2008-04-29

An Axiomatic Semantics for Functional Reactive Programming

Christopher T. King
Worcester Polytechnic Institute

An Axiomatic Semantics for Functional Reactive Programming

by

Christopher T. King

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2008

APPROVED:

Prof. Kathi Fisler, Advisor _____

Prof. Dan Dougherty, Reader _____

Prof. Michael Gennert, Department Head _____

Abstract

Functional reactive programming (FRP) is a paradigm extending functional languages with primitives which operate on state. Typical FRP systems contain many dozens of such primitives. This thesis aims to identify a minimal subset of primitives which captures the same set of behavior as these systems, and to provide an axiomatic semantics for them using first-order linear temporal logic, with the aim of utilizing these semantics in formal verification of FRP programs. Furthermore, we identify several important properties of these primitives and prove that they are satisfied using the Coq proof assistant.

Contents

Chapter 1

Introduction and Related Work

Functional reactive programming (FRP) is a relatively new paradigm encompassing several different kinds of systems, all with the purpose of extending functional languages with primitives which operate on state. FRP exhibits many properties which make it seem to be an ideal target for application of formal verification techniques. This thesis aims to explore the space of formal verification of FRP programs, both identifying exactly *what* these properties are which FRP seems to exhibit, developing a basic framework for FRP which does in fact enjoy these properties, and using this framework to perform formal verification of FRP programs.

In this chapter, we will give an overview of the ideas common to all functional reactive systems, as well as explain the differences between them. Most importantly, we will identify exactly what properties are desirable for a functional reactive system.

1.1 Overview of FRP

In the broadest sense, functional reactive programming (FRP) is a programming paradigm which extends functional programming with a notion of time. Two special datatypes, *event streams* and *behaviors*, are introduced. Event streams are values which are discrete functions of time, whereas behaviors are values which are continuous functions of time. Together they are termed *signals*. The points at which an event stream is defined are termed *events*. Events are said to *occur* on an event stream.

Both types of signals are designed to represent "real-world" phenomena. Some examples of event streams are:

- mouse clicks
- key presses

- timer events
- network packets
- MIDI data

Examples of behaviors include:

- cursor position
- buffer contents
- time of day
- temperature
- audio data

Event streams and behaviors may be manipulated using a set of predefined combinators. Let us illustrate with an example.

1.1.1 Example: Alarm clock

Consider a program implementing a graphical alarm clock. It has the following requirements:

- The current time must be displayed on the screen.
- The user must be able to enter an alarm time in a text box.
- The program must emit a tone when the current time reaches the alarm time.
- The user must be able to silence the tone by clicking a "snooze" button.

Our program has available the following inputs:

- the current time (*curtime*, a behavior)
- the value of the text box (*alarmtime*, a behavior)
- the "snooze" button clicks (*snooze*, an event stream)

We must provide the following outputs:

- graphics to render on the screen (*screen*, a behavior)
- audio to be played on the speaker (*audio*, a behavior)

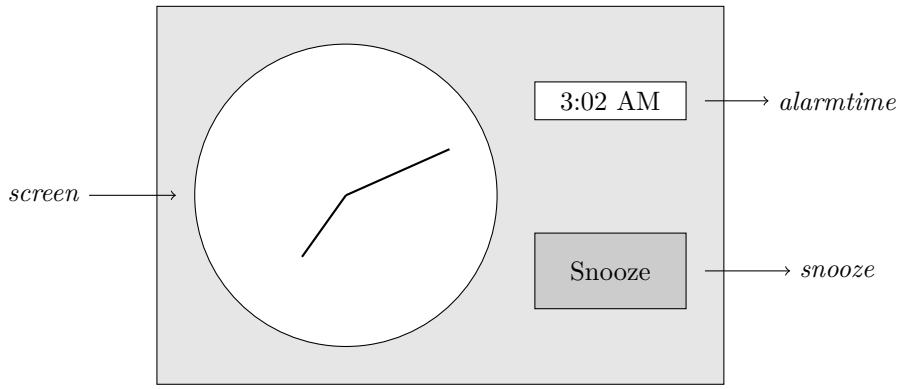

Figure 1.1: Alarm clock GUI

To draw the clock on the screen, we can use the following FRP pseudocode:

$$hour\_hand = \text{line}\left([0,0], \left[\frac{1}{3}\cos\left(-\frac{2\pi}{12}curtime.hours + \frac{\pi}{2}\right), \frac{1}{3}\sin\left(-\frac{2\pi}{12}curtime.hours + \frac{\pi}{2}\right)\right]\right)$$

$$minute\_hand = \text{line}\left([0,0], \left[\frac{2}{3}\cos\left(-\frac{2\pi}{60}curtime.minutes + \frac{\pi}{2}\right), \frac{2}{3}\sin\left(-\frac{2\pi}{60}curtime.minutes + \frac{\pi}{2}\right)\right]\right)$$

$$clock\_face = \text{circle}\left([0,0], 1\right)$$

$$screen = hour\_hand \cup minute\_hand \cup clock\_face$$

Here, the output behavior *screen* is set to be always equal to the rendering of a circle and two lines pointing outward from the origin at angles representing the current time. Unlike in traditional GUI programming, no mutation is needed to update the screen, the FRP system internally manages this in order to ensure that the defined relationship always holds true.

To determine when the alarm goes off, we create the following internal event stream:

$$trigger = \textbf{when}\left(curtime \geq alarmtime\right)$$

The FRP library function **when** creates an event stream which occurs at the instant its argument becomes true. Here, an event will occur on *trigger* at the instant when the current time becomes greater than the set alarm time.

Let us define our output tone and silence behaviors:

$$tone = \sin\left(2\pi \cdot 440 \cdot curtime.seconds\right)$$

$$silence = 0$$

Finally, we must switch between the output tone and silence when *trigger* occurs and when the user presses the "snooze" button:

$$audio = \textbf{switch}\left(silence, (\textbf{on}\ trigger \rightarrow tone) + (\textbf{on}\ snooze \rightarrow silence)\right)$$

The FRP syntax ($\mathbf{on}\, x \rightarrow y$) creates an event stream which occurs whenever x occurs, but has the value y. Here, we create higher-order event streams which carry the behaviors *tone* and *silence* in their events. The two event streams are merged with the + operator, and the FRP library function **switch** creates a behavior which switches between *tone* and *silence* when they occur in the event stream. Initially we start with *silence*. This relational description contrasts with the imperative "push" or "pull"-based approaches typically used in audio interface code, where either audio data is pushed through a queue, or an output buffer is read by the system at regular intervals.

And that is all! Using FRP, we have defined in eight lines of declarative code what would otherwise take dozens of lines using a traditional GUI toolkit in an imperative language. The implementation of our alarm clock remains very close to our specification, making verification nearly trivial.

1.1.2 Monadic and Arrow-based FRP styles

FRP systems generally follow one of two styles, *monadic* or *arrow-based*. The first system to call itself FRP was the Fran [9] language built on top of Haskell. This system follows a monadic approach, meaning that signals are values which can be manipulated as functors (by pointwise transforming their values), and that higher-order signals may be simplified as with monad join. Other FRP systems which follow the monadic approach include FrTime [4], Flapjax [17], and OCaml RT [16].

An alternative FRP approach is to use arrows [15] to represent signal *transformers* rather than simply the signals themselves. Conceptually, this allows both the input and the output of a signal transformer to be manipulated, as opposed to the monadic approach, which allows only the output to be manipulated. Fran's descendant, Yampa [14], is an example of an arrow-based FRP system.

In this paper we will focus on the monadic approach.

1.1.3 Comparison with synchronous dataflow

FRP is most closely related to synchronous dataflow programming. Dataflow programming languages such as Lustre [12] are similarly declarative in nature and have a notion of time. However, time in dataflow languages is fully discrete: there is a unit timestep to which all signals must be quantized. While potentially simplifying the implementation, we lose not only the abstraction of continuous time, but also the opportunity for some useful optimizations, as we will see in section 1.1.4.

One further benefit FRP provides over dataflow programming is the ability to manipulate signals as first-class objects. This is necessary for systems which must dynamically change their structure at runtime. For example, consider a document editor which may have multiple documents open: we would like to reuse the same code to control each document. Because FRP allows us to construct and re-route signals at run time, we can acheive this without resorting to higher-level constructs.

1.1.4 Properties

Following are some properties of FRP systems which are desirable because they abstract parts of the system which otherwise would be implementation-dependent, and aid formal verification proofs of FRP programs.

Timestep irrelevance Because computers operate in discrete steps, any interpreter for an FRP program must (short of mathematical analysis) discretize time: behaviors cannot *truly* change continuously, nor can events *truly* last for an infinitesimally short period of time. So the choice arises, in implementation, as to what duration these discrete timesteps should be.

But this choice of timestep duration is just that: an implementation detail. Ideally, we would like to prevent the construction of a program which can observe either the existence or duration of these discrete timesteps, thus maintaining the illusion of idealized time. This property is referred to as *timestep irrelevance*.

Time invariance A further desirable property of FRP programs is that they be componental: that is, any analysis of an FRP program remains valid when it is included as a component in a larger program. Timestep irrelevance partially guarantees this: two FRP programs composed in parallel (meaning, their execution starts at the same time) will behave as if they were executed separately, regardless of whether such composition affects the system's timestep.

To extend this property to FRP programs composed sequentially (meaning, their execution starts at different times), we would like also to require of FRP programs the property that the behavior of an FRP program remains locally the same regardless of at what global time it was invoked. This property is referred to as *time invariance*.

1.2 Verification

Verifiability, either via proof methods, or model checking, is a desirable property of any programming language. FRP is no exception. In particular, verification of functional reactive programs is desirable because exisiting verification techniques in the domains for which FRP is well-suited (those involving input/output and state) are difficult to use, if at all existent. Interactive systems, when written in traditional imperative languages, simply have too much state and parallelism for verification to be made tractable.

Consider the example of the graphical alarm clock from section 1.1.1. If written in a language such as Java using the AWT or Swing GUI toolkit, this program would likely utilize callbacks (asynchronous externally generated function calls) to process events generated by the system, and update the display and audio using imperative constructs running in separate threads. The control flow in such a system is hectic! Verifying a simple local property such as "the clock hands always rotate clockwise" becomes non-trivial as we are forced to consider non-local effects due to the threading and callbacks. It would seem that

we could more easily reason about the near-algebraic equations used in the FRP implementation of this program.

