                                      -- ^A pointer (not syntactic)
 | TmPntrDeref a Label                                 -- ^A pointer dereference (not syntactic)
 | TmAlloc a                                           -- ^An allocator token
 | TmFix a Name (Maybe (Type a)) (Term a)              -- ^A fixpoint

-- |A Pattern used in a let binding
data Pattern
  = PBind Name            -- ^Bind a name
  | PDelay Name           -- ^Bind a delayed name
  | PCons Pattern Pattern -- ^Match on a Cons cell
  | PStable Pattern       -- ^Match on a stable value
  | PTup Pattern Pattern  -- ^Match on tuple
```

Listing 2: AST data-types.

Notice the type-parameter `a` that is present on all constructors. This parameter represents a generic "annotation" that the constructors can be annotated with. In most cases, this is either a `SourcePosition` which denotes the position of an AST node in the source-code, or simply `()` (Unit), which carries no meaning. One could also simply have put a `Maybe SourceCode` parameter at each data constructor, but abstracting that away with a type parameter is actually a more elegant solution, and more extensible to boot. One could for example easily imagine annotating nodes with other data during e.g. optimizations or transformations.

There are many approaches to annotating ADTs with extra data in Haskell, e.g. using type-level fixpoints (the `Fix` type) to factor out the recursion from the underlying functor, or using mutual recursion with an annotation type. While the scheme chosen here requires a bit of boiler-plate, this is one of the simpler ones, which should also help simplify the rest of the implementation.

The `Pattern` ADT allows pattern matching in `let` bindings. Patterns can be nested (except for `PBind` and `PDelay`). The tuple pattern is also included, although it is not mentioned in the syntax in the paper (yet used in examples).

4.3 Parsing and concrete syntax

The concrete syntax of ModalFRP resembles the syntax in the paper closely. However, there are a few differences:

- In Haskell-style, lambdas are represented by back-slashes and the dot becomes an arrow as such: $\lambda x.e \equiv$ `\x` $\rightarrow$ `e`.
- Lambdas can have more than one parameter (this is implemented as syntax-sugar).
- The delay forms represented as δ in the paper are now explicitly written `delay` and the subscripts have been replaced by an extra argument as such: $\delta_{e'}(e) \equiv$ `delay`(`e`$'$, `e`) and `let` $\delta(x) = e$ `in` $e' \equiv$ `let delay`(`x`) $=$ `e in e`$'$.
- The `case` syntax is represented as a mixup of ML and Haskell styles:

```
case e of
  | inl x -> e'
  | inr y -> e''
```

  compared to `case`(e, `inl` $x \rightarrow e'$, `inr` $y \rightarrow e''$) in the paper.
- Lambdas and fixpoints can optionally have type signatures attached to their parameters.
- `into` and `out` must have a type annotation, written in parentheses prior to their arguments. Why this is necessary is explained in detail in section 4.4.
- The product type is written `A * B` instead of $A \times B$.
- The stable type is written `#A` instead of $\Box A$.
- The later type is written `@A` instead of $\bullet A$
- The recursive type is written `mu a. A` instead of $\hat{\mu}\alpha.\ A$
- A few standard constructs have been added to the language like booleans, if-then-else and standard binary operators on numbers and booleans.

The parser is written using the parser combinators exported from the `parsec` package. Parser combinators allow for very short and concise parsers, while still retaining the full expressiveness of the Haskell language and ecosystem. Figure 3 shows the entire parser for types.

```haskell
ty :: Parser (Type SourcePos)
ty =  tyrec <|> buildExpressionParser tytable tyexpr <?> "type"

tyexpr = tynat <|> tybool <|> tyalloc <|> tyvar <|> parens ty

tytable = [ [ prefix' "S" (withPos TyStream)
            , prefix' "#" (withPos TyStable)
            , prefix' "@" (withPos TyLater)
            ]
          , [binary' "*"  (withPos TyProd) AssocRight]
          , [binary' "+"  (withPos TySum)  AssocRight]
          , [binary' "->" (withPos TyArr)  AssocRight]
          ]

tynat   = reserved "Nat"   >> withPos (\p -> TyPrim p TyNat)
tybool  = reserved "Bool"  >> withPos (\p -> TyPrim p TyBool)
tyalloc = reserved "alloc" >> withPos TyAlloc
tyvar   = withPos TyVar   <*> identifier

tyrec = withPos TyRec <*> (reserved "mu" *> identifier <* symbol ".") <*> ty

withPos :: (SourcePos -> a) -> Parser a
withPos fn = fn <$> getPosition
```

Listing 3: The entire parser for types.

Notice the extensive use of the `withPos` combinator. This allows the position of the token to be recorded in the AST. This information can then be used for better error messages later. `Parsec`'s `buildExpressionParser` combinator takes a "table" of operators and a parser and builds a correct expression parser with precedence rules according the position of an operator in the table. Thus, an operator *op* binds tighter than *op′* if *op* appears before *op′* in the table. As can be seen, it can also be specified whether a binary operation associates right, left or not at all.

The parsers for the rest of the surface syntax follow a similar structure as the parser for types, so we will not elaborate further on them.

4.4 Type-checking and Inference

Once the concrete syntax has been parsed into an AST, it must be type-checked to make sure that it can be evaluated safely.

To ease the burden of annotating the program with types, an algorithm for inferring the types of terms has been implemented. The algorithm is an adaptation of the classic inference algorithm "Algorithm W" for Hindley-Milner type systems [18, 19] and later modified in [20] to separate constraint generation and unification. [2]

We shall first present Algorithm W very briefly, and then move on to describing the modifications made to adapt it for ModalFRP.

4.4.1 Algorithm W

Algorithm W works in two stages:

1. Traverse the abstract syntax, gathering constraints on the types of terms based on their form and position in the AST.

[2] Thanks to Stephen Diehl for his excellent tutorial found at http://dev.stephendiehl.com/fun/006_hindley_milner.html

2. Solve all the gathered constraints by *unification*. This yields either an error (the program was not well-typed) or the principal type of the program, that is, the most general type the program can have.

The constraints generated in the first phase are derived from the syntax and the typing rules that are described in the paper. For example, when inferring the type of `\x -> x + 1`, the rules

$$\frac{\Gamma, x : A \ \texttt{now} \vdash e : B \ \texttt{now}}{\Gamma \vdash \lambda x.\ e : A \to B \ \texttt{now}} \qquad \frac{\Gamma \vdash x : \mathbb{N} \ \texttt{now} \qquad \Gamma \vdash y : \mathbb{N} \ \texttt{now}}{\Gamma \vdash x + y : \mathbb{N} \ \texttt{now}}$$

give rise to the goal type $a \to \mathbb{N}$ and the constraints that $[a \sim \mathbb{N}, \mathbb{N} \sim \mathbb{N}]$ as such: When the algorithm encounters the lambda, it will generate a fresh type-variable a and then proceed into the body of the lambda, but with the assumption $x : a$ in the typing context Γ. When it encounters the "+" it will infer the type of the left and right operands, which will be a and $\mathbb{N}$ respectively. It will then generate the constraints that $a \sim \mathbb{N}$ and $\mathbb{N} \sim \mathbb{N}$ according to the rule above.

To then find the principal type of the expression, it will solve these constraints. Each constraint can either be solved, which may cause a substituion to occur, or it cannot, in which case the algorithm terminates with failure. In this case, the first constraint to solve is $a \sim \mathbb{N}$. This constraint unifies, since a is just an arbitrary type, so it can certainly also be a $\mathbb{N}$. Solving this constraint causes us to replace all occurences of a with $\mathbb{N}$, yielding a new goal type $\mathbb{N} \to \mathbb{N}$ and single constraint $[\mathbb{N} \sim \mathbb{N}]$. The last constraint can be solved immediately with no substitutions, since the two types are equal. Once we've solved all the constraints, we've found the principal type which is of course $\mathbb{N} \to \mathbb{N}$.

4.4.1.1 Implementation of Algorithm W A complete walkthrough of the implementation is beyond the scope of this report, however listing 4 shows a few of the definitions that are involved.

