Time-Flies: Push-Pull Signal-Function Functional Reactive Programming

by

Edward AmsdenB.S.

THESIS

Presented to the Faculty of

the Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Rochester Institute of Technology

December 2012

Time-Flies: Push-Pull Signal-Function Functional Reactive Programming

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Matthew Fluet, Supervisor

Arthur Nunes-Harwitt, Reader

Dr. Zach Butler, Observer

Acknowledgments

I wish to thank the multitudes of people who helped me. Time would fail me to tell of ...

Abstract

Time-Flies: Push-Pull Signal-Function Functional Reactive Programming

Edward Amsden, M.S.
Rochester Institute of Technology, 2012

Supervisor: Dr. Matthew Fluet

Functional Reactive Programming (FRP) is a promising class of abstractions for encoding interactive and time-dependent programs in functional languages which has proven difficult to implement efficiently. Signal-function FRP is a subclass of FRP which does not provide direct access to the time-varying values by the programmer, but rather provides signal functions as first-class objects in the program. FRP systems provide values defined at all points in time (*behaviors* or *signals*) and values defined at countably many points in time (*events*).

All signal-function implementations of FRP to date have utilized demand-driven or "pull-based" evaluation for both events and signals, producing output from the FRP system whenever the consumer of the output is ready.

This greatly simplifies the implementation of signal-function FRP systems, but leads to inefficient and wasteful evaluation of the FRP system.

In contrast, an input-driven or "push-based" system evaluates the network whenever new input is available. This frees the system from evaluating the network when nothing has changed, and then only components necessary to react to the input are reevaluated. This form of evaluation has been applied to events in standard FRP systems [7] but not in signal-function FRP systems.

I describe the implementation of a push-based signal-function FRP system. The semantics of the system are discussed, and its performance and expressiveness for practical examples of interactive programs are compared to existing signal-function FRP systems through the implementation of a networking application.

Table of Contents

List of Figures

Chapter 1

Introduction

Most of the useful programs which programmers are asked to write must react to inputs which are not available at the time the program starts, and produce effects at many different times throughout the execution of the program. Examples of these programs include GUI applications, web browsers, web applications, video games, multimedia authoring and playback tools, operating system shells and kernels, servers, robotic control programs, and many others. This ability is called reactivity.

Functional reactive programming (FRP) is a promising class of abstractions for writing reactive programs in functional languages. FRP abstractions translate desirable properties of the underlying functional languages, such as higher-order functions and referential transparency, to reactive constructs, generally without modifications to the underlying language. This permits the compositional construction of reactive programs in purely functional languages.

Functional reactive programming was introduced, though not quite by name, with Fran [8], a compositional system for writing reactive animations. From the start, two key challenges emerged: the enforcement of *causality*[1] in

[1]Causality is the property that a value in an FRP program depends only on present or

FRP programs, and the efficient evaluation of FRP programs.

The first challenge was addressed by representing FRP programs, not as compositions of signals and events, but as compositions of transformers of behaviors and events called *signal functions*. By not permitting direct access to behaviors or events, but representing them implicitly, signal functions prohibit accessing past or future time values, only allowing access to values in the closure of the signal function and current input values. Signal function FRP programs are written by composing signal functions, and are evaluated using a function which provides input to the signal function and acts on the output values. This evaluation function is elided in existing literature on signal functions. A further advantage of signal function FRP programs is that they are more compositional, since additional signal functions may be composed on the input or output side of a signal function, rather than simply the output side in classic (non-signal-function) FRP systems.

The second challenge was addressed for classic FRP by the creation of a hybrid push-pull FRP system [7]. This system relied on the distinction between reactivity and time dependence to decompose behaviors into phases of constant or simply time-dependent values, with new phases initiated by event occurrences. However, this implementation did not address concerns of causality or compositionality. Further, the existence of events as first-class values in classic FRP forced the use of an impure and resource-leaking technique to

---

past values. Obviously, a program which violates this property is impossible to evaluate, but in some FRP systems such programs are unfortunately quite easy to write down.

compare events to determine which had the first occurrence after the current time. This technique made use of the Haskell `unsafePerformIO` function to fork two threads to compare the improving values of the next occurrence time for two events being merged. Further, since the classic FRP interface permits access to only the output of a behavior or event, and is always bound to its implicit inputs, the best the push-based implementation could do is to block when no computation needs to be performed. The computation cannot be suspended entirely as a value representation.

To address both challenges in a FRP implementation, it seems desirable to combine the approaches, creating a push-based signal-function FRP system. The current implementation approach for signal-function FRP is inherently pull-based [13], but recent work on N-ary FRP [16], a variant of signal-function FRP which enumerates inputs and distinguishes between events and behaviors, provides a representation which suggests a method of push-based evaluation. In previous implementations of FRP, events were simply option-valued signals. This approach has the advantage of being simple to implement in a pull-based setting, but does not have an obvious semantics for event merging, and necessitates pull-based evaluation, since there is no way to separate the events which may be pushed from the truly continuous values which must be repeatedly sampled.

This thesis describes the design and implementation of Time-Flies, a push-pull signal-function FRP system. This system is presented as a library of combinators for constructing and evaluating FRP programs.

Chapter 2

Background

The key abstraction of functional programming is the function, which takes an input and produces an output. Functions may produce functions as output and/or take functions as input. We say that functions are *first-class values* in a language.

Even in functional languages, reactive programs are generally written in an imperative style, using low-level and non-composable abstractions including callbacks or object-based event handlers, called by event loops. This ties the model of interactivity to low-level implementation details such as timing and event handling models.

Functional Reactive Programming implies that a model should keep the characteristics of functional programming (i.e. that the basic constructs of the language should remain first-class) while incorporating reactivity into the language model. In particular, functions should be lifted to operate on reactive values, and functions themselves ought to be reactive.

The goal of FRP is to provide compositional and high-level abstractions for creating reactive programs. The key abstractions of FRP are behaviors or

signals[1], which are time-dependent values defined at every point in continuous time, and events, which are values defined at countably many points in time. An FRP system will provide functions to manipulate events and signals and to react to events by replacing a portion of the running program in response to an event. Behaviors and events, or some abstraction based on them, will be first class, in keeping with the spirit of functional programming. Programs implemented in FRP languages should of course be efficiently executable. This has proven to be the main challenge in implementing FRP.

The two general approaches to FRP are "classic" FRP, where behaviors and signals are first-class and reactive objects, and "signal-function" FRP, where transformers of signals and events are first-class and reactive objects.

2.1 Classic FRP

The earliest and still standard formulation of FRP provides two primitive type constructors: `Behavior` and `Event`, together with combinators that produce values of these types. The easiest semantic definition for these types is given in Figure 2.1[2].

When these two type constructors are exposed directly, the system is known as a *Classic FRP* system. If the type aliases are taken as given in the semantic definition, a simple, yet problematic, implementation is given in

[1]Behaviors are generally referred to as *signals* in signal function literature. This is unfortunate, since a signal function may operate on events, signals, or some combination of the two.

[2]The list constructor `[]` should be considered to contain infinite as well as finite lists.

```
type Event a = [(Time, a)]
type Behavior a = Time -> a
```

Figure 2.1: Semantic types for Classic FRP

Figure 2.2.

Obvious problems include the lack of ordering enforcement on event occurrences, the necessity of waiting for the first event occurrence in a switch, and the necessity of maintaining the full history of a behavior for evaluation.

2.1.1 History of Classic FRP

Classic FRP was originally described as the basis of Fran [8] (Functional Reactive ANimation), a framework for declaratively specifying interactive animations. Fran represented behaviors as two sampling functions, one from a time to a value and a new behavior (so that history may be discared), and the other from a time interval (a lower and upper time bound) to a value interval and a new behavior. Events are represented as "improving values", which, when sampled with a time, produce a lower bound on the time to the next occurrence, or the next occurrence if it has indeed occurred.

The first implementation of FRP outside the Haskell language was Frappé, an FRP implementation for the Java Beans framework. Frappé built on the notion of events and "bound properties" in the Beans framework, providing abstract interfaces for FRP events and behaviors, and combinators as concrete classes implementing these interfaces. The evaluation of Frappé used Bean components as sources and sinks, and the implementation of Bean events

```
time :: Behavior Time
time = id

constant :: a -> Behavior a
constant = const

delayB :: a -> Time -> Behavior a -> Behavior a
delayB a td b te = if te <= td
                   then a
                   else b (te - td)

instance Functor Behavior where
  fmap = (.)

instance Applicative Behavior where
  pure = constant
  bf <*> ba = (\t -> bf t (ba t))

never :: Event a
never = []

once :: a -> Event a
once a = [(0, a)]

delayE :: Time -> Event a -> Event a
delayE td = map (\(t, a) -> (t + td, a))

instance Functor Event where
  fmap f = map (\(t, a) -> (t, f a))

switcher :: Behavior a -> Event (Behavior a) -> Behavior a
switcher b [] t = b t
switcher b [(to, bo)] t = if t < to
                          then b t
                          else bo t
```

Figure 2.2: An obvious, yet inefficient and problematic, implementation of Classic FRP

and bound properties to propagate changes to the network [4].

2.1.2 Current Classic FRP Systems

The FrTime[3] [3] language extends the Scheme evaluator with a mutable dependency graph, which is constructed by program evaluation. This graph is then updated by signal changes. FrTime does not provide a distinct concept of events, and chooses branches of the dependency graph by conditional evaluation of signals, rather than by the substitution of signals used by FRP systems.

The Reactive [7] system is a push-pull FRP system with first-class behaviors and events. The primary insight of Reactive is the separation of reactivity (or changes in response to events whose occurrence time could not be known beforehand) from time-dependence. This gives rise to *reactive normal form*, which represents a behavior as a constant or simple time-varying value, together with an event stream carrying values which are also behaviors in reactive normal form. Push-based evaluation is achieved by forking a Haskell thread to evaluate the head behavior, while waiting on the evaluation of the event stream. Upon an event occurrence, the current behavior thread is killed and a new thread spawned to evaluate the new behavior. Unfortunately, the implmentation of Reactive uses a tenuous technique which depends on also forking threads to evaluate two haskell values concurrently in order to implement event merging. This relies on the library author to ensure consis-

[3]FrTime is available in the current version of Dr Racket (5.2.1).

tency when this technique is used, and leads to thread leakage when one of the merged events is itself the merging of other events.