To the author's knowledge, there has been no work done to date on formal verification of FRP programs. Formal denotational and operational semantics have been given for various systems (including Yampa [5], and FrTime [3]), but none have been specified axiomatically (which would further bridge the verification gap by abstracting parts of the system, such as dependency relations). However, there has been work done on formal verification of synchronous dataflow languages: in particular, both a model checker [13] and a proof obligation generator [7] have been developed for Lustre. The similarity between synchronous dataflow and FRP would seem to indicate that the same is possible for FRP.

1.3 Goals

The main goal of this thesis is to identify a minimal subset of primitives which captures the same set of behavior as the above-mentioned FRP systems, with the end result of creating a system which caters to formal verification. We will define the primitives by means of an axiomatic semantics in first-order linear temporal logic. Such a specification has the advantage that it neither requires nor favors any particular implementation, while still providing an adequate framework for formal verification. These primitives will be presented in chapter 2.

An example of the style of verification we would like to perform is as follows. Consider the alarm clock example from section 1.1.1, and the property from section 1.2: "the clock hands always rotate clockwise". Assuming we have access to an angle_of() function which returns the angle of a line object, we can specify this property formally in first-order linear temporal logic (see section 2.1) as:

$$\Box\left((\text{angle\_of}(\mathit{hour\_hand}) - \bigcirc\text{angle\_of}(\mathit{hour\_hand}) \mod 2\pi) > \pi\right)$$

where the notation $n \mod m$ denotes the smallest nonnegative value r such that $km + r = n$, k being an integer. The proof of this property then follows immediately from elementary algebra, the fact that *curtime* is monotonically increasing, and the assumption that *curtime* is quantized to small enough intervals such that the hands need never jump 180° or more. (Quantization of inputs is a necessary evil in the discrete world of computers, as we shall see.)

To further bolster use of these primitives in formal verification, we will show that they satisfy the properties of timestep irrelevance and time invariance through formal proofs built using the Coq proof assistant. We will also show embeddings of the two FRP systems FrTime and Yampa in our system to support our claim of completeness. These proofs and embeddings will be presented in chapter 3.

Another goal of this thesis had been to develop a verification framework for FRP programs using the proposed primitives. Several attempts were made at this, but none led to a workable system. These attempts, as well as other ideas for future work, are documented in the conclusion of the paper in chapter 4.

Chapter 2

Syntax and Semantics

In this chapter we first give an overview of the formalisms used in the rest of the document. We then present the formal syntax and semantics of the proposed minimal set of FRP primitives.

2.1 First-order Linear Temporal Logic

To specify the semantics of the FRP primitives, we will utilize the first-order linear temporal logic [10] (FOLTL). FOLTL extends first-order logic with a notion of time: propositions may have different truth values at different times (or states). More formally, propositions, predicates, and functions are extended to be functions of time by implicitly parameterizing them over the natural numbers (each denoting a distinct state). Furthermore, the following operators are added:

Next $\bigcirc P$ means that "P is true in the next state."

Globally $\Box P$ means that "P is true in this state and in all following."

Finally $\Diamond P$ means that "P is true in at least one state now or in the future."

Until $P \,\mathcal{U}\, Q$ means that "P is true up to the state in which Q is true."

Releases $P \,\mathcal{R}\, Q$ means that "Q is true up to and including the state in which P is true, if any."

In the denotational semantics given in table 2.1, i, j, and k are natural numbers, and the notation $[P]_i$ is a first-order logic statement denoting the truth value of the FOLTL formula P at time i. Note that $\bigcirc x$ may actually apply to any term, denoting the value of that term as interpreted in the next state. Additionally, we define the "weak until" operator $P \,\mathcal{W}\, Q \equiv (P \,\mathcal{U}\, Q) \vee \Box P$, with the meaning that "P is true up to the state in which Q is true, if any."

It is also important to note the notion of *local time*. Because functions in FOLTL are implicitly functions of time, when we define their semantics using

$$
\begin{aligned}
[\neg P]_i &\iff \neg [P]_i \\
[P \wedge Q]_i &\iff [P]_i \wedge [Q]_i \\
[P \vee Q]_i &\iff [P]_i \vee [Q]_i \\
[P \rightarrow Q]_i &\iff [P]_i \rightarrow [Q]_i \\
[\bigcirc x]_i &\iff [x]_{i+1} \\
[\Box P]_i &\iff \forall j, [P]_{i+j} \\
[\Diamond P]_i &\iff \exists j, [P]_{i+j} \\
[P \,\mathcal{U}\, Q]_i &\iff \exists j, [Q]_{i+j} \wedge \left(\forall k, k < j \rightarrow [P]_{i+k}\right) \\
[P \,\mathcal{R}\, Q]_i &\iff \forall j, [Q]_{i+j} \vee \left(\exists k, k < j \wedge [P]_{i+k}\right) \\
[\forall x, P(x)]_i &\iff \forall x, [P(x)]_i \\
[\exists x, P(x)]_i &\iff \exists x, [P(x)]_i
\end{aligned}
$$

Table 2.1: Denotational semantics of FOLTL

FOLTL, the specifications will be relative to the time frame in which the functions are invoked. This time frame will be referenced as local time 0. More formally, for any function f:

$$\frac{f(x) : \{y | P(y)\}}{[f(x) = y]_i \iff [P(y)]_i}$$

2.2 Monads

A *monad* is a structure consisting of a datatype $M[\alpha]$ parameterized over a type α and three associated functions, $unit : \alpha \rightarrow M[\alpha]$, $map : (\alpha \rightarrow \beta) \rightarrow M[\alpha] \rightarrow M[\beta]$, and $join : M[M[\alpha]] \rightarrow M[\alpha]$. Many data types are monads, for example, lists or the "option" type (a type with an exceptional value). Monads follow several laws which allow for algebraic simplification of expressions built using them: by ensuring that our FRP primitives form monads, we will be able to utilize these reductions in formal proofs.

Since monads have been covered in much detail elsewhere, we will instead briefly summarize the laws governing a monad [18] in table 2.2.

A monad may also be characterized by a function $return : \alpha \rightarrow M[\alpha]$ (equivalent to *unit*) and a function $bind : M[\alpha] \rightarrow (\alpha \rightarrow M[\beta]) \rightarrow M[\beta]$ (equivalent to $join \cdot map$). These two equivalent characterizations will be used interchangeably in the rest of this document.

(i)	$map(\lambda x : \alpha.x) = \lambda x : M[\alpha].x$
(ii)	$map(g \cdot f) = map(g) \cdot map(f)$
(iii)	$map(f) \cdot unit = unit \cdot f$
(iv)	$map(f) \cdot join = join \cdot map(map(f))$
(v)	$join \cdot unit = \lambda x : M[\alpha].x$
(vi)	$join \cdot map(unit) = \lambda x : M[\alpha].x$
(vii)	$join \cdot join = join \cdot map(join)$

where $g \cdot f$ is function composition ($g \cdot f \equiv \lambda x.g(f(x))$).

Table 2.2: Monad laws

2.3 Primitives

This section will describe the twelve FRP primitives and their semantics. We assume the existence of a language $\mathcal{L}$ reducible to the typed lambda-calculus which is extended with the following primitives to generate a new language $\mathcal{L}^+$. We make the further assumptions that time is discretizable (see section 1.1.4).

The semantics of each primitive are defined by FOLTL formulae. The semantics of a functional reactive program $P : \mathcal{L}^+$ at time t are then given by the interpretation of that program with all primitives replaced with their semantic meaning at time t. The bodies of the primitives are referenced to the time frame in which the primitive functions are invoked.

For readers familiar with the FrTime or Yampa systems, analogies of each primitive to ones present in those systems will be made where appropriate. A full formal mapping however will be provided in section 3.5.

2.3.1 Data types

$\mathcal{L}^+$ adds to $\mathcal{L}$ two new type constructors, $E[\alpha]$ and $B[\alpha]$, corresponding to event streams and behaviors over type α, respectively. For the purposes of defining semantics, we assume the existence of two functions, $\epsilon : E[\alpha] \to \alpha \cup \{\bot\}$ and $\beta : B[\alpha] \to \alpha$, which map event streams and behaviors, respectively, to their value (or $\bot$, in the case of an event non-occurrence) as interpreted in the present FOLTL time frame (which can be considered an implicit argument to all functions). However it is important to note that these two functions are *not* available as primitives themselves.

2.3.2 Event stream functions

Six primitives work only with event streams: $\texttt{return}_e$, $\texttt{map}_e$, $\texttt{join}_e$, $\texttt{zero}_e$, $\texttt{plus}_e$, and $\texttt{fix}_e$. The first five of these together with the event stream type constructor define a monad (see Appendix A).

Unit event $\texttt{return}_e(x)$ is an event stream with exactly one event occurrence, at local time 0, with value x:

$$\texttt{return}_e(x : \alpha) : \{e : E[\alpha] | \epsilon(e) = x \wedge \bigcirc\Box\,(\epsilon(e) = \bot)\}$$

$\texttt{return}_e$ is equivalent to Yampa's `now` function.

Event mapping $\texttt{map}_e(f)(e_1)$ is an event stream which is synchronous with the event stream e_1, but carries values obtained by applying f pointwise to the values carried by e_1:

$$\texttt{map}_e(f : \alpha \to \beta)(e_1 : E[\alpha]) : \{e : E[\beta] | \Box\,(\epsilon(e) = f'(\epsilon(e_1)))\}$$

where $f'(\bot) = \bot$ and $f'(x) = x$ otherwise. $\texttt{map}_e$ is analagous to Yampa's `fmap` function, and equivalent to FrTime's `map-e` function.

Zero event $\texttt{zero}_e$ is the event stream with no event occurrences:

$$\texttt{zero}_e : \{e : E[\alpha] | \Box\,(\epsilon(e) = \bot)\}$$

It is equivalent to Yampa's `never` value, or to an event receiver to which events are never sent in FrTime.

Event merging $\texttt{plus}_e(e_1)(e_2)$ merges the event streams e_1 and e_2, with e_1 taking precedence in the case of simultaneous events:

$$\texttt{plus}_e(e_1 : E[\alpha])(e_2 : E[\alpha]) : \{e : E[\alpha] | \Box\,(\epsilon(e) = \epsilon(e_1) \oplus \epsilon(e_2))\}$$

where $\bot \oplus y = y$ and $x \oplus y = x$ otherwise. It is analogous to Yampa's `lMerge` function, and equivalent to FrTime's `merge-e` function.