```haskell
-- |A substitution is just a mapping from type-variables to their \'actual\' types
type Subst t = Map TVar (Type t)

-- |A quantified (forall) type
data Scheme t = Forall [TVar] (Type t)

-- |A scheme and a 'Qualifier'
type QualSchm t = (Scheme t, Qualifier)

-- |A typing context is a map from names to 'QualSchm's
newtype Context t = Ctx {unCtx :: Map Name (QualSchm t)}

-- |A type exception is a type-error and an associated type context
newtype TyExcept t = TyExcept (TyErr t, Context t)

-- |The state of infer is just a list of fresh names
type InferState = [Name]

-- |An Infer monad writes a list of unification constraints and a list
-- of types that should be stable
type InferWrite t = ([Constraint t], [StableTy t])

-- |Monad that generates unification constraints to be solved later
newtype Infer t a = Infer
  {unInfer :: RWST (Context t) (InferWrite t) InferState
                   (Except (TyExcept t)) a
  }
-- | A 'Constraint' represents that two types should unify
newtype Constraint t = Constraint (Type t, Type t)

-- |A 'StableTy' represents a constraint that a type should be stable
newtype StableTy t = StableTy { unStableTy :: (Type t) }

-- |A 'Unifier' is a substitution that will yield the principal type
-- that satisfies the constraints, or fail
newtype Unifier t = Unifier (Subst t, [Constraint t])

-- |The state of the solver is a unifier and a supply of fresh names
type SolveState t = (Unifier t, [Name])

-- |The 'Solve' monad is a stack of a state-monad with 'SolveState' and
-- an exception monad that throws 'TyExcept's
newtype Solve t a = Solve (StateT (SolveState t) (Except (TyExcept t)) a)
```

Listing 4: Central definitions involed in the type-inference algorithm. The type variable `t` represents the type of annotations.

4.4.2 Constraint generation

In constrast to the normal Algorithm W, the typing context contains not just schemes, but schemes qualified with a temporal qualifier (`QualSchm`). Constraint-generation is implemented as a monadic computation with read/write/state effects (the `Infer` monad). The monad can read from the typing context, write constraints that it generates, and its state contains a supply of fresh names. Fresh names are simply implemented as an infinite list of single-letter strings of the alphabet, prefixed with some increasing number (i.e. $[0a, 0b, 0c, \dots 0z, 1a, 1b, \dots]$).

Every time a fresh name is generated, a name is taken from the head of the list, and the tail is stored as the new state. The fact that the names are *prefixed* with a number make them illegal names for the user to pick, and thus we've separated the machine-generated names from the user-picked names, and we can guarantee that they actually are fresh. More efficient and elegant ways to create fresh names exist, but this simple approach is sufficient for this prototype.

The actual constraint generation is then a straight-forward translation of the typing rules. There are quite a lot of cases, so we will just show two, which can be seen in listing 5 along with the related typing rules.

$$\frac{\Gamma \vdash e : A \to B \ \texttt{now} \qquad \Gamma \vdash e' : A \ \texttt{now}}{\Gamma \vdash e\ e' : B \ \texttt{now}} \to \mathrm{E} \qquad\qquad \frac{\Gamma \vdash e : A \ \texttt{now}}{\Gamma \vdash \texttt{inl}\ e : A + B \ \texttt{now}} + \mathrm{LI}$$

```haskell
infer term = case term of
  -- ... --
  TmApp a e1 e2 -> do
    t1 <- infer e1
    t2 <- infer e2
    tv <- TyVar a <$> freshName
    uni t1 (TyArr a t2 tv)
    return tv

  TmInl a e -> do
    ty <- infer e
    tvr <- TyVar a <$> freshName
    return (TySum a ty tvr)
  -- ... --

-- |Record that two types must unify
uni :: Type t -> Type t -> Infer t ()
uni t1 t2 = tell ([Constraint (t1, t2)], [])
```

Listing 5: Examples of inference rules and their implementations.

Whenever a rule has no specification on the type of an assumption, we simply just infer it. However, when there is an expectation on the form of a type, we encode that expectation by generating a constraint using the `uni` function.

4.4.3 Constraint solving

Solving the generated constraints from the first step, simply amounts to iterating through the constraints and attempting to unify the two types mentioned in each constraint. That is, for each constraint $t_1 \sim t_2$, attempt to unify t_1 with t_2. If unification fails, the expression is ill-typed. If it succeeds, it will give rise to a substitution that is applied to all the remaining constraints. The process then continues with the next constraint. When there are no more constraints, the result is a substitution that can be applied to the goal type inferred in the first step. Applying the substitution yields the principal type of the expression.

Excerpts of the implementation can be seen in listing 6.

```haskell
-- |Solves a Solve monad, or fails if there are no solutions
solver :: Solve t (Subst t)
solver = do
  Unifier (su, cs) <- getUni
  case cs of
    [] -> return su
    (Constraint (t1,t2) : cs0) -> do
      Unifier (su1, cs1) <- unifies t1 t2
      putUni $ Unifier (su1 `compose` su, cs1 ++ (apply su1 cs0))
      solver

-- |Attempt to unify two types
unifies :: Type t -> Type t -> Solve t (Unifier t)
-- A type unifies to itself with no substitution
unifies t1 t2 | unitFunc t1 == unitFunc t2 = return emptyUnifier
-- Any type unifies to a type-variable by binding the type to the name
unifies (TyVar _ v) t = v `bind` t
unifies t (TyVar _ v) = v `bind` t
-- Two function types unifies if their domains and codomains unifies
unifies (TyArr _ t1 t2)  (TyArr _ t3 t4)   = unifyMany [t1,t2] [t3,t4]
-- S a ~ S b if a ~ b
unifies (TyStream _ t1)  (TyStream _ t2)   = t1 `unifies` t2
-- Two recursive types unify if their inner types unify after we've replaced
-- the name of the bound type-variable to a fresh name in both types
unifies (TyRec a af t1)  (TyRec _ bf t2)   = do
  fv <- freshName
  apply (M.singleton af (TyVar a fv)) t1 `unifies`
    apply (M.singleton bf (TyVar a fv)) t2
unifies t1 t2 = do
  unif <- getUni
  typeErr (CannotUnify t1 t2 unif) emptyCtx

-- |Bind a type-variable to a type.
-- Throws an OccursCheckFailed error if the variable occurs in type to be bound.
bind :: TVar -> Type a -> Solve a (Unifier a)
bind a t | unitFunc t == TyVar () a = return emptyUnifier
         | occursCheck a t = do
              unif <- getUni
              typeErr (OccursCheckFailed a t unif) emptyCtx
         | otherwise       = return $ Unifier (M.singleton a t, [])
```

Listing 6: Constraint solving.

Finally, listing 7 demonstrates putting constraint generation and solving together.