A recent thesis [6] described Elm, a stand-alone language for reactivity. Elm provides combinators for manipulating discrete events, and compiles to Javascript, making it useful for client-side web programming. However, Elm does not provide a notion of switching or of continuous time behaviors, though an approximation is given using discrete-time events which are actuated at repeated intervals specified during the event definition. The thesis asserts that Arrowized FRP (signal-function FRP, Section 2.2) can be embedded in Elm, but provides no support for this assertion.

The reactive-banana [1] library is a push-based FRP system designed for use with Haskell GUI frameworks. In particular, it features a monad for the creation of behaviors and events which may then be composed and evaluated. This monad includes constructs for binding GUI library constructs to primitive events. It must be "compiled" to a Haskell IO action for evaluation to take place. The implementation of reactive-banana is similar to FrTime, using a dependency graph to update the network on event occurences. Reactive-banana eschews generalized switching in favor of branching functions on behavior values, similarly to FrTime, but maintains the distinction between behaviors and events. Rather than a generalized switching combinator which allows the replacement of arbitrary behaviors, reactive-banana provides a step combinator which creates a stepwise behavior from the values of an event stream.

```
(>>>)  :: (Arrow a) => a b c -> a c d -> a b d
arr    :: (Arrow a) => (b -> c) -> a b c
first  :: (Arrow a) => a b c -> a (b, d) (c, d)
second :: (Arrow a) => a b c -> a (d, b) (d, c)
(***)  :: (Arrow a) => a b c -> a b d -> a b (c, d)
loop   :: (Arrow a) => a (b, d) (c, d) => a b c
```

Figure 2.3: Arrow combinators.

2.2 Signal Function FRP

An alternative approach to FRP was first proposed in work on Fruit [5], a library for declarative specification of GUIs. This library utilized the abstraction of Arrows [9] to structure *signal functions*. Arrows are abstract type constructors with input and output type parameters, together with a set of routing combinators (see Figure 2.3). Signal functions are the first-class abstraction in this approach, they represent time-dependent and reactive transformers of events and signals, which are themselves not first class, since such values cannot be directly manipulated by the programmer. This approach has two motivations: it increases modularity since both the input and output of signal functions may be transformed, as opposed to signals or events which may only be transformed in their output, and it avoids a large class of time and space leaks which have emerged when implementing FRP with first-class signals and events.

Similarly to FrTime, the netwire [17] library eschews dynamic switching, in this case in favor of *signal inhibition*. Netwire is written as an arrow transformer, which, together with the Kliesli arrow instance for the IO

monad[4], permits it to lift IO actions as sources and sinks at arbitrary points in a signal function network. Signal inhibition is accomplished by making the output of signal functions a monoid, and then combining the outputs of signal functions. An inhibited signal function will produce the monoid's zero as an output. Primitives have defined inhibition behavior, and composed signal functions inhibit if their outputs combine to the monoid's zero.

Yampa [12] is an optimization of the Arrowized FRP system first utilized with Fruit (see above). The implementation of Yampa makes use of Generalized Algebraic Datatypes to permit a much larger class of type-safe datatypes for the signal function representation. This representation, together with "smart" constructors and combinators, enables the construction of a self-optimizing arrowized FRP system. Unfortunately, the primary inefficiency, that of unnecessary evaluation steps due to pull-based evaluation, remains. Further, the optimization is ad-hoc and each new optimization requires the addition of new constructors, as well as the updating of every primitive combinator to handle every combination of constructors. However, Yampa showed noticeable performance gains over previous Arrowized FRP implementations.

A recent PhD thesis [16] introduced N-Ary FRP, a technique for typing Arrowized FRP systems using dependent types. The bulk of the work consisted in using the dependent type system to prove the correctness of the FRP system discussed. This work introduced the typing construct of signal vectors, which

[4]Any monad forms a Kliesli arrow.

permit the distinction of signal and event types at the level of the FRP system, instead of making events merely a special type of signal.

2.3 Outstanding Challenges

At present, there are two key issues apparent with FRP. First, while signal-function FRP is inherently safer and more modular than classic FRP, it has yet to be efficiently implemented. Classic FRP programs are vulnerable to time leaks and violations of causality due to the ability to directly manipulate reactive values. Second, the interface between FRP programs and the many different sources of inputs and sinks for outputs available to a modern application writer remains ad-hoc and is in most cases limited by the implementation.

One key exception to this is the reactive-banana system, which provides a monad for constructing primitive events and behaviors from which an FRP program may then be constructed. However, this approach is still inflexible as it requires library support for the system which the FRP program will interact with. Further, being a Classic FRP system, reactive-banana lacks the ability to transform the inputs of behaviors and events, since all inputs are implicit.

Chapter 3

System Design and Interface

3.1 Goals

The (not yet fully realized) goal of FRP is to provide an efficient, declarative abstraction for creating reactive programs. Towards this overall goal, there are three goals which this system is intended to meet.

3.1.1 Efficient Evaluation

Efficient evaluation is the motivation for push-based evaluation of events. Since FRP programs are expected to interact with an outside world in real time, efficiency cannot be measured by the runtime of a program. Thus, when speaking of efficiency, we are expressing a desire that the system utilize as few system resources as possible for the task at hand, while responding as quickly as possible to external inputs and producing output at a consistently high sample rate.

3.1.2 Composability

A composable abstraction is one in which values in that abstraction may be combined in such a way that reasoning about their actions together involves little more than reasoning about their actions separately. In a signal

function system, the only interaction between composed signal functions ought to be that the output of one is the input of another. Composability permits a particularly attractive form of software engineering in which successively larger systems are created from by combining smaller systems, without having to reason about the components of the systems being combined.

3.1.3 Ease of Integration

It is fine for a system to be composable with regards to itself, but an FRP system must interact with the outside world. Since we cannot anticipate every possible form of input and output that the system will be asked to interact with, we must interface with Haskell's IO system. In particular, most libraries for user interaction (e.g. GUI and graphics libraries such as GTK+ and GLUT) and most libraries for time-dependent IO (e.g. audio and video systems) make use of the event loop abstraction. In this abstraction, event handlers are registered with the system, and then a command is issued to run a loop which detects events and runs the handlers, and uses the results of the handlers to render the appropriate output.

We would like for the FRP system to be easy to integrate with such IO systems, while being flexible enough to enable its use with other forms of IO systems, such as simple imperative systems, threaded systems, or network servers.

3.2 Semantics

A rigorous and formal elucidation of the semantics of signal-function FRP remains unattempted, but there is a sufficient change to the practical semantics of signal-function FRP between previous signal-function systems and TimeFlies to warrant some description.

In previous systems such as Yampa, events were understood (and typed) as option-valued signals. This approach is undesirable for several reasons. The most pressing reason is that it prohibits push-based evaluation of events, because events are embedded in the samples of a signal and must be searched for.

Another concern is that this approach limits the event rate to the sampling rate. The rate of sampling should, at some level, not matter to the FRP system. Events which occur between sampling intervals are never observed by the system.

This concern drives the next concern. Events are not instantaneous in this formulation. If a signal is option valued, the sampling instant must fall within the interval where there is an event occurrence present for that event to be observed. If events are instantaneous, the probability of observing an event occurrence is zero.

Therefore, TimeFlies employs the N-Ary FRP type formulation to represent signals and events as distinct entities in the inputs and outputs of signal functions. This means we are now free to choose our representation of events,

and to separate it from the representation and evaluation of signals.

This freedom yields the additional ability to make events independent of the sampling interval altogether. The semantics of event handling in TimeFlies is that an event occurrence is responded to immediately, and does not wait for the next sampling instant. This allows events to be instantaneous, and further, allows multiple events to occur within a single sampling interval.

There are two tradeoffs intrinsic to this approach. The first is that events are only partially ordered temporally. There is no way to guarantee the order of observation of event occurrences occurring in the same sampling interval. Further, the precise time of an event occurrence cannot be observed, only the time of the last sample prior to the occurrence.

In return for giving up total ordering and precise observation of the timing of events, we obtain the ability to employ push-based evaluation for event occurrences, and the ability to non-deterministically merge event occurrences. When events being input to a non-deterministic merge have simultaneous occurrences, we simply select one arbitrarly to occur first. This does not violate any guarantee about time values, since they will both have the same time value in either case, and does not violate any guarantee about ordering, since no guarantee of their order is given.

A formal semantic description of signal function FRP would clarify the consequences of this decision somewhat, but is outside the scope of this thesis.

3.3 Types

In a strongly and statically typed functional language, types are a key part of an interface. Types provide a mechanism for describing and ensuring properties of the interface's components and about the systems created with these components.

3.3.1 Signal Vectors

In order to type signal functions, we must be able to describe their input and output. In most signal function systems, a signal function takes exactly one input and produces exactly one output. Multiple inputs or outputs are handled by making the output a tuple, and combinators which combine or split the inputs or outputs of a signal assume this. Events are represented at the type level as a particular type of signal, and at the value level as an option, either an event occurrence or not.

This method of typing excludes push-based evaluation at the outset. It is not possible to construct a "partial tuple" nor in general is it possible to construct only part of any type of value. Push-based evaluation depends on evaluating only that part of the system which is updated, which means evaluating only that part of the input which is updated.

In order to permit the construction of partial inputs and outputs, we make use of signal vectors. Signal vectors are uninhabited types which describe the input and output of a signal function. Singleton vectors are parameterized over the type carried by the signal or by event occurrences. The definition of

```
data SVEmpty    -- An empty signal vector component,
                -- neither event nor signal
data SVSignal a -- A signal, carrying values of type a
data SVEvent a  -- An event, whose occurrences carry values of type a
data SVAppend svLeft svRight -- The combination of the signal vectors
                             -- svLeft and svRight
```

Figure 3.1: Signal vector types

the signal vector type is shown in Figure 3.1.