Event switching $\texttt{join}_e$ allows management of higher-order event streams. $\texttt{join}_e(ee)$ initially behaves as $\texttt{zero}_e$, but whenever an event carrying an event stream occurs on ee, $\texttt{join}_e(ee)$ behaves as that event stream until another such event occurs:

$$\begin{aligned}\texttt{join}_e(ee : E[E[\alpha]]) : \{e : E[\alpha] \,|\, &((\epsilon(e) = \bot)\ \mathcal{W}\ (\epsilon(ee) \neq \bot)) \wedge \\ &(\forall e_1 : E[\alpha], \Box\,((\epsilon(ee) = e_1) \to \\ &\bigcirc\,(\epsilon(ee) \neq \bot)\ \mathcal{R}\ (\epsilon(e) = \epsilon(e_1))))\}\end{aligned}$$

$\texttt{join}_e(ee)$ is analogous to Yampa's `pSwitch` function.

Event fixpoint $\texttt{fix}_e(f)$ computes the event stream fixpoint of the function f:

$$\texttt{fix}_e(f : E[\alpha] \to E[\alpha]) : \{e : E[\alpha] | e = f(e)\}$$

Note that $\texttt{fix}_e$ is *not* a monadic fixpoint operator (which would have type $(\alpha \to E[\alpha]) \to E[\alpha]$). It is analogous to Yampa's `loop` function.

2.3.3 Behavior functions

Four primitives work only with behaviors: $\texttt{return}_b$, $\texttt{map}_b$, $\texttt{join}_b$, and $\texttt{fix}_b$. The first three of these together with the behavior type constructor define a monad (see section 3.2).

Unit behavior $\texttt{return}_b(x)$ is a behavior which always has value x:

$$\texttt{return}_b(x : \alpha) : \{b : B[\alpha] | \Box\,(\beta(b) = x)\}$$

$\texttt{return}_b$ is equivalent to the Yampa's `constant` function.

Behavior mapping $\texttt{map}_b(f)(b)$ is a behavior whose value is always f applied to the current value of the behavior b:

$$\texttt{map}_b(f : \alpha \to \beta)(b_1 : B[\alpha]) : \{b : B[\beta] | \Box\,(\beta(b) = f(\beta(b_1))\}$$

$\texttt{map}_b$ is equivalent to the Yampa's `arr`[1] function.

Behavior switching $\texttt{join}_b$ allows management of higher-order behaviors. $\texttt{join}_b(bb)$ behaves as whichever behavior is currently carried by bb:

$$\texttt{join}_b(bb : B[B[\alpha]]) : \{b : B[\alpha] | b = \beta(bb)\}$$

Behavior fixpoint $\texttt{fix}_b(f)$ computes the behavior fixpoint of the function f:

$$\texttt{fix}_b(f : B[\alpha] \to B[\alpha]) : \{b : B[\alpha] | b = f(b)\}$$

Like $\texttt{fix}_e$, $\texttt{fix}_b$ is *not* a monadic fixpoint operator. It also is analogous to Yampa's `loop` function.

2.3.4 Conversion functions

Two primitives are required to link event streams and behaviors: `e2b` and `b2e`. Despite their names, they are not inverses of each other (although they are jointly idempotent). Together, they allow for state to be stored and recalled.

Events to behaviors `e2b` allows storage of state in the form of a behavior. $\texttt{e2b}(i)(e)$ is a behavior which initially has the value i until an event occurs on the event stream e, immediately *after* which $\texttt{b2e}(i)(e)$ takes on the value of the event, until another such event occurs:

$$\begin{aligned}\texttt{e2b}(i : \alpha)(e : E[\alpha]) : \{b : B[\alpha] |\, & ((\epsilon(e) \neq \bot)\ \mathcal{R}\ (\beta(b) = i)) \wedge \\ & (\forall x : \alpha, \Box\,((\epsilon(e) = x) \to \\ & \quad \bigcirc\,((\epsilon(e) \neq \bot)\ \mathcal{R}\ (\beta(b) = x))))\}\end{aligned}$$

`e2b` is equivalent to Yampa's `hold` function when coupled with the unit delay operator `pre`.

[1] Shiver me timbers!

Behaviors to events `b2e` allows retrieval of state from a behavior. `b2e`(b) is an event stream with exactly one event occurrence, at local time 0, which carries the value the behavior b has at that time:

$$\texttt{b2e}(b : B[\alpha]) : \{e : E[\alpha] | \, (\epsilon(e) = \beta(b)) \wedge \bigcirc\Box \, (\epsilon(e) = \bot)\}$$

`b2e` is analogous to Yampa's `tag` function, or to FrTime's `snapshot` syntax.

Chapter 3

Proofs of properties

In order to prove that the monad laws, timestep irrelevance, and time invariance hold true for the formulation of FRP given in chapter 2, the proof assistant Coq [6] was used. This chapter details how the proofs were constructed in Coq.

3.1 Coq

Coq is based on a typed constructive logic called the Predicative Calculus of Inductive Constructions (pCIC). Through pCIC, Coq provides first-class types (using an infinite type hierarchy), universal quantification, λ-abstractions, and function application. Terms (constants and functions) can be built from other terms using these constructs, defined (co)inductively, taken as axioms, or constructed using a proof. Proofs in Coq are constructed interactively using *backward reasoning*. The state of an interactive proof consists of a set of hypotheses and a set of goals, which are manipulated using *tactics*, until no goals remain, at which point the proof is complete.

To embed FOLTL and FRP, we use the following scheme.

Types The type of a logic proposition in Coq, Prop, is extended by adding a parameter of type *nat* (natural number): $nat \to \mathsf{Prop}$.

Logic connectives All logic connectives appearing in table 2.1 are utilized by treating the notation $[P]_i$ as function application: any rule $[P(x)]_i \iff Q\left([x]_{f(i)}\right)$ can be translated as:

$\mathsf{Definition}\, P(x : nat \to \mathsf{Prop}) : nat \to \mathsf{Prop} := \mathsf{fun}\,(i : nat) \Rightarrow Q\,(x\,(f\,i))$.

Temporal connectives All temporal connectives are treated the same as the logic connectives, and are defined for convenience in the `LTL` module (see appendix A.1).

FRP types The two FRP type constructors, $E[\alpha]$ and $B[\alpha]$, and their associated destructor functions $\epsilon : E[\alpha] \to \alpha \cup \{\bot\}$ and $\beta : B[\alpha] \to \alpha$ are defined as axioms. The built-in $option\,\alpha$ type is used to represent $\alpha \cup \{\bot\}$.

FRP functions All FRP functions appearing in section 2.3 of the form are translated as a pair of axioms: one asserting the existence of the function (with the addition of an extra argument of type *nat* denoting the local time), and one defining the semantics of the function. These definitions are contained in the `FRP` module (see appendix A.2).

3.2 Monad laws

To prove that events and behaviors as defined in section 2.3 are indeed monads, we show that they obey all monad laws. The proofs were performed using Coq. Because the monad laws involve equality of functions, but Coq does not provide existential equality, proofs were instead performed on the laws applied to arbitrary values (so e. g. $map(g \cdot f) = map(g) \cdot map(f)$ became $\forall e, map(g \cdot f)(e) = (map(g) \cdot map(f))(e)$. Furthermore, because event streams and behaviors were defined axiomatically, each needed an equality predicate defined: for events, this was taken to be $e_1 \overset{e}{=} e_2 \equiv \Box\,(\epsilon(e_1) = \epsilon(e_2))$, for behaviors, $b_1 \overset{b}{=} b_2 \equiv \Box\,(\beta(b_1) = \beta(b_2))$.

The proofs are largely rewrite-based, since the monad laws involve equalities, as do the semantic definitions. For example, the semantics of $\texttt{return}_e$ state that $\epsilon(e) = x$, where $e = \texttt{return}_e(x)$. This is translated into Coq as $eps\,(return\_e\,i\,x)\,i = Some\,x$ (where the i is a time index), which then serves as a rewrite rule. Another rewrite rule exists for $\texttt{return}_e$ which, when simplified, states $eps\,(return\_e\,i\,x)\,(i+j+1) = None$. Repeated application of these rewrite rules (which is automated by the *rewrite_e* and *rewrite_b* tactic macros) results in a great deal of simplification of the monad law equations.

The remaining proof steps to be performed consist mostly of case-based reasoning. For example, proofs involving $\texttt{return}_e$ must be broken into two cases, the time at which $\texttt{return}_e$ is invoked, and all times afterward. Proofs involving $\texttt{map}_e$ often require two cases, one corresponding to the case of an event occurrence at a given time, and another corresponding to an event non-occurrence.

An interesting case arises when working with proofs involving $\texttt{join}_e$. It becomes necessary in these proofs to know the first event which occurs on an event stream, if such an event exists. The easiest way to prove that finding such an event was possible turned out to be constructing a recursive program which found an returned the event. The program was written using the Russell language, which allows programs with specifications to be written in Coq, generating proof obligations where necessary. The code and proofs can be seen in section A.3.

The monad law proofs themselves can be found in appendix A.4. Only monad laws *i–iii* and *v* are currently proven for events, however all monad laws are proven for behaviors.

3.3 Timestep irrelevance

To prove that the FRP primitives satisfy the propert of timestep irrelevance as outlined in section 1.1.4, if we assume that time is never discretized so as to cause events occuring at distinct times to appear to occur simultaneously, it is sufficient that we assert the unobservability of the existence of discrete timesteps by requiring the following: there cannot exist any FRP program, which, when evaluated on a set of input signals I, produces a different output during the intervals $[0, i)$ and $[j, \infty)$ than when evaluated on the set of input signals which differs from I only in that a behavior $b \in I$ has been held constant on the interval $[i, j)$[1] during which no events occur in I except at time i.

Similarly, to assert the unobservability of the duration of the discrete timesteps, it is sufficient for us to require that: there does not exist any FRP program, which, when evaluated on a set of input signals I, produces a different output than when evaluated on the set of input signals which differs from I only in that the duration of all events $e \in I$ occurring at time i have been extended in duration to occur over the interval $[i, j)$, during which no other events occur and all behaviors are constant.

Formal proofs of timestep irrelevance were not performed. To carry out such proofs in Coq would require embedding a formal grammar and interpreter for FRP (to allow us to show that no such program can be constructed) and is outside the scope of this thesis.

3.4 Time invariance

To prove that the FRP primitives satisfy time invariance as described in section 1.1.4, it suffices to show that: if an FRP program P produces the set of output signals O when evaluated on a set of input signals I, the program P^i produces the set of output signals O^i when evaluated on the set of input signals I^i, where the notation x^i means "shifted to local time i".

