CHALMERS

Streaming Computations in Feldspar

Master of Science Thesis

MARKUS ARONSSON

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, June 2014

Streaming Computations in Feldspar

Markus Aronsson

Examiner: M. Sheeran

Supervisor: E. Axelsson

Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Abstract

We present a library for expressing digital signal processing algorithms using a deeply embedded language in Haskell. It is an extension Feldspar's stream library, which provides a data type for infinite sequences of values. Until now, streams have been confined to a set of explicit functions for expressing recurrence equations, delays and sampling. The extension we present removes the need for such functions, reducing the gap between streaming algorithms and their mathematical description; efficiency is maintained by optimising the network structures. We demonstrate the usefulness of our library by giving example implementations of common filters and control structures found in digital signal processing.

Our library is built on top of Feldspar, a domain specific language for programming digital signal processing algorithms, and, like our extensions, is implemented as a deeply embedded language in Haskell. The Feldspar language itself offers high-level and functional style for describing digital signal processing algorithms, where the final aim of a Feldspar program is to generate high-performance low level code. Feldspar is based around a low-level functional core language, which is semantically similar to machine-oriented languages, such as C, and operates at around the same level of abstraction. A number of libraries built on top of the core language are offered, enabling programming in a higher-order manner, one of which is Feldspar's stream library.

Feldspar stream library contains a modified version of co-iterative streams, a central concept for reasoning about and optimising stream systems. In the co-iterative approach, each stream consists of a transition function and an initial state, where the transition function takes a state and produces a new state and an output value. The interesting property of co-iterative streams is that they allow one to handle infinite streams in a strict and efficient manner, instead of having to deal with them lazily. As the co-iterative approach provides a good foundation for reasoning about streams, the contributions of this project are based around such streams with added support various temporal operations.

Acknowledgements

I would like to express my warmest gratitude to my supervisor Assistant Professor Emil Axelsson for the many useful comments, observations and engagement throughout the learning experience that has been this master's thesis. Furthermore, I would like to thank Professor Mary Sheeran for introducing me to the topic and giving me the opportunity to conduct my study at her research group, as well as for all the support on the way. Also, I like to thank all the members of the research group and visiting professors, who willingly shared some of their own precious time to listen to and discuss my ideas.

I would also like to thank my family for the support they provided me through this thesis and in particular, I must acknowledge my sweetheart and best friend, Emma, without whose love and encouragement, this would have been so much harder.

Markus Aronsson, Gothenburg, Sweden, June 2014

Contents

1

Introduction

In recent years, the amount of traffic passing through the global communications infrastructure has been increasing at a rapid pace. Worldwide, total Internet traffic is estimated to grow at an average rate of 32% annually, reaching approximately eighty million terabytes per month by the end of next year [1, 2]. Mobile communications in particular have been growing at a phenomenal rate, see Figure 1.1, which can be largely attributed to the rising popularity of social networking services and mobile terminals – such as smart-phones and tablets. To meet the growing demands of mobile users, telecom carriers are striving to enhance service quality.

For telecommunications infrastructure, the consequences of such a rapid growth rate have been a dramatic increase in the demand of network capacity and computational power [1]. At the same time, telecom carriers are faced with an increasing need to deliver new services faster, while simultaneously adapting to the recent diversification in available architecture, coupled with an emergence of new and powerful computational platforms, such as multi-core machines. These factors, while positively influencing the available computational power, have also significantly increased the complexity of developing new solutions for telecommunication systems.

Digital signal processing is a central concept for managing content delivery between radio base stations and mobile terminals. Today, digital signal processing algorithms are typically implemented in low level C [3], and often using hardware specific optimisations, such as pragmas or build flags. Programming in a low-level language forces designers to emphasise the use of hardware intrinsic, rather than focusing on describing an algorithm's core concepts. This makes such languages a poor fit for describing the mathematical nature of most signal processing algorithms; C is used mainly because performance is critical in the different applications of signal processing. Furthermore, code portability is severely limited as a direct result of using hardware specific intrinsics, even when porting between systems of the same manufacturer.