```haskell
-- |Run an inference computation and solve it
solveInfer :: Context t -> Infer t (Type t) -> Either (TyExcept t) (Scheme t)
solveInfer ctx inf = do
  (goalTy, freshNames, constraints) <- runInfer ctx inf
  (subst, _) <- runSolve freshNames (Unififer (nullSubst, constraints))
  return (closeOver $ apply subst goalTy)
  where
    closeOver = normalize . generalize emptyCtx
```

Listing 7: Solve the constraints generated by an inference computation.

4.4.4 Modifications to Algorithm W

Since the type system of ModalFRP extends the simply typed lambda calculus with several constructs, it is necessary to modify the standard Algorithm W in some ways.

4.4.4.1 Qualifiers The above algorithm does not take into account the special qualifiers on types, namely `now`, `stable` and `later`. As described in the theory, these qualifiers signify whether something is well-typed now, always or at a later time.

According to the operations of $\Gamma^\bullet$ and $\Gamma^\square$ described in the paper, these qualifiers' interaction with type inference can be implemented during the first step of the inference algorithm (constraint generation). The qualifiers are important in several rules; we shall only look at two of them - the other places are similar.

To infer the of the expresion `stable(`e`)` we must be able to type-check e in a stable context. This means that we must delete all values that are not qualified with `stable` in the typing context. The relevant rules and their implementation are found in listing 8.

$$\frac{\Gamma \vdash e : A\ \texttt{stable}}{\Gamma \vdash \texttt{stable}(e) : \square A\ \texttt{now}}\square\mathrm{I} \qquad \frac{\Gamma^\square \vdash e : A\ \texttt{now}}{\Gamma \vdash e : A\ \texttt{stable}}\textsc{TStable}$$

```haskell
-- |Infer the type of a term.
infer term = case term of
  --- ... ---
  TmStable a e -> do
    t1 <- inferStable e
    return (TyStable a t1)
  --- ... ---

-- |Infer a term in a  /stable/ context
inferStable :: Term t -> Infer t (Type t)
inferStable expr = do
  t <- local stableCtx $ infer expr
  return t

-- |Turn a context into a /stable/ context.
-- This deletes all types in the context that are not stable
stableCtx :: Context t -> Context t
stableCtx (Ctx c1) =
  Ctx $ M.map (maybe (error "stableCtx") (id))
      $ M.filter isJust $ M.map mapper c1
    where
      mapper (t,q) = case q of
        QStable -> Just (t, QStable)
        _       -> Nothing
```

Listing 8: Type-checking of stable terms.

So we can implement the stable types without actually touching the core logic of the inference algorithm, but simply by deleting non-stable names in the typing context.

Similarly for type-checking delayed values of type $\bullet A$, we do the same except we modify the context so that `now` values are deleted, `later` values are moved one "tick" into `now`, and `stable` values are preserved. The relevant rules and their implementation are found in listing 9.

$$\frac{\Gamma \vdash e : A \ \texttt{later} \qquad \Gamma \vdash e' : \texttt{alloc now}}{\Gamma \vdash \delta_{e'}(e) : \bullet A \ \texttt{now}} \bullet \mathrm{I} \qquad\qquad \frac{\Gamma^{\bullet} \vdash e : A \ \texttt{now}}{\Gamma \vdash e : A \ \texttt{later}} \textsc{TLater}$$

```haskell
-- |Infer the type of a term.
infer term = case term of
  --- ... ---
  TmDelay a u e -> do
    tu <- infer u
    uni tu (TyAlloc a)
    te <- inferLater e
    return (TyLater a te)
  --- ... ---

-- |Infer a term to be /later/
inferLater :: Term t -> Infer t (Type t)
inferLater expr = do
  t <- local laterCtx $ infer expr
  return t

-- |Turn a context into a /later/ context.
-- This effectively steps the context one tick, deleting all
-- /now/ types and changing all /later/ values
-- to /now/
laterCtx :: Context t -> Context t
laterCtx (Ctx c1) =
  Ctx $ M.map (maybe (error "laterCtx") (id))
      $ M.filter isJust $ M.map mapper c1
    where
      mapper (t,q) = case q of
        QNow    -> Nothing
        QStable -> Just (t, QStable)
        QLater  -> Just (t, QNow)
```

Listing 9: Rules and code for type-checking later terms.

4.4.4.2 Stable types The concept of *Stable* types of the paper is very simple. A *Stable* type is a type whose values are all time independent, and as such can be freely annotated with a `stable` qualifier using the `promote` construct. Examples are natural numbers and booleans. The rules in figure 8 formalize this notion.

$$\frac{A \ \texttt{stable} \qquad B \ \texttt{stable}}{A \times B \ \texttt{stable}} \qquad\qquad \frac{A \ \texttt{stable} \qquad B \ \texttt{stable}}{A + B \ \texttt{stable}}$$

$$\frac{}{b \ \texttt{stable}} \qquad\qquad \frac{}{\Box A \ \texttt{stable}}$$

Figure 8: Stability of types.

While the concept is simple, the presence of inference complicates the issue of determining whether a type is stable. This is because, during constraint generation, we cannot reliably say anything about the principal type of a term. How do we determine if a type-variable a is of a stable type? We simply cannot. The solution is to defer the decision of stability to the constraint solver, and instead generate a new type of constraint that can only be satisfied if the principle type is stable. This design follows [20], where they use the same method to generalize the type of let bindings (called *let generalization* or *let-polymorphism*).

It is then easy to iterate through the "Stable constraints" after the principal type has been found, and check that it is indeed stable.

4.5 Semantics

Evaluating a term in ModalFRP is defined by the operational semantics in the paper. To evaluate a term, we must first define what a term evaluates to, called a `Value`. A `Value` is a term that is evaluated to normal form. The ADT for `Value` can be seen in listing 10.

```
type EvalTerm = Term ()
-- |A Value is a term that is evaluated to normal form
data Value
  -- |A tuple
  = VTup Value Value
  -- |Left injection
  | VInl Value
  -- |Right injection
  | VInr Value
  -- |A closure
  | VClosure Name EvalTerm Env
  -- |A pointer
  | VPntr Label
  -- |An allocation token
  | VAlloc
  -- |A stable value
  | VStable Value
  -- |A value of a recursive type
  | VInto Value
  -- |A stream Cons
  | VCons Value Value
  -- |A value literal
  | VLit Lit
```

Listing 10: ADT for `Values`.

As such, `Value` is a subtype of `Terms`, except for `VClosure` which is a lambda closing over an environment[3]. Furthermore, a type of store, called σ in the paper, must be defined. The store contains pointers that point to either **a)** a `later` term to be evaluated later or **b)** a `now` value that can be used immediately. To model this, the store is represented as a map from pointer labels (`Label`) to `StoreVals`. The complete definitions can be seen in listing 11.