The proofs of time invariance proceed similarly to those of the monad laws, with the addition of two auxiliary functions `shift_e` and `shift_b` which shift an event stream or behavior in time. The proofs themselves can be found in appendix A.5.

3.5 Embedding of other FRP systems

To support my claim that my proposed combinators support FRP in the same style as previous systems, I will show an embedding of all core functions in the monadic FRP system FrTime [1], and the arrow-based FRP system Yampa . The embeddings are complete excluding those operators which would violate either timestep irrelevance or time invariance.

[1] Meaning, $\forall x \in (i, j)$, the value of b at time x equals the value of b at time i.

For conciseness, the monadic *do notation* is used to extend the language of FRP expressions. The sequence $x \overset{e}{\leftarrow} y$; *expr* is translated as $\texttt{join}_e\,(\texttt{map}_e\,(\lambda x.expr)\,(y))$, where *expr* may be either a simple expression or itself a do notation expression.

Additionally, the notation $x+y$ is shorthand for $\texttt{plus}_e(x)(y)$, and the function $\texttt{once}_e(e)$, which returns an event stream containing only the first event occuring on e (if any), is defined by the following pseudocode.

Algorithm 1 $\texttt{once}_e(e)$

```
{ remember whether an event has occured }
let h = e2b(true) (map_e (λx.false) (e))
x ←e e;
c ←e b2e(h);  { c indicates whether x is the first event }
if c then
  return_e(x)
else
  zero_e
end if
```

3.5.1 FrTime

`Type constructors` The `event[a]` type constructor corresponds directly to $E[\texttt{a}]$. The `behavior[a]` constructor corresponds to the type $E[\texttt{a}] \times B[\texttt{a}]$. The purpose of the event stream in this tuple is to represent discontinuous changes in the behavior. The notation x_e refers to the event stream component of this tuple, and x_b refers to the behavior component.

`value-now` The `value-now` operator is not directly encodable using the FRP primitives, because it allows for values to escape the monad. However, the usage `(let ((x (value-now b))) ...)` in the context of an event stream function would correspond to $\texttt{x} \overset{e}{\leftarrow} \texttt{b2e(b)}; ...$, and in the context of a behavior function would correspond to $\texttt{x} \overset{b}{\leftarrow} \texttt{b}; ....$

`delay-by` Implementation of the `delay-by` operator for event streams requires the existence of a function $\texttt{tick}(t : \mathbb{R}) : E[\mathsf{unit}]$ which creates an event stream on which an event occurs after t seconds, but is otherwise straightforward (although outside the scope of this embedding).

The `delay-by` operator for behaviors cannot however be directly encoded. The issue lies with our ideal abstraction of FRP: because behaviors are continuous functions, we would need an infinite amount of storage to store the value of the behavior during the delay interval. However, if we are content to allow our behaviors to be approximated by their accompanying event stream (almost always reasonable) then the problem reduces to that for event streams.

`integral` and `derivative` Both `integral` and `derivative`, like `delay-by`, require external functions dealing with physical time, but are otherwise straightforward and outside the scope of this embedding.

`map-e` The function `(map-e fb e)` can be simulated by the following pseudocode.

Algorithm 2 `(map-e fb e)`

$x \overset{e}{\leftarrow} e$;
$f \overset{e}{\leftarrow} \texttt{once}_e\,(\texttt{fb}_e + \texttt{b2e}(\texttt{fb}_b))$;
$\texttt{return}_e(f(x))$

`filter-e` The function `(filter-e fb e)` can be simulated by the following pseudocode.

Algorithm 3 `(filter-e fb e)`

$x \overset{e}{\leftarrow} \texttt{e}$;
$f \overset{e}{\leftarrow} \texttt{once}_e\,(\texttt{fb}_e + \texttt{b2e}(\texttt{fb}_b))$;
if $f(x)$ **then**
  $\texttt{return}_e(x)$
else
  $\texttt{zero}_e$
end if

`merge-e` The function `(merge-e)` corresponds directly to $\texttt{zero}_e$, and `(merge-e e1 ...)` corresponds directly to $\texttt{plus}_e(\texttt{e1})\,(\texttt{(merge-e ...)})$.

`once-e` The function `(once-e e)` corresponds directly to $\texttt{once}_e(\texttt{e})$.

`changes` The function `(changes b)` simply returns $\texttt{b}_e$.

`hold` The function `(hold e i)` returns the tuple $(\texttt{e}, \texttt{e2b}(\texttt{i})(\texttt{e}))$.

`switch` The function `(switch be i)` can be simulated by the following pseudocode.

Algorithm 4 `(switch be i)`

{ switch the change events }
let $e = \texttt{join}_e\,(\texttt{map}_e\,(\lambda b.b_e)\,(\texttt{be}) + \texttt{return}_e(\texttt{i}_e))$
{ switch the behaviors }
let $b = \texttt{join}_b\,(\texttt{e2b}(\texttt{i}_b)(\texttt{map}_e\,(\lambda b.b_b)\,(\texttt{be})))$
{ insert change events for switches }
let $e' = \texttt{join}_e\,(\texttt{map}_e(\texttt{b2e})(\texttt{be}))$
$(e + e', b)$

`accum-e` The function `(accum-e fe i)` can be simulated by the following pseudocode.

Algorithm 5 `(accum-e fe i)`

{ remember last event occurence }
let $b = \texttt{e2b}(\texttt{i})(e)$
$f \overset{e}{\leftarrow} \texttt{fe}$;
$x \overset{e}{\leftarrow} \texttt{b2e}(b)$;
$\texttt{return}_e(f(x))$

where e is the event stream output from this code, as obtained using the $\texttt{fix}_e$ function.

`accum-b` The function `(accum-b fe i)` is simply the tuple $(e, \texttt{e2b}(\texttt{i}, e))$, where e is `(accum-e fe i)`.

`collect-e` The function `(collect-e e i f)` can be simulated by the following pseudocode.

Algorithm 6 `(collect-e e i f)`

{ remember last event occurence }
let $b = \texttt{e2b}(\texttt{i})(e')$
$x \overset{e}{\leftarrow} \texttt{e}$;
$y \overset{e}{\leftarrow} \texttt{b2e}(b)$;
$\texttt{return}_e(\texttt{f}(x, y))$

where e' is the event stream output from this code, as obtained using the $\texttt{fix}_e$ function.

`collect-b` The function `(collect-b e i f)` is simply the tuple $(e, \texttt{e2b}(\texttt{i}, e))$, where e is `(collect-e e i f)`.

`when-e` The function `(when-e b)` can be simulated by the following pseudocode.

Algorithm 7 `(when-e b)`

$x \overset{e}{\leftarrow} \texttt{b}_e$;
if x **then**
  $\texttt{return}_e(x)$
else
  $\texttt{zero}_e$
end if

Lifted procedures Procedures applied to behaviors can be simulated using the $\texttt{map}_e$ and $\texttt{map}_b$ functions. For example, `(f b)`, where `f` is an unlifted function and `b` is a behavior, is simply the tuple $(\texttt{map}_e(\texttt{f})(\texttt{b}_e), \texttt{map}_b(\texttt{f})(\texttt{b}_b))$.

`if` The syntax `(if cb tb eb)` can be simulated by the following pseudocode.

Algorithm 8 `(if cb tb eb)`

{ select the change events }
let $e = \left(c \overset{e}{\leftarrow} \texttt{cb}_e; c\,?\ (\texttt{tb}_e + \texttt{b2e}(\texttt{tb}_b)) : (\texttt{eb}_e + \texttt{b2e}(\texttt{eb}_b))\right)$
{ select the behavior }
let $b = \left(c \overset{b}{\leftarrow} \texttt{cb}_b; c\,?\,\texttt{tb}_b : \texttt{eb}_b\right)$
(e, b)
where the syntax $x\,?\,y : z$ is shorthand for "if x then y else z".

`snapshot` Like `value-now`, `snapshot` is not directly encodable using the FRP primitives. However, the usage `(snapshot (b) ...)` in the context of an event stream function would correspond to $\texttt{b} \overset{e}{\leftarrow} \texttt{b2e}(\texttt{b}); ...$, and in the context of a behavior function would correspond to $\texttt{b} \overset{b}{\leftarrow} \texttt{b}; ...$.

3.5.2 Yampa

Because in Yampa, event streams are simply a subtype of behaviors, the property of timestep irrelevance does not hold: an event stream, when treated as a behavior, can be used to determine the length of a timestep. However we can still approximate Yampa's semantics if we treat event streams specially.

Type constructors To simulate Yampa's arrow-based design, we can use functions over signals. The exact type mappings depend on the expression, but in general, the signal function type `SF (Event a) (Event b)` corresponds to $E[\texttt{a}] \to E[\texttt{b}]$, `SF a b` corresponds to $E[\texttt{a}] \times B[\texttt{a}] \to E[\texttt{b}] \times B[\texttt{b}]$ (the meaning of the tuples being as described in section 3.5.1), etc. The `Event a` type constructor, when used alone, corresponds to the type $\texttt{a} \cup \{\bot\}$.

`arr` The function `arr f` is simulated by the expression $(\lambda b.\,(\texttt{map}_e(\texttt{f})(b_e), \texttt{map}_b(\texttt{f})(b_b)))$ when applied to behaviors. The corresponding construct for event streams, `arr (fmap f)`, is simply $\texttt{map}_e(\texttt{f})$.

`loop` The function `loop a` is simulated by the expression $(\lambda e.\texttt{a}\ (e, \texttt{fix}_e\,(\lambda r.\,\{\texttt{a}\,(e,r)\}_2)))$ for event streams, where the notation $\{t\}_i$ denotes the ith component of the tuple t. For behaviors, the same expression appies, with $\texttt{fix}_e$ replaced by $\texttt{fix}_{eb}$, defined as:

$$\begin{aligned}\texttt{fix}_{eb}(f) \equiv &\texttt{fix}_e\left(\lambda e.\,\{f\,(e, \texttt{fix}_b\,(\lambda b.\,\{f(e,b)\}_2))\}_1\right),\\ &\texttt{fix}_b\left(\lambda b.\,\{f\,(b, \texttt{fix}_e\,(\lambda e.\,\{f(e,b)\}_1))\}_2\right)\end{aligned}$$

Arrow instance methods Because we are simulating signal functions using functions, the other arrow instance methods are trivially defined and can be summarized in the following table:

`a >>> b`	$(\lambda s.\texttt{b}\,(\texttt{a}\,s))$
`a <<< b`	$(\lambda s.\texttt{a}\,(\texttt{b}\,s))$
`first a`	$(\lambda\,(s_1, s_2)\,.\,(\texttt{a}\,s_1, s_2))$
`second a`	$(\lambda\,(s_1, s_2)\,.\,(s_1, \texttt{a}\,s_2))$
`a *** b`	$(\lambda\,(s_1, s_2)\,.\,(\texttt{a}\,s_1, \texttt{b}\,s_2))$
`a &&& b`	$(\lambda s.\,(\texttt{a}\,s, \texttt{b}\,s))$
`returnA`	$(\lambda s.s)$

Basic signal functions The constant `identity` is simply the identity function, $(\lambda s.s)$, and the function `constant x` is simply $(\lambda s.\texttt{return}_b(\texttt{x}))$.