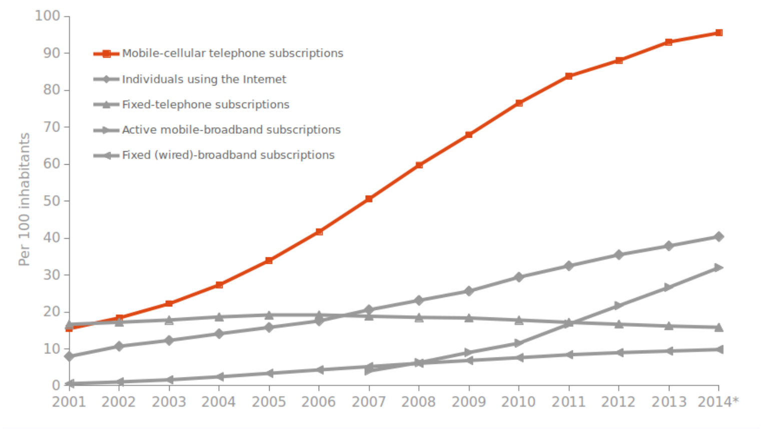

Figure 1.1: Global ITC Development, 2001-2014

The Feldspar project aims to counter the complexity inherent in using a low level language for digital signal processing, by raising the level of abstraction at which its algorithms are programmed – without sacrificing vital performance. Feldspar [4, 5] is a domain specific language embedded in Haskell [6], it is a purely functional language and provides a compositional approach for expressing algorithms in a higher order manner. These descriptions can then be translated by Feldspar's associated compiler into sequential and hardware specific ISO99 C code.

1.1 Problem Description

Feldspar is designed for use in the digital signal processing domain, where stream computations are a common feature. Support for such computations is however quite limited. The stream library [7] provided by Feldspar currently offers a slightly modified version of generalised co-iteration [8], where all streams are restricted to operate according to one global clock frequency.

Even though the use of a global clock is potentially beneficial to performance, it limits the kind of system one can model. For instance, consider the likely scenario of modeling a system running on a limited architecture. In such systems, where the architecture for example only allows a certain number of parallel operations, it is necessary to replace parallelism with iteration. This affects the time instant when results become available, and consequently the systems clock. As sub-components, now running at possibly different frequencies, must still communicate with each other, a model supporting multiple clocks is required – where their synchronicity could be expressed. Furthermore, introducing clocks permits stream processors to be statically scheduled, which could potentially alleviate the need for control structures inherent to dynamic scheduling.

Another drawback of the current implementation is the lack of support for general recurrence equations; a unique operator is instead supplied for each conceivable use case. Restricting the feedback in such a way enables the use of memory efficient buffers. A feedback network with a non-standard recurrence will however require arbitrary cuts, refitting the graph for use with Feldspar's recurrence equations and hence deviating from its original description. Consider the mathematical definition for a finite impulse response filter of rank N, where each value of the output stream is a weighted sum of the most recent input values:

$$y_n = \sum_{i=0}^{N} b_i * x_{n-i}$$

This filter can be deconstructed into its three main components: a number of unit delays, multiplications with some coefficients and a summation of the amplified signals. We can represent the decomposed filter graphically as in Figure 1.2.