```
-- |A pointer label is just an Integer
type Label = Integer

-- |An item in the store
data StoreVal
  -- |is a value that is available now or
  = SVNow Value
  -- |a term along with an environment that is available later
  | SVLater EvalTerm Env

-- |A store is a map from pointer-labels to store values
type Store = Map Label StoreVal
```

Listing 11: The store σ modeled in Haskell.

[3]Hence there is no injection from `VClosure` to `TmLam`.

However, one more thing is needed to model the store, because we must be able to create new pointer-labels when we allocate a value on the store. Therefore, we can deduce that the evaluation must happen in a `State` monad with the state described in listing 12.

```
-- |The state for the Eval monad is a store and a counter
-- that generates pointer labels
data EvalState = EvalState
  { _store    :: Store
  , _labelGen :: Int
  }
```

Listing 12: State for the evaluation monad.

Furthermore, the operational semantics in the paper use substitution to bind names in e.g. function application or let bindings. Rather than explicitly substituting terms, we can carry around an environment that map names to terms. This is both simpler to implement and more performant. Thus, the final definition of the evaluation monad can be seen in listing 13.

```
type Env = Map String (Either EvalTerm Value)
newtype EvalM a = EvalM {unEvalM :: StateT EvalState (Reader Env) a}
```

Listing 13: The monad for evaluation.

The environment can hold either `Terms` or fully-evaluated `Values`. In a fully strict language, this would only be `Values`, since all functions would be call-by-value. However, the semantics for the fixpoint (listing 14) require lazy evaluation in order to unfold the fixpoint.

$$\frac{\langle \sigma; [\mathtt{fix}\ x.\ e/x]e\rangle \Downarrow \langle \sigma', v\rangle}{\langle \sigma; \mathtt{fix}\ x.\ e\rangle \Downarrow \langle \sigma', v\rangle}$$

Listing 14: Semantics for the fixpoint.

The evaluation is then a relatively straightforward translation of the operational semantics into Haskell code. Some examples can be seen in listing 15. The two rules for `case` have been collapsed into one, since that is more natural in Haskell. The substitutions are emulated by locally inserting the values into the environment (`local`) and evaling the next term. The evaluation of `TmDelay` shows how to allocate a term on the store and return a pointer to it using `allocVal`. The condition that $\mathtt{l} \notin \mathrm{dom}(\sigma')$ is satisfied since we always generate a new label `l` by simply incrementing a counter.

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \texttt{inl}\ v\rangle \qquad \langle \sigma'; [v/x]e'\rangle \Downarrow \langle \sigma'', v''\rangle}{\langle \sigma; \texttt{case}(e, \texttt{inl}\ x \to e', \texttt{inr}\ y \to e'') \Downarrow \langle \sigma''; v''\rangle}\ \textsc{Case}_L \qquad \frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \texttt{inr}\ v\rangle \qquad \langle \sigma'; [v/y]e'\rangle \Downarrow \langle \sigma'', v''\rangle}{\langle \sigma; \texttt{case}(e, \texttt{inl}\ x \to e', \texttt{inr}\ y \to e'') \Downarrow \langle \sigma''; v''\rangle}\ \textsc{Case}_R$$

$$\frac{\langle \sigma, e'\rangle \Downarrow \langle \sigma', \diamond\rangle \qquad \mathtt{l} \notin \mathrm{dom}(\sigma')}{\langle \sigma, \delta_{e'}(e)\rangle \Downarrow \langle \sigma', \mathtt{l} : e\ \texttt{later}; \mathtt{l}\rangle}\textsc{Delay}_I$$

```haskell
-- |Main evaluation function. This encodes the operational semantics from the paper
eval :: EvalTerm -> EvalM Value
eval term = case term of
  -- ... --
  TmCase _a trm (nml, trml) (nmr, trmr) -> do
    res <- eval trm
    case res of
      VInl vl -> local (M.insert nml (Right vl)) $ eval trml
      VInr vr -> local (M.insert nmr (Right vr)) $ eval trmr
      _       -> crash "not well-typed"

  -- delay a term by allocating it on the store
  TmDelay _a e' e -> do
    v <- eval e'
    case v of
      VAlloc -> do
        env' <- ask
        label <- allocVal (SVLater e env')
        return $ VPntr label
      _ -> crash $ "expected VAlloc, got" ++ ppshow v

-- |Allocate a value on the store, changing the state
allocVal :: StoreVal -> EvalM Label
allocVal v = do
  label <- genLabel
  modify (alloc label)
  return label
  where
    alloc label st = st { _store = M.insert label v (_store st) }
```

Listing 15: Implementation of operational semantics.

To advance the evaluation beyond one step, we must "tick" the store, thereby removing `now` values and promoting `later` terms to `now` values. Listing 16 shows the tick semantics specified in the paper and the corresponding Haskell implementation.

$$\frac{}{\cdot \Longrightarrow \cdot} \qquad \frac{\sigma \Longrightarrow \sigma' \qquad \langle \sigma', e \rangle \Downarrow \langle \sigma'', v \rangle \qquad \mathtt{l} \notin \mathrm{dom}(\sigma'')}{\sigma, \mathtt{l} : e \ \mathtt{later} \Longrightarrow \sigma'', \mathtt{l} : v \ \mathtt{now}}$$

$$\frac{\sigma \Longrightarrow \sigma' \qquad \mathtt{l} \notin \mathrm{dom}(\sigma')}{\sigma, \mathtt{l} : v \ \mathtt{now} \Longrightarrow \sigma', \mathtt{l} : \mathtt{null}} \qquad \frac{\sigma \Longrightarrow \sigma' \qquad \mathtt{l} \notin \mathrm{dom}(\sigma')}{\sigma, \mathtt{l} : \mathtt{null} \Longrightarrow \sigma', \mathtt{l} : \mathtt{null}}$$

```
-- There is an implicit ordering imposed here, so we evaluate the
-- pointers from smallest-to-biggest since allocated values can
-- only depend on other allocations with a pointer label that is
-- smaller than their own.
tick :: EvalState -> EvalState
tick st
  | M.null (_store st) = st
  | otherwise = M.foldlWithKey' tock st (_store st) where
      tock acc k (SVLater e env) =
          let (v, st') = runExpr acc env e
          in  st' { _store  = M.insert k (SVNow v) (_store st') }
      tock acc k (SVNow _)  = acc { _store = M.delete k (_store acc) }
```

Listing 16: Tick semantics and the Haskell implementation.

The sequential nature of the tick semantics is captured by the left-fold over the map (`M.foldlWithKey'`). The pattern matches in `tock` then correspond to the second and third rule of the semantics, respectively. The last rule is totally dispensed with, since there is no reason for us to keep around references to lots of null-pointers – we simply remove them from the map completely.

4.6 Haskell Integration

It is possible to evaluate closed programs in ModalFRP using the methods described above. However, its usefulness as an embedded language is limited if it cannot interoperate with the host language. Thus, there must be a way to convert values from ModalFRP into Haskell, and from Haskell into ModalFRP. This relation is captured as a function that takes a Haskell stream and a program in ModalFRP and returns a new Haskell stream that has been processed by the ModalFRP program – i.e. a stream transformer of type `transform :: [a] -> Term t -> [b]`. The problem here is that we cannot statically guarantee that such a definition will be well-formed, since we have no way of knowing what type in ModalFRP the `Term` has. Does it expect a stream of `a`s? Or something completely different? To regain this static safety, we must reflect the type of a term in ModalFRP into the Haskell type system.

To do this, we must take advantage of several of the Haskell extensions that have been implemented by GHC. Specifically, the `DataKinds` and `GADTs` extensions opens up Haskell to some forms of type-level programming.

4.6.1 Type-reflection

With `DataKinds` enabled, GHC automatically promotes every suitable datatype to be a *kind*, and its value constructors to be type constructors. A *kind* is the type of a type. For example the kind of `Int` is `*` while the kind of `[]` is `* -> *` and the kind of `Map` is `* -> * -> *`. The only types that are inhabitable[4] have kind `*`.

With `DataKinds`, the ADT in listing 17 also give rise to a new kind called `Ty` along with equivalently named type-constructors (one for each value-constructor) that when fully applied has kind `Ty`. For example the kind of `Int` is `*` while the kind of `TNat` is `Ty`, and the kind of `(:*:)` is `Ty -> Ty -> Ty`.

[4] As in "can have concrete values" , and not the opposite meaning of the word.