Initialization The initalization functions make no sense in this embedding, since they can introduce pointwise discrepancies in behaviors.

`never` The constant `never` is simply $(\lambda s.\texttt{zero}_e)$.

`now` The function `now x` is simply $(\lambda s.\texttt{return}_e(\texttt{x}))$.

`edgeBy` The function `edgeBy f i` can be simulated by the following pseudocode.

Algorithm 9 `edgeBy f i`

let b be the signal function input
let $e = \texttt{return}_e(\texttt{i})$
$x \overset{e}{\leftarrow} b_e$;
$y \overset{e}{\leftarrow} e + \texttt{b2e}(b_b)$;
if $\texttt{f}(x) \neq \bot$ **then**
  $\texttt{return}_e(\texttt{f}(x))$
else
  $\texttt{zero}_e$
end if

All other edge functions are defined in Yampa in terms of `edgeBy`.

`notYet` The constant `notYet` can be simulated by the expression

$$(\lambda e.\texttt{join}_e\,(\texttt{return}_e\,(\texttt{zero}_e) + \texttt{map}_e\,(\texttt{return}_e)\,(e)))$$

`takeEvents` The function `takeEvents n` can be simulated by the following pseudocode.

Algorithm 10 `takeEvents n`

let e be the signal function input
let $i = \texttt{fix}_b\left(\lambda i.\texttt{e2b}(0)\left(x \overset{e}{\leftarrow} e; i' \overset{e}{\leftarrow} \texttt{b2e}(i); i' + 1\right)\right)$
$x \overset{e}{\leftarrow} e$;
$c \overset{e}{\leftarrow} \texttt{b2e}(i)$; { c gives count of previous events }
if $c < \texttt{n}$ **then**
  $\texttt{return}_e(x)$
else
  $\texttt{zero}_e$
end if

The constant `once` is defined in Yampa in terms of `takeEvents`. The function `dropEvents` is defined similarly to `takeEvents`.

`switch` The function `switch i k` can be simulated for event streams by the following pseudocode.

Algorithm 11 `switch i k`

let s be the signal function input
let $(e, se) = \texttt{i}(s)$
$\texttt{join}_e\,(\texttt{map}_e(\texttt{k})\,(\texttt{once}_e(se)) + \texttt{return}_e(e))$

The algorithm for behavior inputs is similar (see section 3.5.1 for an example).

`rSwitch` The function `rSwitch i` can be simulated by the expression

$$(\lambda(s, sfe).\texttt{join}_e\,(\texttt{map}_e\,(\lambda sf.sf\;s)\,(sfe) + \texttt{return}_e(sf\;\texttt{i})))$$

for event stream inputs. The algorithm for behavior inputs is similar.

`dSwitch` and `drSwitch` The delayed switch functions, when applied to behaviors, are not representable in this system, because they would violate the property of timestep irrelevance. When applied to functions, their definitions are similar to those of `switch` and `rSwitch`, with the addition of a behavior used as a memory of what the previous switched-in signal is.

`kSwitch` and `dkSwitch` Here our system for representing signal functions breaks down. Because it is not possible to "stop" and "restart" a signal by means other than through `join`, we cannot provide the arbitrary signal storage allowed by the continuations provided in `kSwitch` and `dkSwitch`.

Collections Given a collection structure, embedding Yampa's signal function collection functions should be possible. However this requires a great deal of code beyond the scope of this thesis.

`hold` The function `hold i` can be simulated by the expression $(\lambda e.\,(e, \texttt{e2b}(\texttt{i})(e)))$.

`trackAndHold` The function `trackAndHold i` can be simulated by the following pseudocode.

Algorithm 12 `trackAndHold i`

Let b be the signal function input. The event stream component of this function is given by:

$x \overset{e}{\leftarrow} b_e$;
if $x \neq \perp$ **then**
  $\texttt{return}_e(x)$
else
  $\texttt{zero}_e$
end if

The behavior component of this function is $\texttt{join}_b\,(\texttt{e2b}(\texttt{return}_b(\texttt{i}))(be))$, where be is given by:

$x \overset{e}{\leftarrow} b_e$;
if $x \neq \perp$ **then**
  $\texttt{return}_e(b_b)$
else
  $y \overset{e}{\leftarrow} \texttt{b2e}(b_b)$ { get the old value of b_b }
  $\texttt{return}_e(\texttt{return}_b(y))$
end if

`accumBy` The function `accumBy f i` can be simulated by the following pseudocode.

Algorithm 13 `accumBy f i`

let e be the signal function input
let $b = \texttt{e2b(i)}(e')$
$x \overset{e}{\leftarrow} e$;
$y \overset{e}{\leftarrow} \texttt{b2e}(b)$;
$\texttt{return}_e(\texttt{f}(y)(x))$

where e' is the event stream output from this code, as obtained using the $\texttt{fix}_e$ function.

The function `accum i` is defined in Yampa in terms of `accumBy`.

`accumFilter` The function `accumFilter f i` can be simulated by the following pseudocode.

Algorithm 14 `accumFilter f i`

let e_1 be the signal function input
let $e = \texttt{fix}_e \left(\lambda e'.\text{let } b = \texttt{e2b(i)}(\{e'\}_2) \text{ in } x \overset{e}{\leftarrow} e; y \overset{e}{\leftarrow} \texttt{b2e}(b); \texttt{return}_e(\texttt{f}(y)(x))\right)$

$x \overset{e}{\leftarrow} e$;
if $x \neq \perp$ **then**
  $\texttt{return}_e(x)$
else
  $\texttt{zero}_e$
end if

`pre` and `iPre` Yampa's unit delay operators are not representable in this system, because they violate the property of timestep irrelevance. However, they are typically used to break ill-founded behavior loops. This effect can be achieved by instead using the `b2e` function. For example, the expression `arr f`, normally translated as:

$$(\lambda b.\, (\texttt{map}_e(\texttt{f})(b_e), \texttt{map}_b(\texttt{f})(b_b)))$$

when written as `arr (\b -> f (pre b))`, could instead be translated as:

$$\left(\lambda b.\, \left(\texttt{map}_e(\texttt{f})\left(x \overset{e}{\leftarrow} b_e; \texttt{b2e}(b_b)\right), \texttt{map}_b(\texttt{f})(b_b)\right)\right)$$

assuming that the function `f` is concerned only with changes in b.

Chapter 4

Conclusion

In this document, we have described the properties of timestep irrelevance and time invariance which preserve an ideal view of functional reactive systems, while providing for both efficient verification and implementation. We presented a small set of primitives sufficient to construct a functional reactive language which satisfy these and other properties, and prove these claims.

4.1 Future Work

As this thesis simply lays the foundation of a minimal subset of FRP, there are many ways in which future work can branch from here. These include reduction of the primitives needed, analyzing fixpoints, and implementation of a system using the primitives.

4.1.1 Further reductions

During the course of verifying the properties of the above outline set of combinators, the author noticed several other reductions which may be performed:

- $\texttt{return}_e(x)$ may be written as $\texttt{b2e}(\texttt{return}_b(x))$
- $\texttt{return}_b(x)$ may be written as $\texttt{e2b}(x)(\texttt{zero}_e)$
- $\texttt{join}_e(ee)$ could be removed in favor of the simpler $\texttt{join}_{eb}(eb) : B[E[\alpha]] \rightarrow E[\alpha]$, and then written as $\texttt{join}_{eb}(\texttt{e2b}(\texttt{zero}_e)(ee))$

This would reduce the number of necessary combinators from twelve to ten, and simplify one. Of course, this would result in the specifications of $\texttt{return}_e$ and $\texttt{return}_b$ being overly complicated, and inefficient if used directly in an implementation.

A different type of simplification, one inspired by Fran [9], would be to restrict event streams to have at most one occurrence of an event. A "traditional" multi-event stream of type $E[\alpha]$ could then be represented using this new type

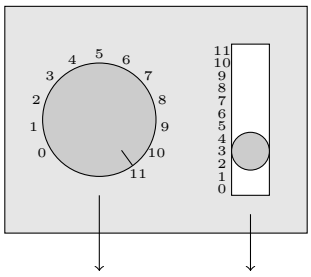

Figure 4.1: Two input controls

constructor (let us call it $\dot{E}$) recursively as $\dot{E}[\alpha \times E[\alpha]]$: that is, a single-event stream which carries as its value the value of the first occurrence of the multi-event stream, and a "continuation" multi-event stream containing the rest of the events.

Most notably, this would greatly simplify the $\texttt{once}_e$ function used repeatedly in chapter 3.5. It is also possible that this would simplify verification of the properties of the combinators.

4.1.2 Fixpoints

A recurring theme in FRP seems to be that of fixpoints. Most extant FRP systems deal with fixpoints by means of a unit delay operator, such as Yampa's `pre`. But, as shown in section 3.5.2, such operators violate the property of timestep irrelevance. They also cause the computations constructed by the fixpoints to be potentially non-terminating, which is undesirable in the real-time environments in which FRP is often intended to be used.

We do not want to disallow fixpoints, because many important constructs require them. For example, maintaining recurrent state, as with FrTime's `accum-e` function, requires fixpoints, although the use here is well-founded: the unit delay built into the `e2b` function ensures this. Furthermore, there are recurrent equations such as $x = x^{-1} + 1$ for which there are mathematical closed forms which could be determined through some form of analysis.

There is a particularly interesting case of fixpoints which occurs in GUI systems based on FRP. Consider the GUI in figure 4.1, consisting of two real-valued inputs. These two widgets could be created by the hypothetical functions `dial` and `slider`, each of type $\mathbb{R} \to \textit{widget} \times B[\mathbb{R}]$, taking an initial value and returning both a widget (to be composed into a larger GUI) and a behavior representing the widget's current value.