Having an uninhabited signal vector type allows us to construct representations of inputs and outputs which are hidden from the user of the system, and are designed for partial representations.

3.3.2 Signal Functions

The type constructor for signal functions is shown in Figure 3.2. For the `init` parameter, only one possible instantiation is shown. The usefulness of this type parameter, along with another instantation which is hidden from users of the library, is discussed in the section on implementation of signal functions (Section 4.1).

Signal functions with signal vectors as input and output types form a Haskell `GArrow` [11]. Specifically, the signal function type constructor (with the initialization parameter fixed) forms the arrow type, the `SVAppend` type constructor forms the product type, and the `SVEmpty` type constructor forms the unit type.

The representation of signal functions is discussed in Section 4.1. Here

```
-- Signal functions
-- init: The initialization type for
-- the signal function, always NonInitialized
-- for exported signal functions
-- svIn: The input signal vector
-- svOut: The output signal vector
data SF init svIn svOut

data NonInitialized

type svIn :~> svOut = SF NonInitialized svIn svOut
type svLeft :^: svRight = SVAppend svLeft svRight
```

Figure 3.2: Signal function types.

it suffices to say that the use of GADTs permits the construction of values which make use of uninhabited types as instantiations of type parameters.

The type synonyms `:~>` and `:^:` are included for readability and are not crucial to the FRP system.

3.3.3 Evaluation Monad

A *monad* is a standard, composable abstraction for writing functions with a context, used in Haskell for IO [14, 15] among other tasks. A monad is simply a 1-arity type constructor together with two functions. The first function, `return`, takes a value of type `a` and produces a value of type `m a`, where `m` is the type constructor. The second, called `bind` and stylized in the Haskell standard library as the infix operator `(>>=)`, takes a value of type `m a` and a function from `a` to `m b` and produces a value of type `m b`. This allows a value to be operated on out of the context and a new context to be assigned.

A monad can have other primitives which manipulate the context in some way. For instance, the primtives in Haskell's `IO` monad produce actions which, when interpreted as part of the `main` action, produce some side-effect. The `State` monad provides `get` and `put` operations to work with a state value stored in the context.

Monad transformers [10] provide a means to combine the functionality of multiple monads. A monad transformer is a monad with an extra type parameter. This type parameter is instantiated with the type constructor of the underlying monad, and an extra operation `lift` is provided which converts values in the underlying monad to values in the monad transformer.

The evaluation monad is a monad transformer. This permits it to be used in conjunction with the `IO` monad (or any other monad) to describe how input is to be obtained for the signal function being evaluated, and how outputs are to be handled.

The evaluation monad, in addition to the standard monad operators, provides a means of *initializing* a signal function, and a means of translating the monadic value describing evaluation to a value in the underlying monad. This means, for instance, that we can obtain an action in the `IO` monad to evaluate a signal function.

The type of the evaluation monad must track the input type of the signal function. The monad's context stores a mapping from outputs to handling actions. An existential type can thus be used to "hide" the output type of the

```
-- A signal function's evaluations state
data SFEvalState svIn m
-- The evaluation monad
data SFEvalT svIn m a
```

Figure 3.3: Evaluation monad types.

signal function. However, inputs must come from external values, so the input type cannot be hidden. There are thus three type parameters to the monad's type constructor: the input signal vector, the type of the underlying monad, and the monadic type parameter. The type is shown in Figure 3.3.

3.4 Combinators

Signal functions are constructed from combinators, which are primitive signal functions and operations to combine these primitives. These combinators are grouped as basic signal functions, lifting operations for pure functions, routing, reactivity, feedback, and time dependence.

3.4.1 Basic Signal Functions

The basic signal functions (Figure 3.4) provide very simple operations. The `identity` signal function, as expected, simply copies its input to its output. The `constant` signal function produces the provided value as a signal at all times. The `never` signal function has an event output which never produces occurrences. The `asap` function produces an event occurrence with the given value at the first time step after it is switched into the network. The `after` function waits for the specified amount of time before producing the event

```
-- Pass the input unmodified to the output
identity :: sv :~> sv

-- Produce a signal which is at all times the supplied value
constant :: a -> SVEmpty :~> SVSignal a

-- An event with no occurrences
never    :: SVEmpty :~> SVEvent a

-- An event with one occurrence, as soon as possible after
-- the signal function is initialized
asap     :: a -> SVEmpty :~> SVEvent a

-- An event after the specified amount of time has elapsed.
after    :: Double -> a -> SVEmpty :~> SVEvent a
```

Figure 3.4: Basic signal functions.

occurrence.

With the exception of `identity`, all of the basic signal functions have empty inputs. This allows these signal functions to be used to insert values into the network which are known when the signal function is created, without having to route those values from an input.

3.4.2 Lifting Pure Functions

Two combinators are provided to lift pure functions to signal functions (Figure 3.5). The `pureSignal` combinator applies the pure function to a signal at every sample point. The `pureEvent` combinator applies the function to each occurrence of an input event.

```
-- Apply the given function to a signal at all points in time
pureSignalTransformer :: (a -> b) -> SVSignal a :~> SVSignal b

-- Apply the given function to each event occurrence
pureEventTransformer  :: (a -> b) -> SVEvent a :~> SVEvent b
```

Figure 3.5: Lifting pure functions.

3.4.3 Routing

The routing combinators are used to combine signal functions, and to re-arrange signal vectors in order to connect signal functions. The routing combinators are shown in Figure 3.6.

Only those combinators which modify or combine signal functions (`(>>>)`, `first`, `second`) are reactive, and then only if they inherit their reactivity from the signal function(s) they modify. The rest do not react to or modify the input in any way, except to re-arrange it, copy it, or discard it altogether.

3.4.4 Reactivity

Reactivity is introduced by means of the `switch` combinator (Figure 3.7). The design of this combinator allows modular packaging of reactivity. A signal function can determine autonomously when to replace itself, based only on its input and state, by emitting an event occurrence carrying its replacement. The combinator consumes and hides the event carrying the replacement signal function, so the reactivity is not exposed by the resulting reactive signal function.

There are other formulations of a reactive combinator which may be

```
-- Use the output of one signal function as the input for another
(>>>) :: (svIn :~> svBetween) -> (svBetween :~> svOut) -> svIn :~> svOut

-- Pass through the right side of the input unchanged
first :: (svIn :~> svOut) -> (svIn :^: sv) :~> (svOut :^: sv)

-- Pass through the left side of the input unchanged
second :: (svIn :~> svOut) -> (sv :^: svIn) :~> (sv :^: svOut)

-- Swap the left and right sides
swap :: (svLeft :^: svRight) :~> (svRight :^: svLeft)

-- Duplicate the input
copy :: sv :~> (sv :^: sv)

-- Ignore the input
ignore :: sv :~> svEmpty

-- Remove an empty vector on the left
cancelLeft :: (SVEmpty :^: sv) :~> sv

-- Add an empty vector on the left
uncancelLeft :: sv :~> (SVEmpty :^: sv)

-- Remove an empty vector on the right
cancelRight :: (sv :^: SVEmpty) :~> sv

-- Add an empty vector on the right
uncancelRight :: sv :~> (sv :^: SVEmpty)

-- Make right-associative
associate :: ((sv1 :^: sv2) :^: sv3) :~> (sv1 :^: (sv2 :^: sv3))

-- Make left-associative
unassociate :: (sv1 :^: (sv2 :^: sv3)) :~> ((sv1 :^: sv2) :^: sv3)
```

Figure 3.6: Routing combinators.

```
switch ::    (svIn :~> (svOut :^: SVEvent (svIn :~> svOut)))
          -> svIn :~> svOut
```

Figure 3.7: Combinator for reactivity.

```
-- Alternate version of switch,
-- implemented in terms of supplied version
switch_gen ::    (svIn :~> (svOut :^: SVEvent a))
              -> (a -> svIn :~> svOut)
              -> svIn :~> svOut
switch_gen sf f =
  switch (sf >>> second (pureEventTransformer f))

-- Supplied version in terms of alternate version
switch ::    (svIn :~> (svOut :^: SVEvent (svIn :~> svOut)))
          -> svIn :~> svOut
switch sf = switch_gen sf id

-- Repeated switch, which takes replacement signal functions
-- externally.
rswitch ::    (svIn :~> svOut)
           -> (svIn :^: SVEvent (svIn :~> svOut)) :~> svOut
rswitch sf =
  switch (first sf >>> second (pureEventTransformer rswitch))
```

Figure 3.8: Alternate combinators for reactivity.

implemented using the one supplied. These are shown in Figure 3.8 and may be provided in a future version of the TimeFlies library.

3.4.5 Feedback

It is often useful for a signal function to receive a portion of its own output as input. This is especially useful when we have two signal functions which we would like to mutually interact. We cannot just serially compose

```
loop ::    ((svIn :^: svLoop) :~> (svOut :^: svLoop))
        -> svIn :~> svOut
```

Figure 3.9: Feedback combinator.

them, we must also bring the output of the second back around to the first. Many signal-processing algorithms also depend on feedback. The combinator which provides this ability is shown in Figure 3.9.

This combinator provides decoupling for signals (the input signal is the output signal at the previous time-step) but not events (event occurrences are supplied to the combinator immediately). This means that the programmer has the responsibility to ensure that feedback does not generate an infinite sequence of events in a single time-step.

3.4.6 Event-Specific Combinators

Several combinators are provided for manipulating, suppressing, and generating events. Output events, in these combinators, are triggered by input events.

Each of the combinators has an option variant and a list variant. The option variant produces an output event occurrence whenever the application of the supplied function to the input event produces a value. The list version produces an event occurrence for each of the elements of the output list. The combinators are shown in Figure 3.10.