```
-- | The type of FRP-types that can be reflected
data Ty
  = TNat
  | TBool
  | TAlloc
  | Ty :*:  Ty
  | Ty :+:  Ty
  | Ty :->: Ty
  | TStream Ty
```

Listing 17: Representation of ModalFRP types that can be reflected into Haskells' type system.

Using a GADT (generalized algebraic data type) we can now parameterize a type over the `Ty` kind, as seen in listing 18.

```
-- |Singleton representation to lift Ty into types
-- using kind-promotion
data Sing :: Ty -> * where
  SNat    :: Sing TNat
  SBool   :: Sing TBool
  SAlloc  :: Sing TAlloc
  SProd   :: Sing t1 -> Sing t2 -> Sing (t1 :*:  t2)
  SSum    :: Sing t1 -> Sing t2 -> Sing (t1 :+:  t2)
  SArr    :: Sing t1 -> Sing t2 -> Sing (t1 :->: t2)
  SStream :: Sing t  -> Sing (TStream t)
```

Listing 18: Singleton type for type-level reflection.

The type defined in listing 18 is an example of a so-called *singleton type*. A singleton type only has one value (hence the name), and thus a value of a singleton-type has a unique type that represents that value. Therefore, there is an isomorphism between a value of a singleton type and its type, and since types can depend on types in Haskell, this construction can simulate a limited form of dependent types (i.e. types that depend on values).

With this in place one can associate a `Term` with a Haskell type that reflects its `Type` (listing 19).

```
-- |An FRP program of a type, executed in an environment
data FRP :: Ty -> * where
  FRP :: Env -> Term () -> Sing t -> FRP t
```

Listing 19: A term in ModalFRP with its type reflected into Haskell.

Note that there is no static guarantee that we cannot associate an incorrect type with a term. However, all construction of `FRP` values should happen at compile time, and should not be the responsibility of a user, so the burden of not abusing this data-type falls only on us.

4.6.2 QuasiQuoters

To use the ModalFRP programs in Haskell, we must have access to their definitions at compile time. Writing out the AST by hand is painful, and thus not a possibility. However, GHC Haskell has excellent support for embedding custom languages inside it using `QuasiQuoter`s. Using `QuasiQuoter`s, you can embed any string inside a Haskell source file, and then specify at compile time how to generate Haskell ASTs from this string. Using this mechanism along with the parser and type-checker for ModalFRP, users can write ModalFRP programs using the syntax described previously, and have their programs parsed and type-checked at compile time. Figure 20 shows the implementation of the `QuasiQuoter` for `Decls`.

```haskell
-- |Quote and type-check a declaration
decl :: QuasiQuoter
decl = QuasiQuoter
  { quoteExp = quoteFRPDecl
  , quotePat = undefined  -- we don't use it in pattern positions
  , quoteDec = undefined  -- we don't use it in declaration positions
  , quoteType = undefined -- we don't use it in type positions
  }

-- |Quote and type-check a declaration
quoteFRPDecl :: String -> Q Exp
quoteFRPDecl s = do
  dcl <- parseFRP P.decl s
  case inferDecl' dcl of
    Left err -> fail . show $ err
    Right ty -> do
      sing <- typeToSingExp (_type dcl)
      trm  <- dataToExpQ (const Nothing) (unitFunc $ _body dcl)
      env  <- dataToExpQ (const Nothing) initEnv
      return $ ConE (mkName "FRP") `AppE` env `AppE` trm `AppE` sing
```

Listing 20: The QuasiQuoter for declarations.

`QuasiQuoter`s are tightly integrated with `TemplateHaskell` which is Haskell's system for meta-programming. The result of a quotation in expression position is a Haskell expression, represented by its Haskell abstract syntax tree. We get the ModalFRP program as a string, and use our normal parser to parse it into a declaration. We then infer its type, and if it does not type-check the computation crashes which, since this occurs at compile-time, will also become a compile-time error. We then convert the type to its singleton representation, and the term to its Haskell AST representation using the generic `dataToExpQ` function. Finally, we construct the Haskell AST of a `FRP` value using combinators exposed by `TemplateHaskell`.

4.7 Evaluation with Haskell inputs

Integrating Haskell values as in `transform :: [a] -> Term t -> [b]` also poses the problem of how to deal with Haskell values during evaluation. To this end, we must first add a typeclass that specifies how to marshall Haskell values to ModalFRP values, and back again (listing 21).

```haskell
class FRPHask (t :: Ty) (a :: *) | a -> t where
  toHask :: Sing t -> Value -> a
  toFRP :: Sing t -> a -> Value

-- Example instance
instance FRPHask TNat Int where
  toHask sing (VLit (LNat x)) = x
  toHask sing v               = haskErr "expected nat value" sing v
  toFRP _ x = VLit . LNat $ x
```

Listing 21: Type-class that represents a conversion from a FRP value of a type to a Haskell value of a type.

Second, we must add a term to the language that represents a time-varying input from the outside world. Thus the constructor `TmInput a Name` is added to the `Term` ADT, and the evaluator monad must carry around an additional environment that contains the value of the inputs at the current tick. `TmInput` is not surface syntax, and cannot be written by the user. It is only inserted by the runtime. Then, we must add a rule to the implementation of the semantics that pulls a value from the input environment and uses it in a computation (listing 22).

```haskell
newtype Inputs = Inputs (Map Name Value)

data EvalRead = EvalRead { _env :: Env, _inputs :: Inputs }

-- |The Eval monad handles evaluation of programs
newtype EvalM a = EvalM {unEvalM :: StateT EvalState (Reader EvalRead) a}

getInput :: Name -> EvalM Value
getInput nm = do
  Inputs inputs <- _inputs <$> ask
  let v = unsafeLookup nm inputs
  l <- allocVal (SVLater (TmInput () nm) initEnv)
  return (VCons v $ VPntr l)

-- |Main evaluation function. This encodes the operational semantics from
-- the paper
eval :: EvalTerm -> EvalM Value
eval term = case term of
  -- ... --
  TmInput _a nm -> getInput nm
  -- ... --
```

Listing 22: Extra constructs added to the implementation to deal with inputs.

Using these definitions, we can construct the `transform` function, seen in listing 23.