Consider now the case where we wish to *link* the two widgets together: a change in the value of one should be reflected in the other. This problem has been addressed in the context of declarative web programming by Greenberg [11] using an idea akin to database lenses. I propose however an alternative solution which would require no additional constructs beyond those provided by FRP. We can enhance our widget functions to accept as input a behavior rather than

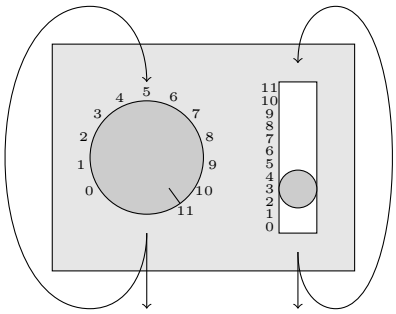

Figure 4.2: Two input/output controls

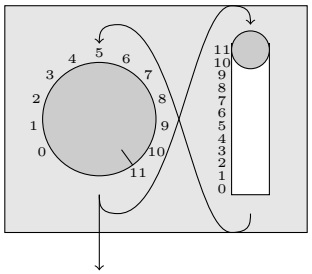

Figure 4.3: Linked input/output controls

an initial value: $B[\mathbb{R}] \to \mathit{widget} \times B[\mathbb{R}]$. Our new `dial′` and `slider′`, instead of displaying their internal state, display this input behavior. The output behavior continues to reflect the internal state as before, although is constrained by the input behavior.

We can recover the original behavior of our unlinked widgets by finding the fixpoint of `dial′` or `slider′`: by connecting a widget's output to its own input, it will display its internal state as expected (see figure 4.2). We can take this a step further however, by connecting the output of one widget to the input of the other, and vice-versa (see figure 4.3): the fixpoint of this system would be the one in which both widgets effectively share the same internal state, thus acheiving our desired behavior. This can be generalized to any number of linked widgets by simply connecting them in a loop.

It is important to note that the calculation of this fixpoint is non-trivial: much program analysis would be required to determine which widget is currently "driving" the loop. Furthermore, there must be a method provided to the programmer to initialize the fixpoint loop, since the initial solution would be otherwise undefined.

4.1.3 Implementation

Much focus has been given recently to efficient implementation of FRP systems. Techniques such as lowering [2] have been developed to perform algebraic simplification of functional reactive programs, and attention has been given to eliminating unnecessary updates [8]. Both these problems can be addressed by the primitives defined in this paper: algebraic simplification can be performed on FRP expressions by using the monad laws, and the guarantee of timestep irrelevance allows efficient push-based updates to be used without sacrificing the abstraction of continuous behaviors (and already has been done in OCaml RT [16]).

It should also be possible to inhabit the axioms defined in the Coq proofs with actual implementations. Such an implementation, although potentially inefficient, could simplify many of the proofs and allow for a reference implementation to be extracted into another language such as OCaml or Haskell.

4.1.4 Verification framework

During the course of development of this thesis, much work was done on developing a framework for formal verification of FRP systems, which would allow for the verification of properties such as that described in section 1.3. The first attempt was made using Coq as a foundation. We embedded linear temporal logic in Coq similarly to that shown in section 3.1 (using $nat \to \mathsf{Prop}$ to represent modal propositions, etc.), but also provided modal versions of all the Coq built-ins. This allowed modal formulas to be written in a very natural style in Coq. For example, the definition of the semantics of the $\texttt{return}_e$ operator could be written simply as:

```
Definition sem_return_e(A: Type) := { return_e: A -> Mt(E A) |
  forall e x, globally_valid (M e = return_e x =>
    eps e = M (Some x) /\ [X] [G] NoEvent(e)) }.
```

However, this effort was abandoned for two reasons. First, defining modal versions of all the Coq built-ins became very tedious: there were many special cases (for example, higher-order functions) which could not be covered by automated methods. Secondly, many of Coq's built-in proof tactics became useless due to the extra hidden λ-abstractions now present in every formula.

Taking into account the awkwardness of parameterizing all expressions by *nat*, a second attempt using Coq was made, in which modality was added by means of the following three axioms:

```
Axiom M: nat -> Prop -> Prop.
Axiom M_sum: forall (j k: nat) (P: Prop), M j (M k P) = M (j+k) P.
Axiom M_zero: forall (P: Prop), M 0 P = P.
```

This definition allowed for LTL formulas still to be represented very naturally, and for proofs to be carried out without needing to manage λ-abstractions. However,

a problem arose when dealing with expressions involving equality. Consider the expression $x = 1 \wedge \bigcirc(x = 2)$. While this is a valid LTL formula, written as-is in Coq, this is not valid: the equality $x = 1$ can be used to rewrite the x inside of $\bigcirc(x = 2)$, resulting in the invalid formula $\bigcirc(1 = 2)$. The solution to this problem would have been to utilize Coq's support for user-defined equality relations, but defining such was outside the scope of this project.

A third attempt was made at constructing a proof obligation generator for Coq using OCaml. This approach seemed promising, having been used successfully for verification of Lustre programs [7]. The overall design of the tool is to take as input a program written using the FRP primitives presented in section 2.3 and a set of assertions written in FOLTL, and produce as output a proof obligation in Coq. For example, given the following program which creates a behavior `v` counting events on the event stream `tick`:

```
v = collect_b succ tick 0
tick: { e: E A | [F*] (eps e = ()) }
succ(u: unit)(x: int): { y: int | y = x + 1 }
```

and assertions stating that `v` is monotonically increasing:

```
forall x y: int, [G]((x = beta v /\ [X] (y = beta v)) -> x <= y)
forall x y: int, [F*]((x = beta v /\ [X] (y = beta v)) -> x < y)
```

the tool would produce as output the following proof obligation in Coq:

```
(forall j, exists k, k >= j /\ (eps e k = Some ())) ->
(forall u, (eps tick 0 = Some u) ->
  exists y, (beta v 0 = y) /\ (y = 0 + 1)) ->
((eps e 0 = None) -> (beta v 0 = 0)) ->
(forall j u, (eps tick (j+1) = Some u) ->
  exists x y, (beta v (j+1) = y) /\ (beta v j = x) /\ (y = x + 1)) ->
(forall j, (eps tick (j+1) = None) -> (beta v (j+1) = beta v j)) ->
(forall j x y, (x = beta v j) /\ (y = beta v (j+1)) -> x <= y) /\
(forall j, exists k, k >= j /\
  forall x y, (x = beta v k) /\ (y = beta v (k+1)) -> x < y)
```

A parser and the beginnings of a translator were written, but not enough time remained to complete work on this tool, so it remains as future work.

Bibliography

[1] The Father Time Language (FrTime). `http://pre.plt-scheme.org/plt/collects/frtime/doc.txt`.

[2] K. Burchett, G.H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, 2007.

[3] G.H. Cooper. PhD thesis, Brown University. Not yet published.

[4] G.H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. *European Symposium on Programming*, 2006.

[5] A. Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, 2004.

[6] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[7] C. Dumas and P. Caspi. A PVS proof obligation generator for Lustre programs. *7th International Conference on Logic for Programming and Automated Reasoning*, 2000.

[8] C. Elliot. Simply efficient functional reactivity. 2008.

[9] C. Elliott and P. Hudak. Functional reactive animation. *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, 1997.

[10] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, 8:995–1072, 1990.

[11] M. Greenberg. Declarative, composable views, 2007. Undergraduate honors thesis, Brown University.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[13] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering, IEEE Transactions on*, 18(9):785–793, 1992.

[14] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. *Advanced Functional Programming: 4th International School, Afp 2002, Oxford, Uk, August 19-24, 2002: Revised Lectures*, 2003.

[15] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[16] C. King. Objective Caml Reactive Toolkit. `http://users.wpi.edu/~squirrel/ocamlrt/`.

[17] L. Meyerovich. Flapjax: Functional Reactive Web Programming, 2007. Undergraduate honors thesis, Brown University.

[18] P. Wadler. Comprehending monads. *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.

Appendix A

Proofs

A.1 Module LTL

```
Set Implicit Arguments.
Implicit Type P Q R: nat → Prop.

Definition next(A: Type)(x: nat → A) := fun i ⇒ x (S i).
Definition until P Q :=
  fun i ⇒ ∃ j, Q (i+j) ∧ ∀ k, k < j → P (i+k).
Definition release P Q :=
  fun i ⇒ ∀ j, Q (i+j) ∨ ∃ k, k < j ∧ P (i+k).
Definition finally P := fun i ⇒ ∃ j, P (i+j).
Definition globally P := fun i ⇒ ∀ j, P (i+j).
Definition infinitely_often P := globally (finally P).
Definition almost_everywhere P := finally (globally P).
Definition wuntil P Q := fun i ⇒ until P Q i ∨ globally P i.
Definition srelease P Q := fun i ⇒ release P Q i ∧ finally P i.
```

A.2 Module FRP

```
Set Implicit Arguments.
Implicit Type A B C: Set.
Implicit Type i j k: nat.

Require Import LTL.

Axiom ev beh: Set → Set.
Axiom eps: ∀ A, ev A → nat → option A.
Axiom beta: ∀ A, beh A → nat → A.

Definition NoEvent A (e: ev A) := fun i ⇒ eps e i = None.
Definition SomeEvent A (e: ev A) := fun i ⇒ ∃ x, eps e i = Some x.

Axiom return_e: ∀ A, nat → A → ev A.
```

```
Axiom return_e_sem_0: ∀ A i (x: A),
  let e := return_e i x in
  eps e i = Some x.
Axiom return_e_sem_n: ∀ A i (x: A),
  let e := return_e i x in
  next (globally (NoEvent e)) i.

Axiom map_e: ∀ A B, (nat → A → B) → nat → ev A → ev B.
Axiom map_e_sem: ∀ A B (f: nat → A → B) i e1,
  let e := map_e f i e1 in
  globally (fun i ⇒ eps e i = option_map (f i) (eps e1 i)) i.

Axiom zero_e: ∀ A, nat → ev A.
Implicit Arguments zero_e [A].
Axiom zero_e_sem: ∀ A i,
  let e: ev A := zero_e i in
  globally (NoEvent e) i.

Axiom plus_e: ∀ A, nat → ev A → ev A → ev A.
Axiom plus_e_sem: ∀ A i (e1 e2: ev A),
  let e := plus_e i e1 e2 in
  globally (fun i ⇒ IF NoEvent e1 i then eps e i = eps e2 i else eps e i = eps
e1 i) i.

Axiom join_e: ∀ A, nat → ev (ev A) → ev A.
Axiom join_e_sem: ∀ A i (ee: ev (ev A)),
  let e := join_e i ee in
  wuntil (NoEvent e) (SomeEvent ee) i ∧
  (∀ e1: ev A, globally (fun i ⇒ (eps ee i = Some e1) →
    release (next (SomeEvent ee)) (fun i ⇒ eps e i = eps e1 i) i) i).

Axiom fix_e: ∀ A, (nat → ev A → ev A) → nat → ev A.
Axiom fix_e_sem: ∀ A (f: nat → ev A → ev A) i,
  let e := fix_e f i in
  globally (fun i ⇒ eps e i = eps (f i e) i) i.

Axiom return_b: ∀ A, nat → A → beh A.
Axiom return_b_sem: ∀ A i (x: A),
  let b := return_b i x in
  globally (fun i ⇒ beta b i = x) i.

Axiom map_b: ∀ A B, (A → B) → nat → beh A → beh B.
Axiom map_b_sem: ∀ A B (f: A → B) i b1,
  let b := map_b f i b1 in
  globally (fun i ⇒ beta b i = f (beta b1 i)) i.

Axiom join_b: ∀ A, nat → beh (beh A) → beh A.
Axiom join_b_sem: ∀ A i (bb: beh (beh A)),
  let b := join_b i bb in
  globally (fun i ⇒ beta b i = beta (beta bb i) i) i.
```

```
Axiom fix_b: ∀ A, (beh A → beh A) → nat → beh A.
Axiom fix_b_sem: ∀ A (f: beh A → beh A) i,
  let b := fix_b f i in
  globally (fun i ⇒ beta b i = beta (f b) i) i.
```