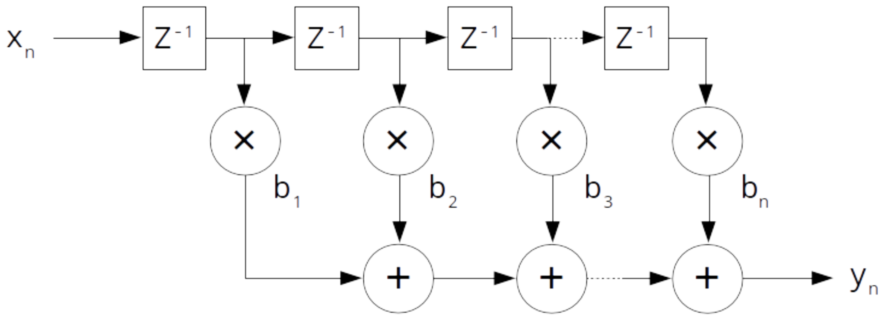

Figure 1.2: A direct form discrete-time FIR filter of order N

where each unit delay is expressed as a z^{-1} operation in Z-transform notation. As domain experts in DSP tend to be comfortable with the idea of composing sub-components in this way, using boxes and arrows, one would ideally capture this in the filter's corresponding Feldspar program. Feldspar does however not allow arbitrary streams to be delayed, we are instead forced to express the filter as a vector computation using its set of recurrence equations.

```
import Feldspar
import Feldspar.Stream
import qualified Feldspar.Vector as V

recurrenceI :: (Type a, Type b) ⇒
                  Vector1 a
               → Stream (Data a)
               → (Vector1 a → Data b)
               → Stream (Data b)

fir :: Vector1 Float → Stream (Data Float) → Stream (Data Float)
fir a inp = recurrenceI (V.replicate (V.length a) 0) inp
                        (V.scalarProd a)
```

1.2 Contributions

This project aims to investigate what kind of streaming model is needed by the intended applications of Feldspar and implement support for it, both in terms of language constructs and compilation. In order to achieve this goal, the following contributions have been made:

- We investigate current work in the area, developing a general view of related languages and their respective programming paradigms. By surveying a number of languages from each paradigm we develop a broad sense of their respective benefits, indicating which best fits the intended applications of Feldspar. We also perform an in-depth review of a single language from each paradigm and one for Feldspar's stream library.

- We develop a new streaming model for use in Feldspar and implement support for it. Our intent is to supply generic programming blocks, with which the various composite streaming computations can be expressed, leaving as much of the optimisation as possible to be handled automatically during compilation. Our model includes support for multi-clocked systems, allowing us to reason about program correctness when applied to a real-world setting.

- We make use of type-safe observable sharing in our extension, an often critical component for allowing deeply embedded languages to be effectively optimised when compiled. By doing so, we allow for the sharing present in our abstract syntax trees to be directly observed through a graph representation of the trees; notating any sharing of computed results and recursively defined computations. By inspecting the sharing present in program we successfully avoid the need for explicit looping constructors.

- We support compilation of our new streaming model into monadic expressions, which can then be turned into sequential C code by Feldspar's compiler. We also utilise several pre-processing stages to reduce and optimise the reified program's graph before compilation; pre-processing the program's graph allows us to, for example, support arbitrary feedback networks while still retaining the efficient memory management offered by Feldspar's recurrence equations. While the data types support multiple clocks, we have yet to implement support for them during compilation.

1.3 Methodology and Limitations

An initial feasibility study was conducted at the beginning of the project, where we investigated what kind of streaming model would be desirable for use in Feldspar; where desirable implies, for example, an emphasis on performance or an intuitive syntax. In conjunction with the initial study, a review of several related languages, and their associated programming paradigms, was performed. By considering languages from multiple programming paradigms we hoped to find a combination of techniques which would allow for efficient compilation, while still retaining the higher order properties of Feldspar. We also reviewed Feldspar and its stream library, comparing its approach to the other models we reviewed.

The literature review consisted of a general study into each paradigm, analysing how each approach handles the concept of signals, events, control structures, and other common features in the digital signal processing domain. A single language from each paradigm was selected for an in-study, during which we investigated how they implemented the core concepts from their respective paradigms. This approach enabled us to study the general theories underlying each paradigm in more detail, while also exploring their respective benefits when applied to a functional programming setting. Results from the literary review and initial study later served as inspiration when we developed our own extensions to Feldspar's stream processing library.

After the initial studies, a number of examples were implemented for each of the studied languages. Applying each language to a common set of example circuits helped us discern any advantages a specific approach may have over the others, as well as further investigate the limitations of Feldspar's streams in comparison to the others. These examples were constructed in such a way as to model common scenarios in signal processing: filters, control oriented systems, and modeling systems running on a limited architecture. We then developed our own streaming model, suited for Feldspar and based on the results from these comparisons and studies.