```haskell
-- |Use a FRP program to transform a Haskell stream @[a]@ to @[b]@
transform :: (FRPHask t1 a, FRPHask t2 b)
          => FRP (TStream TAlloc :->: TStream t1 :->: TStream t2)
          -> [a] -> [b]
transform frp [] = []
transform (FRP env trm (us `SArr` SStream s1 `SArr` SStream s2)) as =
  run initialState (mkExpr trm) as
  where
    run sig e []       = []
    run sig e (x : xs) = step (runExpr (tick inputs sig) inputs env e)
      where
        inputs = mkInputs x
        step (VCons y (VPntr l), sig') = toHask s2 y : run sig' (tmpntrderef l) xs
        step v                         = crash v

    mkInputs x = Inputs (M.singleton "input" (toFRP s1 x))
    crash v    = error $ "got " ++ ppshow v ++ " expected VCons x (VPntr l)"
    mkExpr tm  = tm <| fixed tmalloc <| TmInput () "input"
```

Listing 23: The `transform` stream-transformer.

4.8 Examples

```
frp_ones :: FRP (TStream TAlloc -> TStream TNat)
frp_ones = [prog|
  const : S alloc -> Nat -> S Nat
  const us n =
    let cons(u, delay(us')) = us in
    let stable(x) = promote(n) in
    cons(x, delay(u, const us' x)).

  main : S alloc -> S Nat
  main us = const us 1.
|]

frp> take 10 . execute $ frp_ones
[1,1,1,1,1,1,1,1,1,1]
```

Listing 24: Example of a program that outputs 10 “1”s.

```
frp_switch_safe :: FRP (TStream TAlloc :->: TStream TNat :->: TStream TNat)
frp_switch_safe = [prog|
  nats : S alloc -> Nat -> S Nat
  nats us n =
    let cons(u, delay(us')) = us in
    let stable(x) = promote(n) in
    cons(x, delay(u, nats us' (x + 1))).

  switch : S alloc -> S a -> (mu f. S a + f) -> S a
  switch us xs e =
    let cons(u, delay(us')) = us in
    let cons(x, delay(xs')) = xs in
    case out (mu f. S a + f) e of
      | inl ys -> ys
      | inr t  -> let delay(e') = t in
                  cons(x, delay (u, switch us' xs' e')).

  eventually : S alloc -> Nat -> S a -> (mu f. S a + f)
  eventually us n xs =
    if n == 0
      then into (mu f. S a + f) inl xs
      else let cons(u, delay(us')) = us in
           let cons(x, delay(xs')) = xs in
           let stable(n') = promote(n)  in
           into (mu f. S a + f) inr delay(u, eventually us' (n' - 1) xs').

  main : S alloc -> S Nat -> S Nat
  main us xs =
    let e = eventually us 5 (nats us 0) in
    switch us xs e.
|]

frp> take 11 $ transform frp_switch_safe [10,9..]
[10,9,8,7,6,5,6,7,8,9,10]
```

Listing 25: Example of a stream-transformer that for 5 steps outputs the input stream, then resumes with natural numbers.

```
frp_add :: FRP (TStream TAlloc :->: TStream (TNat :*: TNat) :->: TStream TNat)
frp_add = [prog|
  main : S alloc -> S (Nat * Nat) -> S Nat
  main us ps =
    let cons(u, delay(us')) = us in
    let cons((x,y), delay(ps')) = ps in
    cons(x+y, delay(u, main us' ps')).
|]

main :: IO ()
main = interact (unlines . process . lines) where
  process = map (("result: " ++) . show) . transform frp_add .
            map (read :: String -> (Int,Int))
```

Listing 26: An interactive program that waits for the user to input (x, y) then returns $x + y$ and then waits for user input again.

Listing 25 shows that you can encode the type of events using a temporal recursive type $E\ A \triangleq \hat{\mu}\alpha.\ A + \alpha$, which says that the value of an event is either that something occured or that we're still waiting. In ModalFRP and with $A =$ `S a` this is written `mu f. S a + f`.

5 Empirical evaluation

Figure 9 and 10 show that the memory usage of ModalFRP programs grow as expected – when there is no explicit buffering, it remains constant, while the `scary_const` program grows linearly with time.

While this shows that memory usage is constant for a specific programs, we cannot of course prove that this is true for all programs, especially since the language certainly allows unbounded memory usage – you just need to be very explicit about it.

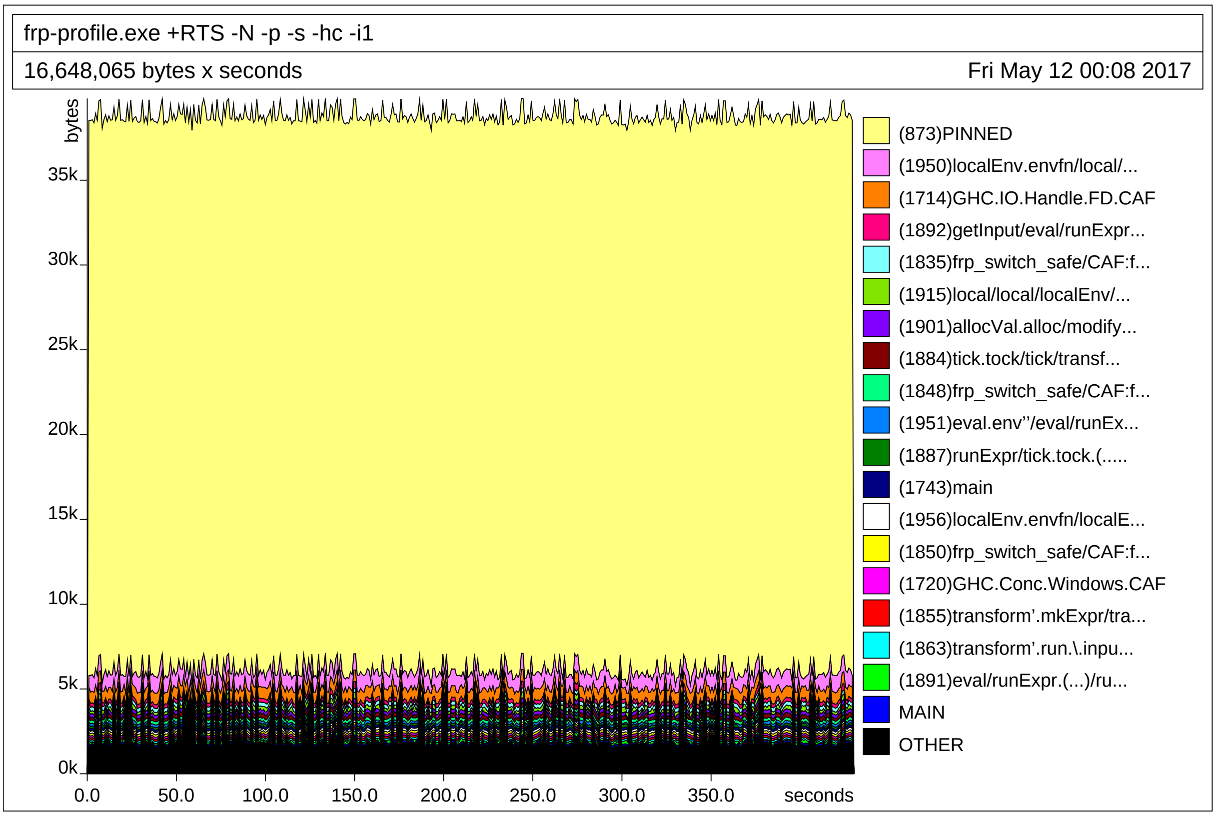


Figure 9: Memory usage of the `frp_switch_safe` program from listing 25 running on Windows 10.

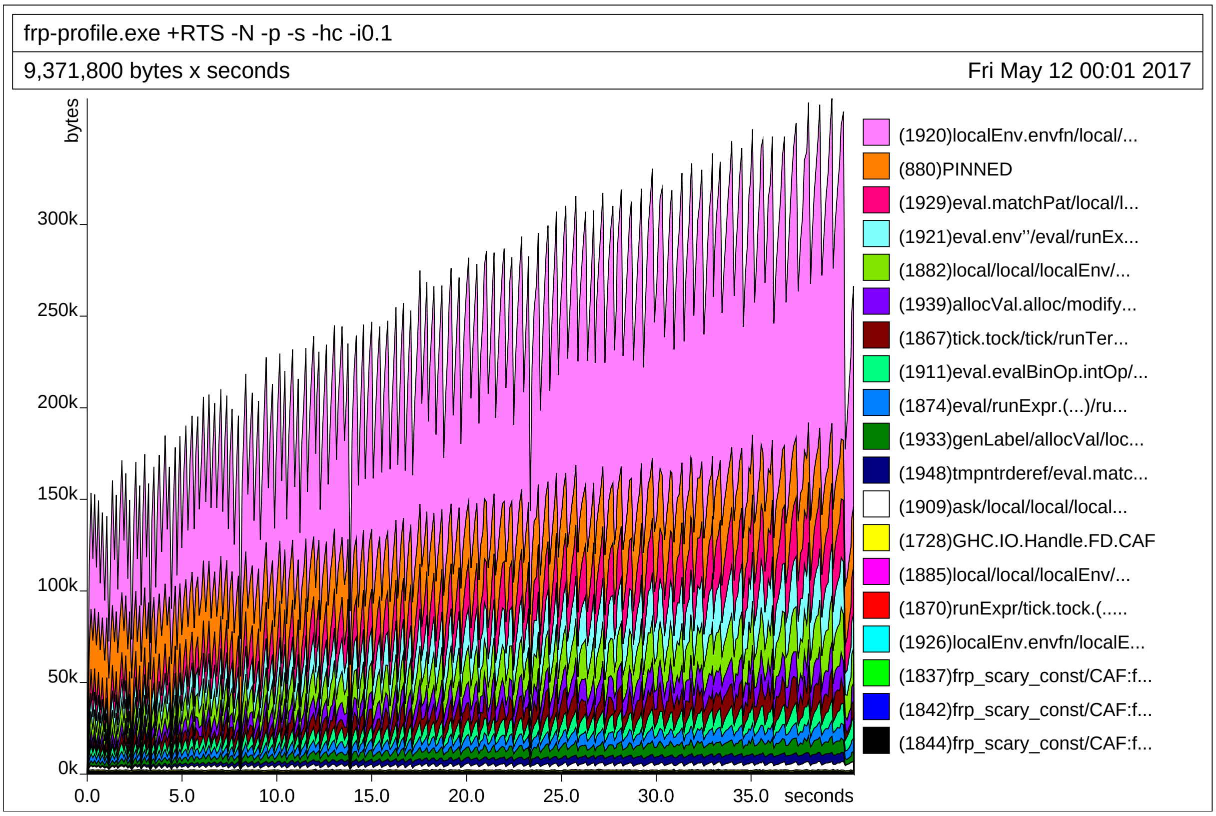


Figure 10: Memory usage of the `scary_const` program from listing 7 running on Windows 10.

All programs that appear in the paper have been encoded and type-checked in ModalFRP (with the exception of `par` which is a bit unwieldy to write in the current syntax which does not support type aliases).

Type-checking the programs revealed a bug in the `fixedpoint` definition (listing 27).