The filter combinators are stateless, and thus apply the function to

```
-- Apply the function to each input occurrence,
-- and produce an occurrence for each Just.
filter :: (a -> Maybe b) -> SVEvent a :~> SVEvent b

-- Apply the function to each input occurrence,
-- and produce an occurrence for each list element
filterList :: (a -> [b]) -> SVEvent a :~> SVEvent b

-- Apply the function to the stored accumulator
-- and the event occurrence, replacing the accumulator
-- and possibly outputting an occurrence
accumulate ::    (a -> b -> (Maybe c, a))
              -> a
              -> SVEvent b :~> SVEvent c

-- Apply the function to the stored accumulator
-- and the event occurrence, replacing the
-- accumulator and outputting an event occurrence
-- for each element of the list
accumulateList ::    (a -> b -> ([c]))
                  -> a
                  -> SVEvent b :~> SVEvent c
```

Figure 3.10: Event-specific combinators.

only the new input value. They are useful for suppressing events, as well as for extracting one of multiple cases of a datatype. For instance, a splitter for events carrying `Either`-valued occurrences could be written as:

```
getLeft :: Either a b -> Maybe a
getLeft (Left x) = Just x
getLeft _ = Nothing

getRight :: Either a b -> Maybe b
getRight (Right x) = Just x
getRight _ = Nothing

split :: SVEvent (Either a b) :~> (SVEvent a :^: SVEvent b)
split = copy >>> first (filter getLeft) >>> second (filter getRight)
```

The accumulate combinators are stateful, applying the supplied function to both the input value and an accumulator. This function has two results: the option or list of output event occurrence values, and the new value for the accumulator.

The accumulator is useful when responses to multiple event occurrences (from one or more sources) must be coordinated. For instance, in the benchmark application (see Chapter 5, a table is maintained that allows knowledge from previous event occurrences (packets from a network switch) to be used in deciding where the present packet ought to go.

3.4.7 Joining

The joining combinators provide the ability to combine two event streams, two signals, or a signal and an event stream. These combinators are shown in

```
union          :: (SVEvent a :^: SVEvent a) :~> SVEvent a
combineSignals :: (a -> b -> c) -> (SVEvent a :^: SVEvent b) :~> SVEvent c
capture        :: (SVSignal a :^: SVEvent b) :~> SVEvent (b, a)
```

Figure 3.11: Joining combinators.

Figure 3.11

The `union` combinator is a non-deterministic merge of event streams. Any event which occurs on either input will occur on the output. For simultaneous event occurrences, the order of occurrence is not guaranteed, but the occurrence itself is. This construct is also guaranteed to respect the relationship of event occurrences to sampling intervals.

The `combineSignals` combinator applies a binary function pointwise to two signals, and produces the result of this application as a third signal.

The `capture` combinator adds the last-sampled value of a signal at the time of an event occurrence to that event occurrence.

These three combinators together provide the ability to combine elements of signal vectors. By combining these combinators, arbitrary signal vectors can be reduced.

3.5 Evaluation

The evaluation interface provides a modified state monad which holds a signal function, together with some additional information, as its state (shown in Figure 3.12. Rather than monadic instructions to put and get the state,

the monad provides instructions to trigger an input event, update an input signal, and trigger sampling of signals in the signal function. Additional state includes the current set of modifications to the input signals (since the last sample) and a set of handlers which actuate effects based on output events or changes to the output signal.

In order to initialize the state, the user must supply a set of handlers, the signal function to evaluate, and initial values for all of the signal inputs (Figure 3.13).

This state can then be passed to a monadic action which will supply input to the signal function. Inputs are constructed using a simple interface with functions to construct sample updates and event occurrences, and to specify their place in the vector (Figure 3.14).

The `SFEvalT` monad is actually a monad transformer, that is, it is parameterized over an underlying monad whose actions may be lifted to `SFEvalT`. In the usual case, this will be the `IO` monad.

`SFEvalT` actions are constructed using combinators to push events, update inputs, and step time, as well as actions lifted from the underlying monad (used to obtain these inputs). An action in the underlying monad which produces the result and a new state is obtained with the `runSFEvalT` function. These combinators are shown in Figure 3.15.

```
-- A vector of handlers for outputs
data SVHandler out sv

-- A dummy handler for an empty output
emptyHandler    :: SVHandler out SVEmpty

-- A handler for an updated signal sample
signalHandler   :: (a -> out) -> SVHandler out (SVSignal a)

-- A handler for an event occurrence
eventHandler    :: (a -> out) -> SVHandler out (SVEvent a)

-- Combine handlers for a vector
combineHandlers ::    SVHandler out svLeft
                   -> SVHandler out svRight
                   -> SVHandler out (svLeft :^: svRight)

-- The state maintained when evaluating a signal function
data SFEvalState m svIn svOut

-- Create the initial state for evaluating a signal function
initSFEval ::    SVHandler (m ()) svOut
              -> SVSample svIn
              -> Double
              -> (svIn :~> svOut)
              -> SFEvalState m svIn svOut
```

Figure 3.12: State maintained when evaluating a signal function

```
-- A sample for all leaves of a signal vector
data SVSample sv

-- Create a sample for a signal leaf
sample          :: a -> SVSample (SVSignal a)

-- A dummy sample for an event leaf
sampleEvt       :: SVSample (SVEvent a)

-- A dummy sample for an empty leaf
sampleNothing   :: SVSample SVEmpty

-- Combine two samples
combineSamples  ::    SVSample svLeft
                   -> SVSample svRight
                   -> SVSample (svLeft :^: svRight)
```

Figure 3.13: Data type for initial input

```
-- Class to overload left and right functions
class SVRoutable r where
  svLeft          :: r svLeft -> r (svLeft :^: svRight)
  svRight         :: r svRight -> r (svLeft :^: svRight)

-- An input event occurrence
data SVEventInput sv
instance SVRoutable SVEventInput sv

-- An updated sample for a signal
data SVSignalUpdate sv
instance SVRoutable SVSignalUpdate sv

-- Create an event occurrence
svOcc           :: a -> SVEventInput (SVEvent a)

-- Create an updated sample
svSig           :: a -> SVSignalUpdate (SVSignal a)
```

Figure 3.14: Data types for ongoing input.

```
-- The evaluation monad
data SFEvalT svIn svOut m a
instance MonadTrans (SFEvalT svIn svOut)
instance (Monad m) => Monad (SFEvalT svIn svOut m)
instance (Functor m) => Functor (SFEvalT svIn svOut m)
instance (Monad m, Functor m) => Applicative (SFEvalT svIn svOut m)
instance (MonadIO m) => MonadIO (SVEvalT svIn svOut m)

-- Obtain an action in the underlying monad
-- from an SFEvalT and a new state.
runSFEvalT ::    SFEvalT svIn svOut m a
              -> SFEvalState m svIn svOut
              -> m (a, SFEvalState m svIn svOut)

-- Push an event occurrence.
push :: (Monad m) => SVEventInput svIn -> SFEvalT svIn svOut m ()

-- Update the value of an input signal sample
-- (not immediately observed)
update :: (Monad m) => SVEventInput svIn -> SFEvalT svIn svOut m ()

-- Step forward in time, observing the updated signal values
step :: (Monad m) => Double -> SFEvalT svIn svOut m ()
```

Figure 3.15: Evaluation combinators

Chapter 4

Implementation

Having established the design and semantics of the reactive system, I present a purely functional implementation. For this implementation, I will make significant use of extensions to the Haskell type system. These extensions provide mechanisms for expressing abstractions not easily expressible in a standard Hindley-Milner type system, while maintaining the type soundness of Haskell.

4.1 Signal Functions

The design of signal functions specifies a family of types for the inputs and outputs of signal functions. Signal functions are not functions in the purest sense, however. They are not mappings from a single instance of their input type to a single instance of their output type. They must be implemented with respect to the temporal semantics of their inputs and outputs.

We therefore start by creating a set of concrete datatypes for the inputs and outputs of signal functions. These datatypes will be parameterized by the input and output types of the signal function, and will not be exposed to the user of the library. Rather, they will specify how data is represented during

the temporal evaluation of signal functions.

In order to create these concrete representations, we employ the Haskell type system extension known as Generalized Algebraic Datatypes, which we divert briefly to explain before exploring its use in the implementation.

We then describe how signal functions are implemented using these concrete types, along with higher-order functions and continuations.

4.1.1 Generalized Algebraic Datatypes

The first consideration is how to make use of the information encoded in the signal vectors used as the input and output types of signal functions. In order to parameterize over these types, we turn to an extension of the Haskell type system known as Generalized Algebraic Datatypes (GADTs).

Algebraic Datatypes (ADTs) are the means by which new primitive types are introduced into Haskell programs. For instance, the following datatypes are declared in the Haskell Prelude (the Haskell module which is in scope for every Haskell program):

```
data Bool    = True   | False
data Maybe a = Just a | Nothing
```

The first example is simply a set of alternatives, representing the values in Boolean algebras. These alternatives may be pattern-matched in a Haskell function either at the top level:

```
boolToString :: Bool -> String
boolToString True  = "Boolean True"
boolToString False = "Boolean False"
```

Or in a `case` expression:

```
boolToString :: Bool -> String
boolToString b = "Boolean " ++ case b of
                                 True  -> "True"
                                 False -> "False"
```

The second example is a parameterized ADT, or type constructor. The actual type is constructed by filling in a concrete type for the variable `a`. Polymorphic functions can fill in quantified variables for this type or a component of it. However, there are no restrictions on what types may be used. In the `Just` data constructor, the use of the type variable indicates a parameter to the data constructor which must take the type filled in for `a`.

This allows the `Maybe` type to generalize the pattern of an optional value. The type `Maybe Int` represents 0 or 1 integer values, and the type `Maybe String` represents 0 or 1 strings. In order to consume values of these optional types, we can pattern match[1]:

```
stringMaybe :: String -> Maybe String -> String
stringMaybe s Nothing = s
stringMaybe _ (Just s)  = s
```

But since we can fill in a type variable for the parameter of the Maybe type constructor, we can generalize this function:

```
maybe :: a -> Maybe a -> a
maybe x Nothing  = x
maybe _ (Just x) = x
```

[1]The underscore character _ represents an input to the pattern which we disregard. It is called a wildcard.