A.3 Module `FRP_facts`

```
Set Implicit Arguments.
Implicit Type A B C: Set.
Implicit Type i j k: nat.

Require Import Arith.
Require Import LTL FRP.

Theorem ev_dec: ∀ A (e: ev A) i, {SomeEvent e i}+{NoEvent e i}.
Proof.
  intros A e i. unfold SomeEvent, NoEvent. destruct (eps e i).
    left. ∃ a. reflexivity.
    right. reflexivity.
Qed.

Lemma le_minus_O: ∀ n m, n ≤ m → n - m = O.
Proof.
  intros n m Hnm. assert ({n < m}+{n = m}) as H.
    apply le_lt_eq_dec. assumption.
  destruct H as [H | H].
    apply not_le_minus_0. apply lt_not_le. assumption.
    subst m. symmetry. apply minus_n_n.
Qed.

Section first_event.

Variable A: Set.
Variable e: ev A.
Variables i j: nat.
Variable Hj: SomeEvent e (i + j).

Program Fixpoint find_first_event (k: { k | ∀ k', k' < j - k → NoEvent e (i +
k')}) {measure proj1_sig k}:
  { f | f ≤ j ∧ SomeEvent e (i + f) ∧ ∀ k, k < f → NoEvent e (i + k) } :=
  match eps e (i + (j - k)) with
  | Some x ⇒ j - k
  | None ⇒
    match k with
    | S k' ⇒ find_first_event k'
    | O ⇒ j
    end
  end.
```

```
Next Obligation.
Proof.
  split.
    apply le_minus.
    split.
      unfold SomeEvent, NoEvent. replace (eps e (i + (j - k))) with (Some x).
∃ x. reflexivity.
      assumption.
Qed.
Next Obligation.
Proof.
  assert ({k'0 < j - S k'}+{k'0 = j - S k'}) as Hjk'0.
    destruct j as [| n].
      elimtype False. apply (lt_n_O k'0). assumption.
      simpl. assert ({k'0 < n - k'}+{k'0 = n - k'}).
        apply le_lt_eq_dec. apply lt_n_Sm_le. assert ({k' ≤ n}+{n < k'}) by
apply le_lt_dec.
        decompose sum H1.
          rewrite minus_Sn_m. assumption.
            assumption.
          rewrite minus_Sn_m. assumption. assert (S n - k' = 0).
            apply le_minus_O. apply lt_le_S. assumption.
          replace (S n - k') with 0 in ×. elimtype False. apply (lt_n_O k'0).
assumption.
      assumption.
  destruct Hjk'0 as [Hjk'0 | Hjk'0].
    auto.
    subst k'0. unfold NoEvent. symmetry. assumption.
Qed.
Next Obligation.
Proof.
  rewrite ← minus_n_O in ×. auto.
Qed.

Program Let first_event_sig := find_first_event j.
Next Obligation.
Proof.
  rewrite ← minus_n_n in ×. elimtype False. apply (lt_n_O k'). assumption.
Qed.

Program Theorem first_event_dec:
  ∃ f, f ≤ j ∧ SomeEvent e (i + f) ∧ ∀ k, k < f→ NoEvent e (i + k).
Proof.
  destruct first_event_sig as [k H]. ∃ k. assumption.
Qed.

End first_event.
```

A.4 Module `monad_laws`

```
Set Implicit Arguments.
Implicit Type A B C: Set.
Implicit Type i j k: nat.

Require Import Arith.
Require Import LTL FRP FRP_facts.

Notation "f @ g" := (fun x ⇒ f (g x))
  (at level 97, left associativity).
Notation "e1 %= e2" := (globally (fun i ⇒ eps e1 i = eps e2 i))
  (at level 70, no associativity).
Notation "b1 @= b2" := (globally (fun i ⇒ beta b1 i = beta b2 i))
  (at level 70, no associativity).

Ltac rewrite_e :=
  repeat (
    rewrite return_e_sem_0 in × || rewrite return_e_sem_n in × ||
    rewrite map_e_sem in × || rewrite join_e_sem in *).

Ltac rewrite_b :=
  repeat (
    rewrite return_b_sem in × || rewrite map_b_sem in × || rewrite join_b_sem
in *).

Theorem M1e: ∀ A e1 i,
  (map_e (fun i (x: A) ⇒ x) i e1 %= (fun x ⇒ x) e1) i.
Proof.
  intros A e1 i j. rewrite_e. destruct (eps e1 (i+j)); reflexivity.
Qed.

Theorem M2e: ∀ A B C (f: nat → B → C) (g: nat → A → B) e1 i,
  (map_e (fun i ⇒ f i @ g i) i e1 %= (map_e f i @ map_e g i) e1) i.
Proof.
  intros A B C f g e1 i j. rewrite_e. destruct (eps e1 (i+j)); reflexivity.
Qed.

Theorem M3e: ∀ A B (f: nat → A → B) x i,
  ((map_e f i @ return_e i) x %= (return_e i @ f i) x) i.
Proof.
  intros A B f x i j.
  rewrite_e. destruct j;
  [rewrite plus_0_r | rewrite <- plus_Snm_nSm]; rewrite_e; reflexivity.
Qed.

Theorem M5e: ∀ A (e: ev A) i,
  ((join_e i @ return_e i) e %= (fun e ⇒ e) e) i.
Proof.
  intros A e i j.
```

```
  pose (He := join_e_sem i (return_e i e)).
  destruct He as [_Her]. pose (Her_ej := Her e 0). rewrite ← plus_n_O in
Her_ej.
  rewrite_e. assert (Some e = Some e) as H1 by reflexivity. pose (H2 := Her_ej
H1).
  destruct (H2 j) as [H3 | H3].
    assumption.
    destruct H3 as [k [_ [x H3]]]. pose (H4 := return_e_sem_n i e k).
    unfold SomeEvent in H3. unfold NoEvent in H4.
    rewrite plus_Sn_m in H4. rewrite H3 in H4.
    elimtype False. cut (Some x = None). discriminate. assumption.
Qed.

Theorem M6e: ∀ A (e: ev A) i,
  ((join_e i @ map_e (@return_e A) i) e %= (fun e ⇒ e) e) i.
Proof.
  intros A e i j.
  pose (Hjoin := join_e_sem i (map_e (return_e (A:=A)) i e)).
  destruct Hjoin as [Hjoinl Hjoinr]. destruct Hjoinl as [Hjoinl | Hjoinl].
    destruct Hjoinl as [k [Hjoinll Hjoinlr]].
    pose (Hfirst := first_event_dec i k Hjoinll).
    destruct Hfirst as [f [Hfirst1 [Hfirst2 Hfirst3]]].
    destruct (le_lt_eq_dec f k Hfirst1) as [Hfk | Hfk].
      pose (H1 := Hjoinlr f Hfk).
      unfold SomeEvent in Hfirst2.
      destruct Hfirst2 as [e1 Hfirst2].
      pose (H := Hjoinr Hfirst2).
      unfold globally in Hjoinr.

    unfold SomeEvent, NoEvent in ×. rewrite map_e_sem in Hjoinll, Hfirst2.

      unfold SomeEvent, NoEvent in Hjoinll. rewrite map_e_sem in Hjoinll.
      assert ({j < k}+{j = k}+{k < j}) as Hjk by apply lt_eq_lt_dec.
      decompose sum Hjk.
        rewrite Hjoinlr.
          destruct (eps e (i + j)).

  pose (Hmap := map_e_sem (return_e (A:=A)) i e j). simpl in Hmap.
  assert ({NoEvent e (i + j)}+{SomeEvent e (i + j)}) as H1 by apply ev_dec.
  unfold SomeEvent, NoEvent in H1. destruct H1 as [H1 | H1].
    rewrite H1 in ×. simpl in Hmap.
    unfold globally in Hjoinr. destruct (eps e (i + j)).

Theorem M4e: ∀ A B (f: nat → A → B) ee i,
  ((map_e f i @ join_e i) ee %= (join_e i @ map_e (map_e f) i) ee) i.
Proof.
  intros A B f ee i j.
  rewrite_e.
```

```
  assert (∀ k, SomeEvent ee (i+k) ↔ SomeEvent (map_e (map_e f) i ee) (i+k))
as Hsync.
    intro k. unfold SomeEvent, NoEvent. rewrite_e. destruct (eps ee (i+k)).
      split; intros; discriminate.
      split; auto.
  pose (Hee := join_e_sem i ee).
  pose (Hmap := join_e_sem i (map_e (map_e f) i ee)).
  destruct Hee as [Heel Heer]. destruct Hmap as [Hmapl Hmapr].
  destruct Heel as [Heel | Heel]; destruct Hmapl as [Hmapl | Hmapl].
    destruct Heel as [k Heel]. destruct Hmapl as [k' Hmapl].
    assert ({k < k'}+{k = k'}+{k' < k}) as Hkk' by apply lt_eq_lt_dec.
    decompose sum Hkk'.
      destruct Heel as [H1 _]. destruct Hmapl as [H2 H3].
      destruct (Hsync k) as [H3 _]. absurd (NoEvent (map_e (map_e f) i ee)
(i + k)).
        apply H3. assumption.
        apply H2.

    assert ({j < k}+{j = k}+{k < j}) as Hjk by apply lt_eq_lt_dec.
    decompose sum Hjk.
      destruct Heel as [_H1]. destruct Hmapl as [_H2].

      rewrite Heel. simpl.
      destruct Hmapl as [Hmapl | Hmapl].
        destruct Hmapl as [k [Hmapll Hmaplr]].
        assert ({j < k}+{j = k}+{k < j}) as H1 by apply lt_eq_lt_dec.
        decompose sum H1.
          rewrite Hmaplr. reflexivity. assumption.
          assert (∃ e1, eps (map_e (map_e f) i ee) (i+k) = Some e1) as H2.
            destruct (eps (map_e (map_e f) i ee) (i+k)).
              ∃ e. reflexivity.
              contradiction Hmapll. reflexivity.
            subst k. destruct H2 as [e1 H2]. pose (H3 := Hmapr e1 j H2).
            destruct (H3 0) as [H4 | H4]. rewrite plus_0_r in H4. rewrite H4.
          unfold globally in Hmapr. pose (H2 := Hmapr
                  assert ({j < k}+{j = k}+{k < j}) as H1 by apply lt_eq_lt_dec.

        rewrite Hmapl. reflexivity.

      destruct Heel as [k [Heell Heelr]].

  destruct (eps ee i).
  rewrite (map_e_sem f).*)
   Theorem M1b: ∀ A b1 i,
  (map_b (fun (x: A) ⇒ x) i b1 @= (fun x ⇒ x) b1) i.
Proof. intros A b1 i j. rewrite_b. reflexivity. Qed.

Theorem M2b: ∀ A B C (f: B → C) (g: A → B) b1 i,
  (map_b (f @ g) i b1 @= (map_b f i @ map_b g i) b1) i.
```