As digital signal processing is a well-explored area, a number of standard solutions have been developed for use in a range of different programming paradigms. The initial literature reviews were therefore limited to encompass only the more established paradigms; focusing mainly on that of synchronous data flow and functional reactive programming. While we considered other paradigms, their reviews were limited to a study of interesting language features. We further limited our studies to languages operating on signals in the time domain and reactive systems, as opposed to transformational systems, where the entire input is present at the start of execution.

2

Background

Digital signal processing (DSP) is the mathematical manipulation of signals, that is, DSP is all about taking a signal, applying some change to it, and then getting a new signal out [9]. That change might be filtration or amplification, and is usually applied to some sampled continuous signal originating from a real-world input – like voice or video recordings [10].

Some time ago, anyone using DSP had to be quite the mathematician to be able to implement and use the related algorithms. Today, several programming languages have been developed specifically for use in the DSP area, which allows one to view signal processors in a more abstract way. Seen in this abstract way, as a form of black box, the DSP system might be composed of other sub-components, or it could be a low-pass filter, or it could be some other complex integrated system. As long as the box accomplishes its defined task, it doesn't matter too much how the box works internally.

Due to the broad applicability of DSP, a number of models have been developed for use in the area, each with its own advantages and limitations. Amongst these models, the synchronous data-flow [11] and functional reactive models [12] are perhaps the most established ones. These two paradigms will therefore be discussed in detail, where a well-established language is chosen from each, to serve as a representative for that paradigm. By thoroughly studying the languages in this initial study, we hope to identify advantages to each approach, and ideally find a combination of the two which will best fit the needs of Feldspar.

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is a paradigm for programming hybrid systems, that is systems consisting of both continuous and discrete components, in a high-level and declarative manner. The key ideas in functional reactive programming are its notions of behaviors and events, where behaviors are continuous and time-varying, reactive values, and events are time-ordered sequences of discrete-time event occurrences [12].

Behaviors and events are both first-class values in functional reactive programming, and are accompanied by a rich set of operators to support the composition of new behaviours and events from existing ones. A reactive program is, simply stated, a collection of mutually recursive behaviors and events, each built from static values or other behaviors and events or both.

The notion of functional reactive programming originated from Fran [13], a domain specific language (DSL) for composing richly interactive, multimedia animations. The language FRP [12, 14] then abstracted out the essence of reactive programming from Fran, by simply ignoring its application specifics, leading to the notions of behaviours and events. These core concepts of functional reactive programming were then applied in a range of different domains, where each language embraced the paradigm in a way suited to its own application. For example, Fruit [15], a graphical user interface library for Haskell, and Elm [16], a functional language that compiles to HTML, CSS, and JavaScript, are both designed using a reactive approach. In addition, variants of functional reactive programming have also been used for modeling real-time systems [17], which led to the development of Yampa [18, 19] - which will be our focus language when studying FRP related languages.

Yampa is a refinement of FRP, realised in the form of a domain-specific language embedded in Haskell (DSEL) and structured using arrows [20] – a generalisation of monads. The language is based around the concepts of signals: continuous, time-varying values. Signals can conceptually be thought of as having the type:

```
Signal a :: Time → a
```

In other words, a type of `Signal a` is a mapping from time – where time is a non-zero real number – to the polymorphic type `a`. Signals can hence be used to capture the behaviour of a system, but they can also be used to model discrete events when given an appropriate choice of type parameter. For instance, by introducing a new data type

```
data Event a = NoEvent | Event a
```

A signal, given the above data type as its type parameter, will act as a source of discrete-time events, in Yampa these are known as event sources: a signal that at any point in time either yields nothing or an event carrying a given type. The distinction between continuous values and discrete events is therefore rather small. Signals are, however, not allowed as first class entities; they only exist indirectly through the notion of signal transformers – which are called signal functions in Yampa.

A signal function can conceptually be thought of as a mapping from signal to signal:

```
SF a b = Signal a → Signal b
```

Signal functions are required to be causal: the output can only depend on past or current input, not future input. The actual type for signal functions is abstract in Yampa, which prevents one from directly constructing signal functions – which could introduce time- and space-leaks [21, p. 4]. Programming in Yampa is instead done by defining functions compositionally, using a set of primitive signal functions, static streams and combinators.