GADTs permit us to specify what types fill in the type parameters for specific constructors. For instance, if we wish to build a tiny expression language, we can write:

```
data Exp a = Const a | Plus (Exp a) (Exp a)
```

Let us assume for the moment that the Haskell addition function is typed:

```
(+) :: Int -> Int -> Int
```

The type actually involves a typeclass constraint, but the point is that the function's type is not parametric.

An attempt to write an evaluation function for our expression type is:

```
eval :: Exp a -> a
eval (Const x) = x
eval (Plus x y) = eval x + eval y
```

But this will not typecheck, since we cannot assume that `a` is `Int`, so the output type of `(+)` does not match the output type of our function, and the input types to `(+)` do not match the input types to our function.

Let's try a slightly modified ADT:

```
data Exp a = Const a | Plus (Exp Int) (Exp Int)
```

The code for our our evaluator is the same. Now the input types match, but the output type still does not.

Here is our expression type as a GADT:

```
data Exp a where
  Const  ::  a -> Exp a
  Plus   :: Exp Int -> Exp Int -> Exp Int
```

Now our evaluation function will typecheck. But why? The type of the `Plus` constructor has had its *output* type restricted to `Int`, and this allows us to assume, in the case where `Plus` is matched, that the type `a` has been restricted to `Int`. Put another way, if a value of an `Exp` type is not an `Exp Int`, then the `Plus` case will not occur.

This capacity to constrain the output types of data constructors, and thus, to constrain the types of expressions in the scope of pattern matches of these data constructors, is called *type refinement*. We will make use of this ability to parameterize concrete datatypes over abstract type structures, rather than to permit typechecking in specific cases, but the principle is the same.

4.1.2 Concrete Representations of Signal Vectors

In Chapter 3 we present signal vectors as a set of types. In order to be completely principled, we should isolate these types into their own `kind` (a sort of type of types), however, the Haskell extension for this was far from stable at the time this system was created.

The types are therefore expressed in the system exactly as they were described in Chapter 3. (To refresh, see Figure 3.1.) The striking observation about these types is that they have *no data constructors*. There are no values

```
data SVSample sv where
  SVSample      ::    a
                   -> SVSample (SVSignal a)
  SVSampleEvent ::    SVSample (SVEvent a)
  SVSampleEmpty ::    SVSample SVEmpty
  SVSampleBoth  ::    SVSample svLeft
                   -> SVSample svRight
                   -> SVSample (SVAppend svLeft svRight)
```

Figure 4.1: Datatype for signal samples.

which take these types.

Instead, we will create concrete representations which are parameterized over these types. These concrete representations will be expressed as GADTs, allowing each data constructor of the representation to fill in a specific signal vector type for the parameter of the representation.

To get our feet wet, we create a representation which carries a value for every `SVSignal` leaf of a signal vector. In order to do this, we restrict each of our constructors to a single signal vector type. This ensures that the only way to represent a sample leaf is with the `SVSample` constructor, which carries a value of the appropriate type. The datatype is shown in Figure 4.1.

So we now have a representation for the signal components of a signal vector. What about the event components? We want to represent event occurrences, each of which will correspond to at most one event in the vector. So a different representation is called for. In this case, there will be only three constructors. One constructor will represent an event leaf, and the other will represent a single value on the left or right side of the node (`SVAppend`), ignor-

```
data SVOccurrence sv where
  SVOccurrence ::    a
                  -> SVOccurrence (SVEvent a)
  SVOccLeft    ::    SVOccurrence svLeft
                  -> SVOccurrence (SVAppend svLeft svRight)
  SVOccRight   ::    SVOccurrence svRight
                  -> SVOccurrence (SVAppend svLeft svRight)
```

Figure 4.2: Datatype for event occurrences.

ing all of the type structure on the other side. This representation describes a path from the root of the signal vector, terminating at an event leaf with a value.

By pattern matching on the path constructors, we can determine which subvector of a signal vector an event occurrence belongs to, repeatedly refining it until we determine which event in the vector the occurrence corresponds to. The datatype for occurrences is shown in Figure 4.2.

We add one more representation for signals, in order to avoid unnecessary representations of the values of all signals when not all signals have changed their values. This representation allows us to represent the values of zero or more of the signals in a signal vector. To accomplish this, we replace the individual constructors for the `SVEmpty` and `SVEvent` leaves with a single, unconstrained constructor. This constructor can represent an arbitrary signal vector. We can use the constructor for signal vector nodes and the constructor for sample leaves to represent the updated values, while filling in the unchanged portions of the signal vector with this general constructor. This datatype is shown in Figure 4.3.

```
data SVDelta sv where
  SVDeltaSignal  ::    a
                    -> SVDelta (SVSignal a)
  SVDeltaNothing ::    SVDelta sv
  SVDeltaBoth    ::    SVDelta svLeft
                    -> SVDelta svRight
                    -> SVDelta (SVAppend svLeft svRight)
```

Figure 4.3: Datatype for signal updates.

4.1.3 Signal Function Implementation Structure

We now have concrete datatypes for an implementation to operate on. Our next task is to represent transformers of temporal data, which themselves may change with time. The common approach to this task is sampling, in which a program repeatedly checks for updated information, evaluates it, updates some state, and produces an output. This is the essence of pull-based evaluation.

Another strategy is notification, in which the program exposes an interface which the source of updated information may invoke. This is a repeated entry point to the program, which causes the program to perform the same tasks listed above, namely, evaluate the updated information, update state, and produce output. The strategy of notification as opposed to repeated checking is the essence of push-based evaluation.

Signal functions are declarative objects, and not running processes. They have no way to invoke sampling themselves. They can, however, expose separate interfaces for when sampling is invoked, and when they are notified of

an event occurrence. This creates two control paths through a signal function. One of these control paths is intended to be invoked regularly and frequently with updates to the time and sample values, and the other is intended to be invoked only when an event occurs. The benefit of separating these control paths is that events are no longer defined in terms of sampling intervals, and need not even be considered in sampling, except when they are generated by a condition on a sample. On the other hand, events can be responded to even if the time has not yet come for another sample, and multiple events can be responded to in a single sampling interval.

We represent signal functions as a datatype with two constructors. The signal functions that a user will compose are non-initialized signal functions. They must be provided with an initial set of input signals (corresponding to time zero). When provided with this input, they produce their time-zero output, and an initialized signal function. The datatype is shown in Figure 4.4

Initialized signal functions carry two continuations. The first continuation takes a time differential and a set of signal updates, and returns a set of signal updates, a collection of event occurrences, and a new initialized signal function of the same type. This is the continuation called when sampling.

The second continuation takes an event occurrence, and returns a collection of event occurrences and a new signal function of the same type. This continuation is only called when there is an event occurrence to be input to the signal function.

```
data Initialized

data NonInitialized

data SF init svIn svOut where
  SF     ::    (SVSample svIn
                  -> (SVSample svOut,
                      SF Initialized svIn svOut))
             -> SF NonInitialized svIn svOut
  SFInit ::    (Double
                  -> SVDelta svIn
                  -> (SVDelta svOut,
                      [SVOccurrence svOut],
                      SF Initialized svIn svOut))
             -> (SVOccurrence svIn
                   -> ([SVOccurrence svOut],
                       SF Initialized svIn svOut))
             -> SF Initialized svIn svOut
```

Figure 4.4: Datatype and empty types for signal functions.

Note that each of these continuations uses one or more of the concrete representations of signal vectors, and applies the type constructor for the representation to the input or output signal vector for the signal function.

4.1.4 Implementation of Signal Function Combinators

Having specified a datatype for signal functions, we must now provide combinators which produce signal functions of this type. Each combinator's implementation must specify how it is initialized, how it samples its input, and how it responds to event occurrences.

We will not detail every combinator here, but we will discuss each of the implementation challenges encountered.

The simplest combinator is the `identity` signal function. This signal function simply passes all of its inputs along as outputs. The initialization function simply passes along the received sample and outputs the initialized version of the signal function. The initialized version of the input is similar, but is self-referential. It replaces itself with itself after every signal function or event. The implementation is shown in Figure 4.5.

In order for our primitive signal functions to be useful, we need a means of composing them. Serial composition creates one signal function from two, by using the output of one as the input of the other. The serial composition combinator is styled `(>>>)`. The implementation of this operator is one place where the advantage of responding to events independently from signal samples becomes clear. The implementation is shown in Figure 4.6.

```
identity :: sv :~> sv
identity =
  SF (\initSample -> (initSample, identityInit))

identityInit :: SF Initialized sv sv
identityInit =
  SFInit (\dt sigDelta -> (sigDelta, [], identityInit))
         (\evtOcc -> ([evtOcc], identityInit))
```

Figure 4.5: Implementation of the `identity` combinator.

The implementation of the non-initialized version is simple. It takes two non-initialized signal functions, and returns the sample output of applying the second to the sample output of the first, along with an initialized signal function constructed by applying the initialized version of the combinator to the initialized signal functions given by the inputs.

The initialized version constructs two continuations. The first receives a time and signal update input, and applies the signal continuation of the first signal function to it. The output is fed, along with the time update, into the second signal function, first the signal update, and then events. The output of the second, along with the accumulated updated signal functions with the initialized serial composition recursively applied, is returned as output.

The event continuation is similar but simpler, as there is no need to deal with signal updates.