Proof. *intros A B C f g b1 i j. rewrite_b. reflexivity.* Qed.

Theorem *M3b*: ∀ *A B* (*f*: *A* → *B*) *x i*,
  ((*map_b f i* @ *return_b i*) *x* @= (*return_b i* @ *f*) *x*) *i*.
Proof. *intros A B f x i j. rewrite_b. reflexivity.* Qed.

Theorem *M4b*: ∀ *A B* (*f*: *A* → *B*) *bb i*,
  ((*map_b f i* @ *join_b i*) *bb* @= (*join_b i* @ *map_b* (*map_b f i*) *i*) *bb*) *i*.
Proof. *intros A B f bb i j. rewrite_b. reflexivity.* Qed.

Theorem *M5b*: ∀ *A* (*b*: *beh A*) *i*,
  ((*join_b i* @ *return_b i*) *b* @= (fun *b* ⇒ *b*) *b*) *i*.
Proof. *intros A b i j. rewrite_b. reflexivity.* Qed.

Theorem *M6b*: ∀ *A* (*b*: *beh A*) *i*,
  ((*join_b i* @ *map_b* (*return_b i*) *i*) *i*) *b* @= (fun *b* ⇒ *b*) *b*) *i*.
Proof. *intros A b i j. rewrite_b. reflexivity.* Qed.

Theorem *M7b*: ∀ *A* (*bbb*: *beh* (*beh* (*beh A*))) *i*,
  ((*join_b i* @ *map_b* (*join_b i*) *i*) *i*) *bbb* @= (*join_b i* @ *join_b i*) *bbb*) *i*.
Proof. *intros A b i j. rewrite_b. reflexivity.* Qed.

A.5 Module `time_invariance`

Set Implicit *Arguments*.
Implicit Type *A B C*: Set.
Implicit Type *i j k*: *nat*.

Require Import *Arith*.
Require Import *LTL FRP FRP_facts*.

Notation "*f* @ *g*" := (fun *x* ⇒ *f* (*g x*))
  (*at level* 97, *left associativity*).
Notation "*e1* %= *e2*" := (*globally* (fun *i* ⇒ *eps e1 i* = *eps e2 i*))
  (*at level* 70, *no associativity*).
Notation "*b1* @= *b2*" := (*globally* (fun *i* ⇒ *beta b1 i* = *beta b2 i*))
  (*at level* 70, *no associativity*).

Ltac *rewrite_e* :=
  repeat (
    rewrite return_e_sem_0 in × || *rewrite return_e_sem_n* in × ||
    rewrite zero_e_sem in × || *rewrite plus_e_sem* in × ||
    rewrite map_e_sem in × || *rewrite join_e_sem* in *).

Ltac *rewrite_b* :=
  repeat (
    rewrite return_b_sem in × || *rewrite map_b_sem* in × || *rewrite join_b_sem*
in *).

Axiom *shift_e*: ∀ *A*, *nat* → *ev A* → *ev A*.

```
Axiom eps_shift: ∀ A i j (e: ev A), eps (shift_e i e) (i + j) = eps e j.
Axiom shift_b: ∀ A, nat → beh A → beh A.
Axiom beta_shift: ∀ A i j (b: beh A), beta (shift_b i b) (i + j) = beta b j.

Theorem return_e_ti:
  ∀ A (x: A) e, (e %= return_e 0 x) 0 →
    ∀ i, (shift_e i e %= return_e i x) i.
Proof.
  intros A x e H1 i j. rewrite eps_shift. unfold globally in H1.
  destruct j.
    rewrite ← plus_n_O. pose (H10 := H1 0).
      simpl in H10. rewrite H10. rewrite_e. reflexivity.
    pose (Hri := return_e_sem_n i x).
      rewrite ← plus_n_Sm. rewrite ← plus_Sn_m. rewrite Hri.
      pose (H1Sj := H1 (S j)). simpl in H1Sj. rewrite H1Sj.
      pose (Hr0 := return_e_sem_n 0 x j). simpl in Hr0. rewrite Hr0. reflexivity.
Qed.

Theorem map_e_ti:
  ∀ A B (f: nat → A → B) e1 e, (e %= map_e f 0 e1) 0 →
    ∀ i, (shift_e i e %= map_e (fun j ⇒ f (j - i)) i (shift_e i e1)) i.
Proof.
  intros A B f e1 e H1 i j. unfold globally in H1.
  rewrite_e. repeat rewrite eps_shift. rewrite minus_plus.
  pose (H1j := H1 j). rewrite_e. simpl in H1j. rewrite H1j. reflexivity.
Qed.

Theorem zero_e_ti:
  ∀ A (e: ev A), (e %= zero_e 0) 0 →
    ∀ i, (shift_e i e %= zero_e i) i.
Proof.
  intros A e H1 i j. unfold globally in H1. rewrite eps_shift.
  pose (H1j := H1 j). rewrite_e. simpl in H1j. rewrite H1j. reflexivity.
Qed.

Theorem lplus_e_ti:
  ∀ A (e1: ev A) e2 e, (e %= plus_e 0 e1 e2) 0 →
    ∀ i, (shift_e i e %= plus_e i (shift_e i e1) (shift_e i e2)) i.
Proof.
  intros A e1 e2 e H1 i j. unfold globally in H1. rewrite eps_shift.
  pose (H1j := H1 j). simpl in H1j.
  pose (Hp0 := plus_e_sem 0 e1 e2 j).
  pose (Hpi := plus_e_sem i (shift_e i e1) (shift_e i e2) j).
  unfold NoEvent in Hp0, Hpi. repeat rewrite eps_shift in Hpi. simpl in Hp0.
  unfold IF_then_else in ×.
  assert ({SomeEvent e1 j}+{NoEvent e1 j}) as He1 by apply ev_dec.
  destruct He1 as [He1 | He1]; destruct Hp0 as [Hp0 | Hp0]; destruct Hpi as [Hpi
| Hpi].
```

destruct He1 as [x He1]. *rewrite He1* in *Hp0*. *destruct Hp0*. *discriminate*.
destruct He1 as [x He1]. *rewrite He1* in *Hp0*. *destruct Hp0*. *discriminate*.
destruct He1 as [x He1]. *rewrite He1* in *Hpi*. *destruct Hpi*. *discriminate*.
destruct Hpi as [_Hpi]. *rewrite Hpi*. *rewrite H1j*. *destruct Hp0* as [_Hp0]. *rewrite Hp0*. *reflexivity*.
destruct Hpi as [_Hpi]. *rewrite Hpi*. *rewrite H1j*. *destruct Hp0* as [_Hp0]. *rewrite Hp0*. *reflexivity*.
unfold NoEvent in *He1*. *rewrite He1* in *Hpi*. *destruct Hpi* as [Hpi _]. *elimtype False*. *apply Hpi*. *reflexivity*.
unfold NoEvent in *He1*. *rewrite He1* in *Hp0*. *destruct Hp0* as [Hp0 _]. *elimtype False*. *apply Hp0*. *reflexivity*.
unfold NoEvent in *He1*. *rewrite He1* in *Hp0*. *destruct Hp0* as [Hp0 _]. *elimtype False*. *apply Hp0*. *reflexivity*.
Qed.

Theorem *return_b_ti*:
∀ *A* (*x*: *A*) *b*, (*b* @= *return_b* 0 *x*) 0 →
∀ *i*, (*shift_b i b* @= *return_b i x*) *i*.
Proof.
intros A x b H1 i j. *pose* (*H1j* := *H1 j*). *cbv beta* in *H1j*.
rewrite beta_shift. *rewrite_b*. *assumption*.
Qed.

Theorem *map_b_ti*:
∀ *A B* (*f*: *A* → *B*) *b1 b*, (*b* @= *map_b f* 0 *b1*) 0 →
∀ *i*, (*shift_b i b* @= *map_b f i* (*shift_b i b1*)) *i*.
Proof.
intros A B f b1 b H1 i j. *pose* (*H1j* := *H1 j*). *cbv beta* in *H1j*.
rewrite_b. *repeat rewrite beta_shift*. *assumption*.
Qed.

Theorem *join_b_ti*:
∀ *A* (*bb*: *beh* (*beh A*)) *b*, (*b* @= *join_b* 0 *bb*) 0 →
∀ *i*, (*shift_b i b* @= *join_b i* (*shift_b i bb*)) *i*.
Proof.
intros A bb b H1 i j. *pose* (*H1j* := *H1 j*). *cbv beta* in *H1j*.
rewrite_b. *repeat rewrite beta_shift*. *assumption*.
Qed.