Signal functions are made an instance of the `Arrow` class, and the combinators provided by the arrow framework are used to compose new signals functions. Arrows serve much the same purpose as monads, that is, they provide a common structure for libraries, but are more general. In particular, they allow for notions of computation that may take multiple inputs or be partially static. For instance, combinators for lifting functions, composition, converting an ordinary signal function into a function on pairs, and a loop combinator are all provided by the arrow library. These have the following types in Yampa, when applied to signal functions:

```
arr   :: (a → b) → SF a b
(>>>) :: SF a b → SF b c → SF a c
first :: SF a b → SF (a, c) (b, c)
loop  :: SF (a, c) (b, c) → SF a b
```

The behavior of a system may evolve over time, for example by transitioning into different states or an error handling mode. These behavioral changes of a system are known as mode switches in Yampa and are achieved through event sources and a family of switching combinators. Consider, for example, the event source

```
keyEvent :: SF KeyboardInput (Event Key)
```

which generates an event stream corresponding to the events of a key being pressed. These event sources can be used in conjunction with Yampa's library of switching combinators, to design control systems which change their behavior according to received events. The simplest such switching combinator is the `switch` combinator, with the type:

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
```

`switch` behaves as its first signal function until the event is produced; the event is then used to construct a new behaviour which the system then switches into. Using this switch operator, one could, for example, define a signal function which constantly outputs the last key pressed:

```
last :: Key → SF KeyboardInput Key
last k = (constant k &&& keyEvent) ‘switch‘ λe → last e
```

where `(&&&)` is another convenience method for the arrow framework, which sends its input to both argument arrows and combines their output. It has the following type

```
(&&&) :: SF a b → SF a b' → SF a (b, b')
```

Yampa's library also offers several other methods for switching behaviours, each one based around the different semantics commonly used in DSP. For instance, there is delayed switching, which changes behaviour in the time instant after the event occurred, and recurring switching, where the behaviour changes each time an event occurs.

2.1.1 Implementing Control Structures in Yampa

As an example, consider the problem of modeling a ball bouncing off the floor, dropped from some initial height. The height and velocity of a free-falling ball can be described using the following formulas:

$$height \ = heigth_0 + \int velocity * \mathrm{d}t$$

$$velocity = velocity_0 + \int -9.81 * \mathrm{d}t$$

These behaviours can then be captured in Yampa by using signals for both formulas:

```
fallingBall :: Pos → Vel → SF () (Pos, Vel)
fallingBall y0 v0 = proc () → do
  v ← (v0 +) ^<< integral ⤙ -9.81
  y ← (y0 +) ^<< integral ⤙ v
  returnA ⤙ (y, v)
```

This declares a process that takes some initial height and velocity of the ball and produces a stream of pairs, representing the current height and velocity of the ball at each time instance. The signals are defined using Yampa's integral function and the `(^<<)` combinator for post-composition of the integral with a pure function.

One should note that while the integral functions operates over a time interval, there is no explicit mention of time in the function. This comes from the fact that Yampa handles the notion of time internally using signals, and as signals are hidden in the signal function type and not first class values we cannot access time directly.

To keep the ball from falling straight through the floor, and instead start bouncing off it, we invert the ball's current velocity whenever its height reaches zero. This describes a typical application of switching in Yampa: a certain behaviour is used continuously until some event occurs, upon which the behaviour switches into another. Expressing this in Yampa can be done by first wrapping the previous function. The wrapper function adds an edge detector to the output stream of our earlier function, triggering as soon as the height reaches zero. This is done by writing:

```
fallingBallEvent :: Pos → Vel → SF () ((Pos,Vel), Event (Pos,Vel))
fallingBallEvent y0 v0 = proc () → do
  x@(y,_) ← fallingBall v0 v0 ⤙ ()
  bounced ← edge              ⤙ y ≤ 0
  returnA ⤙ (x, bounced ‘tag‘ x)
```

The produced events are tagged by the current state of the ball, as it allows us to extract the ball's current velocity when an event is produced. With this function, a signal function for modeling a bouncing ball can then be expressed as:

```
bouncingBall :: Pos → SF () (Pos, Vel)
bouncingBall y0 = go y0 0.0
  where go y0 v0 =
    fallingBallEvent y0 v0 ‘switch‘  λ(y,v) → go y (-v)
```

2.2 Synchronous Dataflow Languages

Dataflow programming (DFP) is a paradigm which internally models applications as directed graphs [22, 23], similar to a dataflow diagram. Nodes in the graph are then executable blocks, representing the different components of an application: they receive input, applies some transformation, and forwards it to the other connected nodes. A dataflow application is then, simply stated, a composition of such blocks, with one or more source and sink blocks. These nodes are linked together by directed edges, representing the data dependencies between components.

While DFP is not too common as a programming paradigm, it does offer some advantages in certain scenarios. For instance, the availability of formal verification tools is an important aspect for critical real-time control software [24]. Domain experts in DSP tend to structure applications using boxes and arrows, and are hence comfortable with the dataflow style of composing applications from sub-components. However, the key advantage is that, in dataflow graphs, several sub-components can be executed simultaneously, that is, it offers implicit concurrency. This concurrency comes from the representation of nodes as independent processing blocks in the internal graph. Since nodes run without any side-effects, the execution model allows nodes to fire as soon as they receive input. These are also run without any risk for deadlocking, since a node's input is also its only data dependencies [22, p. 3].

A later extension to DFP is the introduction of synchronous dataflow (SDF) [11]. SDF is a subset of pure dataflow, in which the number of tokens produced or consumed by nodes during each step of evaluation is known at compile-time. The advantage of this approach is that it can be statically scheduled [25]. This means that it possible to convert the data flow graph into a sequential program, which does not require dynamic scheduling. These sequential programs can, for instance, be finite state machines, which in turn can be translated into efficient code. Due to this advantage, SDF has been of particular use in the DSP domain where time is an important element of computations [26, 27].

The Lucid Synchrone language [28, 29] is a member of the family of synchronous languages. This family includes languages such as ESTEREL [30], a restricted data-flow language, suited for control-dominated model designs, LUSTRE [31], which is designed to works almost as a specification language, and SIGNAL [32], an event-driven language. These languages are designed to model reactive systems, in a way similar to Yampa, but using a dataflow programming paradigm with a heavier focus on being able to prove properties of a system. Lucid Synchrone was introduced as an extension of LUSTRE, and demonstrated that the language could be extended with new and powerful features. For instance, automatic clock and type inference was introduced, and a restricted version of higher-order formalism was added, while still retaining basic properties of LUSTRE.

In Lucid Synchrone, every type or scalar value imported from the host language, OCaml, is implicitly lifted to streams. These streams can be represented as chronograms, showing the sequence of values produced at each time instance. For example,

x	x_0	x_1	x_2	x_3	…
y	y_0	y_1	y_2	y_3	…
$x+y$	x_0+y_0	x_1+y_1	x_2+y_2	x_3+y_3	…

shows the values produced by the expression `x + y`, where `x` and `y` are two streams.

Stream functions, such as `(+)`, are separated into combinatorial and sequential functions. A combinatorial functions may only depend on the current input to produce its result, while sequential functions are causal and may depend on earlier input. One could, for example, define the combinatorial half- and full-adder circuits by