The switch combinator is the means of introducing reactivity into a signal function. This combinator allows a signal function to replace itself by producing an event occurrence. The switch combinator stores the full sample

```
(>>>) ::    (svIn :~> svBetween)
         -> (svBetween :~> svOut)
         -> (svIn :~> svOut)
(SF sigSampleF1) >>> (SF sigSampleF2) =
  SF (\sigSample -> let (sigSample', sfInit1) = sigSampleF1 sigSample
                        (sigSample'', sfInit2) = sigSampleF2 sigSample'
                    in (sigSample'', composeInit sfInit1 sfInit2))

composeInit ::     SF Initialized svIn svBetween
                -> SF Initialized svBetween svOut
                -> SF Initialized svIn svOut
composeInit (SFInit dtCont1 inputCont1) sf2@(SFInit dtCont2 inputCont2) =
  SFInit
    (\dt sigDelta ->
       let (sf1MemOutput, sf1EvtOutputs, sf1New) = dtCont1 dt sigDelta
           (sf2MemOutput, sf2EvtOutputs, sf2New) = dtCont2 dt sf1MemOutput
           (sf2EvtEvtOutputs, sf2Newest) = applySF sf2New sf1EvtOutputs
       in (sf2MemOutput,
           sf2EvtOutputs ++ sf2EvtEvtOutputs,
           composeInit sf1New sf2Newest)
    )
    (\evtOcc ->
      let (sf1Outputs, newSf1) = inputCont1 evtOcc
          (sf2FoldOutputs, newSf2) = applySF sf2 sf1Outputs
      in (sf2FoldOutputs, composeInit newSf1 newSf2)
    )

applySF ::    SF Initialized svIn svOut
           -> [SVOccurrence svIn]
           -> ([SVOccurrence svOut],
               SF Initialized svIn svOut)
applySF sf indices =
  foldr (\evtOcc (changes, SFInit _ changeCont) ->
           let (newChanges, nextSF) = changeCont evtOcc
               in (newChanges ++ changes, nextSF))
        ([], sf)
        indices
```

Figure 4.6: Implementation of serial composition.

for its input vector (which is identical to the input vector of the supplied signal function) to initialize the new signal function. This also demands that it continue wrapping the signal function until the next sampling interval, so that it can actuate the output of initialization as a replacement signal. This has some performance implications, which are discussed in Chapter 5. The implementation is shown in Figure 4.7.

Most of the routing combinators are simple to implement. The only task is to add remove, or replace routing constructors on signal updates and event occurrences. Since these signal functions are stateless and primitive, they can simply return themselves as their replacements. The `swap` combinator is shown as an example in Figure 4.8

The `first` and `second` combinators are not quite as simple, as they must transform a provided signal function. For signal changes, they must split the set of changes into those which will be passed to the signal function and those which will be simply passed along to the output, and then recombine them on the other side. For event occurrences, the occurrence must be pattern-matched to determine whether to call the event continuation from the provided signal function or passed through, and output event occurrences must have the suitable routing constructor re-applied. In any case, when a continuation has been applied, the combinator must be recursively applied to the new signal function. The implementation of `first` is shown in Figure 4.9. The implementation of `second` is the obvious reversal of this implementation.

The looping feedback combinator is particularly tricky. As it is cur-

```
switch :: (svIn :~> (svOut :^: SVEvent (svIn :~> svOut))) -> svIn :~> svOut
switch (SF sigSampleF) =
  SF (\sigSample -> let (sigSampleSF, sf) = sigSampleF sigSample
                        (sigSampleOut, _) = splitSample sigSampleSF
                    in (sigSampleOut, switchInit sigSample sf))

-- Pre-occurrence
switchInit ::    SVSample svIn
              -> SF Initialized svIn
                    (SVAppend svOut
                     (SVEvent (SF NonInitialized svIn svOut)))
              -> SF Initialized svIn svOut
switchInit inputSample sf@(SFInit timeCont changeCont) =
  SFInit (\dt sigDelta ->
            let (sigDeltaSF, occsSF, newSF) = timeCont dt sigDelta
                newInputSample = updateSample inputSample sigDelta
                (sigDeltaOut, _) = splitDelta sigDeltaSF
                (occsOut, occsSwitch) = splitOccurrences occsSF
                (outputDelta, nextSF) =
                  maybe (sigDeltaOut, switchInit newInputSample newSF)
                    (\(SF sigSampleF) ->
                       let (outputSample, switchSF) =
                             sigSampleF newInputSample
                           outputDelta = sampleDelta outputSample
                       in (outputDelta, switchSF))
                    $ occurrenceListToMaybe occsSwitch
            in (outputDelta, occsOut, nextSF))
         (\evtOcc ->
            let (occsSF, newSF) = changeCont evtOcc
                (occsOut, occsSwitch) = splitOccurrences occsSF
                nextSF = maybe (switchInit inputSample newSF)
                               (\(SF sigSampleF) ->
                                  let (sampleOut, switchSF) =
                                        sigSampleF inputSample
                                  in switchWait sampleOut switchSF)
                           $ occurrenceListToMaybe occsSwitch
            in (occsOut, nextSF))

switchWait ::    SVSample svOut
              -> SF Initialized svIn svOut
              -> SF Initialized svIn svOut
switchWait outputSample sf@(SFInit timeCont changeCont) =
  SFInit (\dt sigDelta -> let (sigDeltaOut, outOccs, newSF) =
                                timeCont dt sigDelta
                          in (sampleDelta $
                                updateSample outputSample sigDeltaOut,
                              outOccs,
                              newSF))
         \evtOcc -> let (outOccs, newSF) = changeCont evtOcc
                    in (outOccs, switchWait outputSample newSF))
```

```
-- T.swap is imported from Data.Tuple
swap :: (sv1 :^: sv2) :~> (sv2 :^: sv1)
swap =
  SF ((, swapInit) .
      uncurry combineSamples .
      T.swap . splitSample)

swapInit :: SF Initialized (SVAppend sv1 sv2) (SVAppend sv2 sv1)
swapInit =
  SFInit (flip (const .
                (, [], swapInit) .
                uncurry combineDeltas .
                T.swap . splitDelta))
         (\evtOcc ->
            (case chooseOccurrence evtOcc of
               Left lOcc  -> [occRight lOcc]
               Right rOcc -> [occLeft rOcc], swapInit))
```

Figure 4.8: Swap routing combinator implementation.

```
first :: (svIn :~> svOut) -> (svIn :^: sv) :~> (svOut :^: sv)
first (SF sigSampleF) =
  SF (\sigSample -> let (leftSample, rightSample) = splitSample sigSample
                        (leftSampleOut, sf) = sigSampleF leftSample
                    in (combineSamples leftSampleOut rightSample,
                        firstInit sf))

firstInit ::    SF Initialized svIn svOut
             -> SF Initialized (SVAppend svIn sv) (SVAppend svOut sv)
firstInit (SFInit timeCont inputCont) =
  let firstInitSF =
    SFInit (\dt sigDelta ->
               let (input, rightOutput) = splitDelta sigDelta
                   (sigDeltaOutput, evtOutput, sf1New) = timeCont dt input
               in (combineDeltas sigDeltaOutput rightOutput,
                   map occLeft evtOutput,
                   firstInit sf1New))
           (\evtOcc ->
              case chooseOccurrence evtOcc of
                Left  lChange -> let (changes, sf) = inputCont lChange
                                 in (map occLeft changes, firstInit sf)
                Right rChange -> ([occRight rChange], firstInitSF))
  in firstInitSF
```

Figure 4.9: Implemenation of `first` combinator.

rently implemented, the initial sample for the right side of the input signal vector to the supplied function is the right side of the output sample. This is acceptable, given Haskell's non-strict evaluation strategy, but it is necessary that the right side of the signal function's output not be immediately dependent on its input. The feedback combinator makes use of Haskell's lazy evaluation to feed events back into the combinator, and stores signal updates until the next sample. Signal samples are thus automatically decoupled after initialization. The implementation, which makes extensive use of the recursive nature of Haskell's `let` construct, is shown in Figure 4.10.

The `filter` combinators are simple to implement. Their sampling continuation is superfluous, and the event continuation merely applies the supplied function, and constructs an output list based on the result.

The `accumulate` combinators are implemented in terms of the `filter` and `switch` combinators, as shown in Figure 4.11.

The implementation of the joining combinators is simple. The `union` combinator simply passes along every event occurrence it receives on either input, stripping off the left and right combinator. This is acceptable since we do not insist on a total ordering of events, or an event time resolution greater than the sampling rate. The `combineSignals` combinator maintains the values of both signals, and applies the combination function whenever one is updated. The `capture` combinator maintains the input signal value, and adds it to each event occurrence.

```
loop :: (svIn :^: svLoop) :~> (svOut :^: svLoop) -> svIn :~> svOut
loop (SF sigSampleF) =
  SF (\sigSample ->
        let (sigSampleOut, sfInit) =
              sigSampleF (combineSamples sigSample sigSampleOutRight)
            (sigSampleOutLeft, sigSampleOutRight) =
              splitSample sigSampleOut
        in (sigSampleOutLeft, loopInit deltaNothing sfInit))

loopInit ::    SVDelta svLoop
            -> SF Initialized (SVAppend svIn svLoop) (SVAppend svOut svLoop)
            -> SF Initialized svIn svOut
loopInit loopDelta (SFInit timeCont changeCont) =
  SFInit (\dt sigDelta ->
            let (sigDeltaOut, occsOut, newSF) =
                  timeCont dt (combineDeltas sigDelta loopDelta)
                (occsOutput, loopOccs) = splitOccurrences occsOut
                (occsOutput', loopOccs') = splitOccurrences foldOccs
                (foldOccs, newSF') =
                  applySF newSF (map occRight (loopOccs ++ loopOccs'))
                (outputDelta, newLoopDelta) = splitDelta sigDeltaOut
            in (outputDelta,
                occsOutput ++ occsOutput',
                loopInit newLoopDelta newSF'))
         (\change -> let (occsOut, newSF) = changeCont $ occLeft change
                         (occsOutput, loopOccs) =
                           splitOccurrences occsOut
                         (occsOutput', loopOccs') =
                           splitOccurrences foldOccs
                         (foldOccs, newSF') =
                           applySF newSF
                             (map occRight (loopOccs ++ loopOccs'))
                     in (occsOutput ++ occsOutput',
                         loopInit loopDelta newSF'))
```

Figure 4.10: Implementation of feedback.

```
-- | Accumulate over event occurrences
accumulate :: (a -> b -> (Maybe c, a)) -> a -> SVEvent b :~> SVEvent c
accumulate f a = acc a
  where acc a = switch (pureEventTransformer (f a) >>>
                        copy >>>
                        first (pureEventTransformer fst >>> filter id) >>>
                        second (pureEventTransformer (acc . snd)))

-- | Accumulate over event occurrences, with lists of event outputs
accumulateList :: (a -> b -> ([c], a)) -> a -> SVEvent b :~> SVEvent c
accumulateList f a = acc a
  where acc a = switch (pureEventTransformer (f a) >>>
                        copy >>>
                        first (pureEventTransformer fst >>> filterList id) >>>
                        second (pureEventTransformer (acc . snd)))
```

Figure 4.11: Implementation of event accumulators.