```
X ≜ μ̂α. □(S alloc → α → A)

selfapp : (•A → A) → S alloc → X → A
selfapp f us v =
  let cons(u, δ(us')) = us in
  let stable(w) = out v in
  f (δ_u (w us' (into (stable w))))

fixedpoint : □(•A → A) → S alloc → A
fixedpoint h us =
  let stable(f) = h in
  selfapp f us (into (stable (selfapp f)))
```

Listing 27: `fixedpoint` as defined in the paper.

The problem here is manifested in two places. The first instance can be seen from the typing context in the expression of `selfapp` (figure 28).

```
f  : (•A → A) now
us : S alloc now
v  : μ̂α. □(S alloc → α → A) now
u  : alloc now
us' : S alloc now
out v : □(S alloc → •(μ̂α. □(S alloc → α → A)) → A) now
w : S alloc → •(μ̂α. □(S alloc → α → A)) → A stable
```

Listing 28: Typing context in `selfapp`.

The application of `w us'` to `into (stable w)` does not type-check since the type of `into (stable w)` is `μ̂α. □(S alloc → α → A)` and thus not guarded by a delay modality.

Second, the expression `into (stable (selfapp f))` does not typecheck either, as `into` expects a type with a recursive type guarded by a delay modality, but the expression's type is `S alloc → (μ̂α. □(S alloc → α → A)) → A`. Again, the delay modality is missing.

Instead, the correct program can been seen in listing 29 presented in the concrete syntax of ModalFRP.

```
selfapp : (@a -> a) -> S alloc -> @(mu af. #(S alloc -> af -> a)) -> a
selfapp f us v =
  let cons(u, delay(us')) = us in
  let delay(x) = v in
  f delay(u,
    let stable(w) = out (mu af. #(S alloc -> af -> a)) x in
    let cons(u', us'') = us' in
    let y = delay(u', into (mu af. #(S alloc -> af -> a)) stable(w)) in
    w us' y
  ).