```
let half_add (x,y) = (s,co) where
  s = xor (x,y) and co = x & y

let full_add (x,y,z) = (s,co) where
  rec (s1,c1) = half_add(x,y)
  and (s, c2) = half_add(z,s1)
  and co      = c2 or c1
```

We use the let keyword to introduce a new functions, and in this example, two functions named `half_add` and `full_add` are introduced. `half_add` takes a pair as input and produces another pair, defined by the xor and conjunction of the input pair – where the `xor` and `(&)` operators are lifted versions of the same operators imported from the host language. Two half-adders and an or gate are then used to define the full-adder, where the rec keyword is used to indicate that the output of one half-adder is used as input for the other.

In addition to those operators lifted from the host language, a set of stream specific ones are given as well. Amongst these are the delay operator `pre`, which delays its input and is unspecified at its first time instance, and the initialization operator `(→)`, which combines two streams by taking the first stream's value at the first time instant followed by the second stream's values.

These two operators are commonly used together, often to simply delay a stream by one instant and initialise it with some value. A third delay operator, called `fby`, is therefore introduced, and is defined as: `x fby y = x → pre y`. The behaviours of these operators can also be understood by looking at their chronograms:

x	x_0	x_1	x_2	x_3	...
pre x	*nil*	x_0	x_1	x_2	...
$1 \to x$	1	x_1	x_2	x_3	...
1 fby x	1	x_0	x_1	x_2	...

Delay operators can be used to create recursively defined streams. One could, for example, implement a sequential function such as a counter by:

```
let node counter start = x
  where rec x = start → pre x + 1
```

where `node` is a syntactic indication that the expression is sequential, that is, the expression involves previous values and hence state.

Our definition of the counting function is length preserving, that is, it consumes as many tokens as it produces. However, Lucid Synchrone also support functions which produce and consume a variable number of tokens at each time instant, which allows one to sample and merge streams, even if they are operating different frequencies. This is achieved through the notion of clocks. One such function for sampling a stream is the `when` operator, which samples a stream according to some boolean stream. To sample our counter we write:

```
let node sampled_counter start c = (counter start) when c
```

which will produce the following chronogram

start	0	0	0	0	...
c	F	T	F	T	...
counter	0	1	2	3	...
scounter		1		3	...

The clock type of our sampled counter will hence depend on the boolean stream c. One could also construct an over-sampling function: a function which produces more tokens than it consumes; the stuttering function is an example of such a function. Such sampling and oversampling appear naturally in multi-clock systems when one, for example, considers the communication between subsystems operating on different clocks.

Synchronous dataflow languages, such as Lucid Synchrone and LUSTRE, are first-order languages, that is, a synchronous stream function corresponds directly to a finite state machine and there is a clear distinction between values and such functions. Higher-order in Lucid Synchrone comes instead from the ability to parameterize functions by other functions. However, the language only provides a restricted form of higher-order, since the function taken as a parameter, or returned as a result, is determined statically and cannot be dynamically altered while running. This allows the compiler to transform the system, by for example inlining, to a first-order system.

2.2.1 Modeling Real World limitations in Lucid Synchrone

Consider the case of implementing a power function in hardware. For example, if one wanted to compute the fifth power of a value, and the system had at least four multipliers, then a function could be defined as:

```
let node power x = y where y = x * x * x * x * x

val power : int ⇒ int   // function type
val power :: 'a → 'a    // clock type
```

But what if there only was one multiplier available? Then the fifth power cannot possibly be computed in a single clock cycle. Instead, a function which operates at a slower pace must be defined, where the result is computed incrementally. This is a common example, and a solution to the problem is presented in the manual for Lucid Synchrone [28, p. 24] as:

```
// defines a clock which is four times slower than the global clock
let clock four = sample 4

let node power x = y where
  rec i = merge four x ((1 fby i) whenot four)
  and o = 1 fby (i * merge four x (o whenot four))
  and y = o when four

val power : int ⇒ int
val power :: 'a on four → 'a on four
```

The output stream, `y`, is composed of two smaller streams, `i` and `o`, which holds the current sample and incrementally computed result, respectively. In both streams the merge combinators are used to fetch the current working value, and the result is then obtained by sampling the value stream whenever it has reached the correct power rank. Their behaviour can also be understood by inspecting their chronograms:

four	T	F	F	F	T	F	F	F	T
x	x_0				x_1				x_2
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2
0	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5
y	1				x_0^5				x_1^5

2.3 Feldspar

Feldspar[1] [4, 5] is a domain-specific language for numerical array processing, designed for use in the performance sensitive domain of digital signal processing, where it enables platform independent descriptions of DSP algorithms. It is implemented as a deeply embedded language in Haskell, and is strongly typed with pure semantics and has support for both scalar and vector computations.

Feldspar is based around a low-level functional core language, which is semantically similar to machine-oriented languages, such as C, and operates at around the same level of abstraction. A number of libraries built on top of the core language are offered, enabling programming in a higher-order manner. As is common for deeply embedded DSLs, programs written using these libraries generate an intermediate abstract syntax tree (AST) representing a core language program. Feldspar is then able to interpret these ASTs, translating them into some other language suitable for the target systems. While Feldspar offers full control over the core language, typical high-level programs are usually expressed using generators which enable optimizations to be performed on the fly.

Programming primitive functions in Feldspar works much in the same way as in their corresponding Haskell version. However, programs in the core language have the constructor `Data` added to all types. For example, the standard Haskell functions