Time dependence is introduced by the `time`, `delay`, and `integrate` combinators. The time combinator simply sums the time updates and provides the sum as a signal output. The `delay` combinator keeps a table of events which have come in, along with their schedule occurrence time, and produces them as output when time advances far enough. The integrate combinator performs rectangle-rule integration on signal samples with respect to time.

The implementation strategy leaves room for optimizations. In particular, an additional constructor for time-independent signal functions would allow portions of a signal function to forgo evaluation during time steps unless they had signal updates. Optimizations in the style of Yampa, observed by keeping an updated for the signal function and pattern-matching on it when switching, might further improve performance. Nevertheless, this im-

plementation performs quite well as it currently exists, as we demonstrate in Chapter **??**.

4.2 Evaluation Interface

The evaluation interface provides the means of evaluating a signal function with inputs and producing effects in response to the signal function's outputs. We would like to produce a set of constructs that interacts well with Haskell's system for external IO.

The evaluation interface translates between signal functions and the standard Haskell construct for sequencing effects and external inputs, namely, *monads* [15]. The inspiration for monads is drawn from the rather esoteric domain of category theory, but the concept as applied to programming languages is actually rather simple.

In a programming language with type constructors, a monad is simply a 1-arity type constructor together with two operations. We shall call this type constructor m. The first operation takes a value of any type (call the type a) and produces a value of type $m\,a$ (m applied to a). The second operation takes a value of type $m\,a$ and a function from a to $m\,b$, and produces a $m\,b$. This means that this second operation, monadic application, applies the function to the value in the context of the monad.

For our purposes, there are two interesting properties of monads. One is that for a specific type constructor, opaque primitives may be defined which

can be used, along with return, to build functions for the second argument of the monadic application operation. In our case, we would like operations to push an event to a signal function, to update the signal components of the signal function, and to increment the time and sample the signal function.

This property is what enables monads to be used for input and output. A Haskell `main` program is an IO action. An IO action is a value of the monadic type IO. This action is comprised of a primitive action (either a `return` or a true IO primitive), possibly together with a function from the output of that action to another IO action. Note that values with the IO type may appear anywhere in a Haskell program, but they are only execute when they are returned as a part of the `main` IO program.

The Haskell typeclass for monads is defined:

```
class Monad m where
  return :: a -> m a
  -- | Infix operator for monadic application
  (>>=)  :: m a -> (a -> m b) -> m b
```

The other interesting property of monads is the existence of a class of monads called monad transformers. These are arity-2 type constructors which take as their first parameter a monad type constructor. The partial application of such a type constructor to any monad type constructor then results in a monad. A construct is then provided to lift values of the underlying monad to values in the transforming monad, which may then be sequenced as normal. This leads to the powerful concept of monad transformer stacks, where monads

with many different capabilities are combined without any more plumbing than explicit `lift` operations.

This property will be useful because we wish our evaluation interface to be part of a monad stack, the base of which is Haskell's IO monad. By formulating the evaluation interface as a monad transformer, we need only define the operations relevant to the evaluation of signal functions, and we can depend on the constructs of the IO monad to interact with whatever inputs and outputs are necessary. In some cases, we may not wish to use the IO monad at all (e.g for testing or simulation). In this case, we can parameterize over another monad, such as the Identity monad (which has no special operations and whose context is just the value), or the State monad (which maintains an implicit state accessible by `get` and `put` operations).

The monad tranformer typeclass is defined as:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

Haskell also supports *do*-notation for monads. Do notation simply makes monadic application implicit between lines and reverses the syntatic order of lambda bindings, providing a way to use monads in a manner similar to imperative programming. (See Figure 4.12 for an example.)

4.2.1 Constructs and Interface

The evaluation interface is exported as follows:

```
-- Do notation:
do putStr "Hello: "
   string <- getStr
   putStrLn $ "Nice to meet you " ++ string
   time <- getCurrentTime
   print time
   putStrLn "Look at the time! Bye."

-- Translates to:
putStr "Hello: " >> getStr >>=
(\string -> (putStrLn $ "Nice to meet you " ++ string) >>
 getCurrentTime >>=
 (\time -> print time >> putStrLn "Look at the time! Bye."))
```

Figure 4.12: An example of monadic *do*-notation and its equivalent monadic expression.

```
data SFEvalState svIn m
data SFEvalT svIn m
data HandlerSet sv m
data EventOccurrence sv
data SignalDelta sv

sd      :: a -> SignalDelta (SVSignal a)
sdLeft  :: SignalDelta svl -> SignalDelta (SVAppend svl svr)
sdRight :: SignalDelta svr -> SignalDelta (SVAppend svl svr)
sdBoth  :: SignalDelta svl -> SignalDelta svr
        -> SignalDelta (SVAppend svl svr)

eo      :: a -> EventOccurrence (SVEvent a)
eoLeft  :: EventOccurrence svl -> EventOccurrence (SVAppend svl svr)
eoRight :: EventOccurrence svr -> EventOccurrence (SVAppend svl svr)

sdHS    :: (a -> m ()) -> HandlerSet m (SVSignal a)
eoHS    :: (a -> m ()) -> HandlerSet m (SVEvent a)
hsLeft  :: HandlerSet m svl -> HandlerSet m (SVAppend svl svr)
hsRight :: HandlerSet m svr -> HandlerSet m (SVAppend svl svr)
hsBoth  :: HandlerSet m svl -> HandlerSet m svr
        -> HandlerSet m (SVAppend svl svr)
```

```
initSFEval :: HandlerSet svOut m
           -> SF NonInitialized svIn svOut
           -> SFEvalState svIn m

runSFEvalT :: SFEvalT svIn m a -> SFEvalState svIn m -> m a

push   :: EventOccurrence svIn -> SFEvalT svIn m ()
update :: SignalDelta svIn -> SFEvalT svIn m ()
sample :: Double -> SFEvalT svIn m ()
```

The `SFEvalState` type constructor parameterizes over input types for signal functions, and underlying monads for a monad transformer, but is not itself a monad transformer. It describes the state of signal function evaluation. It consists of a record with four members: the current signal function, the set of handlers for outputs, the current input signal delta, and the last sample time.

The `SFEvalT` monad transformer is a newtype wrapper around the `StateT` monad available in the Haskell `transformers` package. The StateT monad wraps any other monad, providing a state which may be read using the `get` action and replaced with the `put` action. An action in the underlying monad is obtained by supplying a `StateT` value and an initial state value to the `runStateT` function.

The `SFEvalT` monad transformer does not make the `get` and `put` actions available, but uses them in its implementation of the `push`, `update`, and `sample` actions.

The `push` action pushes an event onto the input of the signal function, resulting in immediate evaluation of the relevant components signal function and possible actuation of handlers specified for output events. It is implemented by fetching the `SFEvalState`, applying the event continuation of the signal function contained by the `SFEvalState` to the pushed event, thus obtaining a new signal function and a list of events, applying the handlers contained in the `SFEvalState` to the output events, replacing the signal function in the `SFEvalState`, and replacing the `SFEvalState`.

Chapter 5

Evaluation and Comparisons

We have described the implementation of a push-based signal-function FRP system. A push-based system should theoretically have advantages in event latency and overall performance over a pull-based system. Further, we would like to compare the evaluation interface in the current widely-used system with the one proposed here, again in order to verify that the theoretical benefits do appear in practice.

One way to evaluate this is to write an application in two systems, and benchmark these systems for comparison. In order to compare TimeFlies with Yampa, the most recently released pull-based signal-function FRP system, we decided to implement a suitable application in both.

The application of choice is a learning switch controller for OpenFlow switches. this application must handle input events triggered by switches signalling that they have received a packet for which they have no handling rules, and produce output events which install handling rules for dealing with these packets.

The reactive algorithm is simple. A table from machine addresses to switch ports is maintained. This table is updated in two ways. When a

packet is sent to the controller, meta-data for the packet indicates which port it arrived on. The source address of the packet can then be mapped to that port in the table. Once this occurs, rules are installed for each packet with a destination address matching the source, along with reciprocal rules.

Each mapping in the table has a timestamp. After a certain amount of time, the table entry is deleted. The table entry is also deleted if a packet is received with the same source address and a different port. When an entry is deleted, the switch is signaled to remove all corresponding rules.

In both Yampa and TimeFlies, this algorithm is implemented by using an accumulator, which applies functions carried by event occurrences to a state value. It stores the new state and outputs new event occurrences based on the output of the function.

The state stored is the table (implemented using the `Data.Map` module in the Haskell `containers` library) and the output event occurrences carry messages to be sent to the switch. The events are sourced internally to the learning switch signal function (using the `after` combinator in TimeFlies and the `repeatedly` function in Yampa) to delete expired rules, and externally, by transforming messages from the switch to the controller into accumulator functions.