fixedpoint : #(@a -> a) -> S alloc -> a
fixedpoint h us =
  let cons(u, delay(us')) = us in
  let stable(f) = h in
  let delay(y) = delay(u, into (mu af. #(S alloc -> af -> a)) stable(selfapp f)) in
  selfapp f us delay(u,y).
```

Listing 29: The correct definition of fixedpoint in ModalFRP.

The delay modality is added to the third argument of `selfapp` and the implementations have thus become slightly more involved, by using delay twice. Also note that the concrete syntax of ModalFRP requires type annotations on `into` and `out`, and this introduces quite a bit of visual noise.

While there is no guarantee on the correctness of the implementation (beyond the unit-tests), the above example shows that the type-checker is at least "correct-enough" to find bugs in the paper.

6 Reflection and Future Perspectives

This report has detailed the implementation of the language described in [1]. The implementation shows that it is indeed possible to implement the language as a DSL in Haskell, and to integrate the language with Haskell so that one can, in principle, have the causality, productivity and resource-usage guarantees that ModalFRP provides in parts of one's Haskell programs today.

Needless to say, this is just a prototype, and as such lacks many of the conveniences that modern functional programmers are accustomed to, such as modules, algebraic data-types, records, type aliases, let-polymorphism and polymorphic recursion. Therefore, this prototype is of course not suited for practical programming. However, there is nothing that suggests that the addition of any of these features should compromise any of the guarantees provided by ModalFRP.

Performance is also an area that needs work. As it stands now, the performance of the interpreter is far from optimal. However, obvious optimizations are possible, like exchanging the `State` monad for `ST` or even `IO`. Perhaps a more interesting approach is exploring whether the ModalFRP AST can be directly translated into equivalent Haskell code at compile-time using `TemplateHaskell`. Since the type-system of ModalFRP in itself guarantees that memory leaks cannot occur, one can create an almost one-to-one translation to Haskell by simply eliding the modalities, allocation tokens and associated terms from the input AST. However, care should be taken to preserve the eager evaluation semantics to prevent build-up of thunks. Such a scheme would essentially give zero run-time overhead to ModalFRP programs compared to Haskell! It might also make the type reflection redundant, since we get a "true" Haskell term at compile time. An experimental implementation reveals that this may very well be the case and could definitely be a way forward for the project (the `QuasiQuoter` to do this is called `hsprog`).

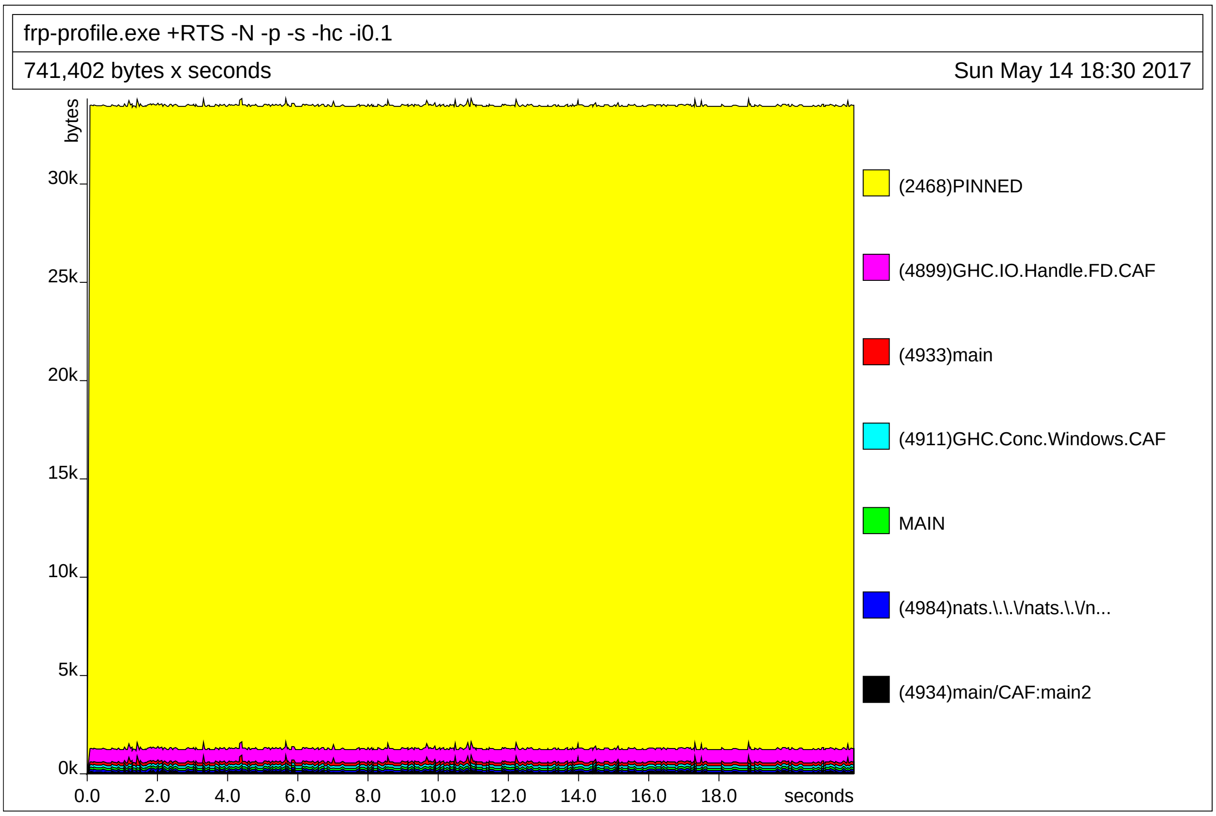


Figure 11: Memory graph of `frp_switch_safe` when directly compiled down to Haskell using the experimental QuasiQuoter.

Figure 11 shows the memory usage of the same program as in figure 9. It shows that memory usage is still constant, and slightly smaller. More interestingly, the program terminates in far less time (22s vs 450s).

Adding inductive datatypes would extend the expressiveness of the language greatly. One would need to strictly separate temporal coinductive types with guarded self-references as in [14], and inductive types (which are non-temporal). Inductive types would also require adding a recursion principle such as structural or primitive recursion.

Bridging the gap between finite and infinite data is an interesting research question. A function such as `observe : ℕ → S a → List A` cannot be typed, as it would violate causality grossly! However, perhaps adding dependent types could help construct a precise enough type, such as `observe : (n : ℕ) → S a → •ⁿ (List A)` which says that in n time steps we can observe the nth prefix of a stream as a list.

References

[1] N. R. Krishnaswami, "Higher-order reactive programming without spacetime leaks," in *International Conference on Functional Programming (ICFP)*, Sept. 2013.

[2] C. Elliott and P. Hudak, "Functional reactive animation," *SIGPLAN Not.*, vol. 32, pp. 263–273, Aug. 1997.

[3] E. Czaplicki, "Elm: Concurrent frp for functional guis," *Senior thesis, Harvard University*, 2012.

[4] Y. H. Group, "Yampa," 2003.

[5] T. C. development team, "The coq proof assistant," 2004.

[6] Z. Wan, W. Taha, and P. Hudak, "Event-driven frp," in *International Symposium on Practical Aspects of Declarative Languages*, pp. 155–172, Springer, 2002.

[7] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 51–64, ACM, 2002.

[8] H. Nakano, "A modality for recursion," in *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*, pp. 255–266, IEEE, 2000.

[9] U. Norell, *Towards a practical programming language based on dependent type theory*, vol. 32. Citeseer, 2007.

[10] N. R. Krishnaswami and N. Benton, "Ultrametric semantics of reactive programs," in *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pp. 257–266, IEEE, 2011.

[11] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka, "Fair reactive programming," in *ACM SIGPLAN Notices*, vol. 49, pp. 361–372, ACM, 2014.

[12] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal, "Guarded dependent type theory with coinductive types," 2015.

[13] R. Atkey and C. McBride, "Productive coprogramming with guarded recursion," in *ACM SIGPLAN Notices*, vol. 48, pp. 197–208, ACM, 2013.

[14] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal, "Programming and reasoning with guarded recursion for coinductive types," in *International Conference on Foundations of Software Science and Computation Structures*, pp. 407–421, Springer, 2015.

[15] A. Vezzosi, *Guarded Recursive Types in Type Theory*. PhD thesis, Chalmers University of Technology, 2015.

[16] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg, "The clocks are ticking: No more delays! reduction semantics for type theory with guarded recursion," in *Logic in Computer Science (LICS), 2017 32snd Annual ACM/IEEE Symposium on*, ACM/IEEE, 2017.

[17] N. R. Krishnaswami, N. Benton, and J. Hoffmann, "Higher-order functional reactive programming in bounded space," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 45–58, 2012.

[18] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[19] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the american mathematical society*, vol. 146, pp. 29–60, 1969.

[20] B. Heeren, J. Hage, and S. D. Swierstra, "Generalizing hindley-milner type inference algorithms," tech. rep., Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002.