The controllers were benchmarked using the `cbench` program [2]. The amount to time to pause between time and signal updates was passed as a parameter to both programs. The `cbench` benchmark operates by sending a

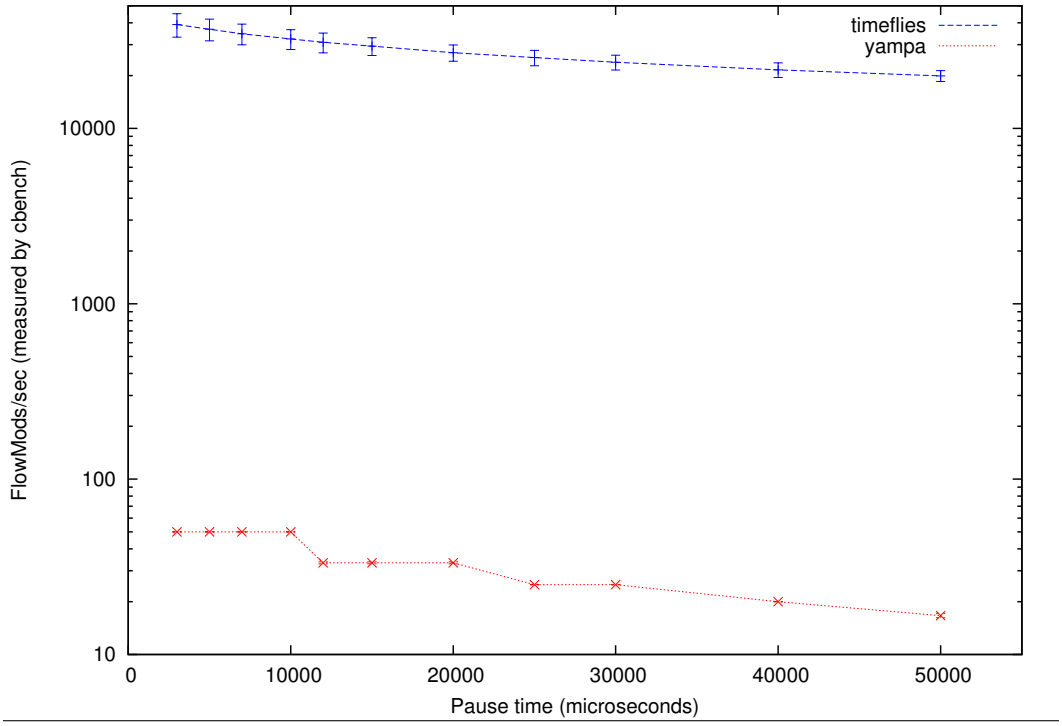

Figure 5.1: Comparison of timeflies vs. Yampa implementing an OpenFlow learning switch controller

simulated packet in event, waiting for a rule response, and then repeating. The output of the benchmark is the number of times this occurs in a second. This is thus a very good test of event latency in a reactive system.

Figure 5.1 shows the performance comparison. Toward the left of the graph is lower pause times (more frequent sampling). The y-axis is a higher rate of responses (higher numbers are preferred).

There is some unexpected correlation between the sampling rate and the response rate for TimeFlies, which is likely due to the more frequent size reductions in the table. Nevertheless, TimeFlies outperforms Yampa in event

response by several orders of magnitude, even at a very high sampling rate. This is due to two factors. First, TimeFlies can react to individual events without evaluating the whole signal function. Second, the interface of time-flies permits us to connect its evaluation directly to GHC's event manager, while the design of Yampa requires us to use threads and synchronized communication[1].

[1]Yampa does include a step-wise evaluation function, but it still couples event evaluation firmly to time steps, and is not well documented.

Chapter 6

Conclusions and Further Work

I have presented TimeFlies, a push-based signal-function FRP system. I have demonstrated that TimeFlies does realize the theoretical benefits of a push-based signal-function system.

The TimeFlies system is a full-fledged and documented FRP library which may be extended with utility functions and further optimized. Its performance in responding to events is demonstrated to be superior to that of the predominant pull-based signal-function FRP library.

Further, the model of events used by TimeFlies subverts problematic semantic questions about the evaluation of events in a system. By using the N-Ary FRP type model and separating the evaluation of events from the time steps used for signals, TimeFlies fully supports non-deterministic merging of events, and provides a semantic guarantee that events are not "lost" during evaluation.

The TimeFlies system would benefit from attentive microbenchmarking and performance tuning, as well as optimizations to avoid evaluating irrelevant parts of the network during the the evaluation of time steps. A formal semantic justification for the formulation of event evaluation (which would require a

full formal semantics of FRP) would enable a far more robust correctness argument, as well as providing a basis for semantic extensions to signal-function FRP.

FRP is not yet mature, and has not been the subject of focused application development. Thus, there is a dearth of design patterns for FRP applications. Such design patterns would yield necessary feedback as to which generalizations and restrictions of FRP would be appropriate and useful, and clarify the necessity of various utility combinators to be included in the standard libraries of FRP systems.

In order to improve the performance of FRP yet further, it may be productive to attempt to introduce parallel evaluation into FRP, taking advantage of the functional purity in the implementation of the signal function combinators. This may involve, for instance, evaluating several time-steps at once in a data-parallel manner,task parallelism between different branches of a signal function, or speculative evaluation of switch combinators.

Many classes of reactive application would benefit from a "dynamic collections" combinator similar to `pSwitch` in Yampa. Such a combinator allows a collection of signal functions to be evaluated as one signal function, with addition or removal of signal functions instead of the total replacement given by the `switch` combinator. This is useful, for instance, when simulating objects for games or computer models, as the behavior of each object can be modeled as a signal function, and these signal functions can be added to and removed from the collection.

The TimeFlies system provides a principled and performant system for future experimentation on FRP, as well as implementation of FRP applications.

Appendix

Appendix 1

Glossary of Type System Terms

One of the primary attractions of the Haskell language, and the reason for its use throughout this work, is its advanced yet practical and usable type system. This type system enables the use of compositional software design that would be rendered infeasible without a type system to both inform and verify composition and implementation. This glossary describes features of the type system which may not be familiar to a general audience.

ADT See *Algebraic Datatype*

Algebraic Datatype An Algebraic Datatype or ADT is a type whose terms are *data constructors*. An ADT is defined by naming a *type constructor* and its parameters (zero or more) (as *type variables*), together with one or more data constructors and the types of their members. Each data constructor takes a fixed number (zero or more) data members, whose types are given following the constructor name. These types are defined in terms of the type variables named as parameters of the type constructor and any type constructors (including the type constructor associated with the ADT) in scope in the module.

Data Constructor A Data Constructor is a component of an Algebraic Datatype which, when applied to values of the appropriate type, creates a value typed with the ADT. Data constructors are the primary element which may be pattern matched in languages such as Haskell.

Existential Type An existential type is a means of "hiding" a *type variable* in an *Algebraic Datatype* or *Generalized Algebraic Datatype*. The Glasgow Haskell Compiler provides existential types in two ways. The first is the `Rank2Types` and `RankNTypes` extensions. With either of these extensions enabled, type variables may be brought into scope for specific *data constructor* in an Algebraic Datatype using the `forall` keyword. Since this type variable is not contained in the parameters of the type constructor, it cannot be constrained by type inference. Thus, no external function can assume anything about its type, except for equality between occurrences of the variable, and *class constraints* given in the definition of the data constructor. The other way is through Generalized Algebraic Datatypes. Any variable occurring in the type of a data member of a data constructor of a Generalized Algebraic Datatype, but not occurring in the type of that constructor, is automatically existentially quantified.

Generalized Algebraic Datatype Similar to an *Algebraic Datatype*, but *type variables* in the *type constructor* declaration serve merely to denote the number of type parameters (and thus may be replaced by a *kind*

signature) and types are given for each *data constructor*. These types must have the type constructor as their top-level term, but may fill in the parameters of the type constructor with arbitrary types. Variables which appear in the data member types but on in the data constructor type are *existentially quantified*, and types appearing in the data constructor type but not the data member types may be instantiated arbitrarily.

Kind Signature A means of specifying the number and kind of types which may instantiate type variables. Type variables in Haskell are not restricted to types, but may be instantiated by type constructors as well. The kind of a variable restricts what it may be instantiated with. A kind signature gives kinds to a type constructor, and thus to its parameters. Specifying the kind of a type constructor perfectly constrains the number of parameters.

Type Constructor A type constructor is a type level term which, when applied to the proper number of types, produces a type. Type constructors, together with *type variables*, form the basis of polymorphism in Haskell and similar languages.

Type Variable A type variable is a type-level term which may be instantiated (by the typechecker via inference, or by the user via annotation) with any type. Together with *type constructors*, type variables form the basis of polymorphism in Haskell and similar languages.

Bibliography

[1] Heinrich Apfelmus. reactive-banana library. `http://hackage.haskell.org/package/reactive-banana`.

[2] cbench benchmark. `git://gitosis.stanford.edu/oflops.git`.

[3] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

[4] Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, UK, 2001. Springer-Verlag.

[5] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[6] Evan Czaplicki. Elm: Concurrent FRP for functional GUIs. `http://www.testblogpleaseignore.com/wp-content/uploads/2012/03/thesis.pdf`, 2012.

[7] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[9] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(13):67 – 111, 2000.

[10] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995.

[11] Adam Megacz. Multi-level languages are generalized arrows. *CoRR*, abs/1007.2885, 2010.

[12] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.

[13] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[14] Simon Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series: Computer & Systems Sciences*, pages 47–96. IOS Press, Amsterdam, The Netherlands, 2001.

[15] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[16] Neil Schulthorpe. *Towards Safe and Efficient Functional Reactive Programming*. PhD thesis, University of Nottingham, UK, 2011.

[17] Ertugrul Söylemez. netwire library. `http://hackage.haskell.org/package/netwire-3.1.0`.

Vita

Edward Amsden was born in Dayton, Ohio in the year 1990, to Andrew and Vivian Amsden. He is pursuing concurrent B.S. and M.S. degrees at the Rochester Institute of Technology. Once his M.S. is completed, he plans to begin his Ph. D. at Indiana University. His research interests include functional programming languages, concurrency and parallelism, computer graphics, and computer audio.

Permanent address: 13567 McCartyville Road
Anna, Ohio 45302

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.