```
(&&) ::          Bool → Bool → Bool
(==) :: Eq  a ⇒ a    → a    → Bool
( ≤ ) :: Ord a ⇒ a    → a    → Bool
(+)  :: Num a ⇒ a    → a    → a
```

have the following types in Feldspar:

```
(&&) ::              Data Bool → Data Bool → Data Bool
(==) :: Eq  a     ⇒ Data a    → Data a    → Data Bool
( ≤ ) :: Ord a     ⇒ Data a    → Data a    → Data Bool
(+)  :: Numeric a ⇒ Data a    → Data a    → Data a
```

where the type classes are Feldspar's representation of the corresponding classes in Haskell. The resemblance between Feldspar and Haskell remains even in larger examples; Feldspar is however more restricted than Haskell. By restricting the allowed constructs Feldspar enables code generation with predictable performance, as well as permitting control over important low-level details. One such major restriction is lack of recursion in the core language – such operations must instead be expressed using Feldspar's libraries.

[1]Version 0.7 of Feldspar was used during this report, available at [33]

The following example is a simple Haskell program which computes the bit-wise or of a mask with all numbers in the range of zero to some specified limit, summing the resulting numbers:

```
func :: Int32 → Int32 → Int32
func x y = sum $ map (y .|.) [0..x]
```

We can define a corresponding function in Feldspar using Haskell's function abstraction:

```
func :: Data Int32 → Data Int32 → Data Int32
func x y = sum $ map (y .|.) (0...x)
```

Notice that removing the word `Data` gives the same definition as in ordinary Haskell. The functions themselves differ only through the use of vectors instead of lists in the Feldspar version, where `(...)` is used to introduce a vector over a specified range. `func` does however no longer have a type of the form `Data a`, it is therefore no longer a core program. Conceptually, it is instead a macro which builds a program from the two programs `x` and `y`.

Feldspar's `printExpr` function allows us to inspect the core language generated from functions, by reifying the AST they generate. Applying `printExpr` to our earlier function, as defined above, produces a result similar to the following one; however, in order to increase readability its layout has been modified and explanatory comments have been added:

```
(λvar0 →      // x, range limit
  (λvar1 →    // y, the mask
    (forLoop
      (i2n (condition (var0 < 0) 0 (var0 + 1)))
      0
      (λvar2 →      // values in array
        (λvar3 →    // accumulator
          (var3 + (var1 .|. (i2n var2))))))))
```

where `i2n` is a conversion from integer to numerical values and `forLoop` represents a functional iteration construct, similar to the standard for-loops found in C, with the number of iterations, initial value and function body given as arguments.

Inspecting the core language above reveals that a number of vector operations have been fused into a single loop, where any intermediate data structures have been eliminated. Fusing operations is one of the most important features of vectors, guaranteeing that whenever two vector functions are composed, the intermediate vector is always removed. Vector fusion hence yields predictable compilation, given its strong guarantee of optimization, and has the additional effect of enabling a compositional style of writing programs, by using many small functions that are fused together.

Compilation into platform dependent ANSI C code is currently supported by Feldspar, where the `icompile` function is used to produce C code; running it on our earlier function `func` produces the following C program:

```
#include "feldspar_c99.h" // some includes are omitted

void test(int32_t v0, int32_t v1, int32_t * out)
{
  uint32_t len0;
  int32_t e1;
  int32_t v3;

  if((v0 < 0))
  {
    e1 = 0;
  }
  else
  {
    e1 = (v0 + 1);
  }
  len0 = ((uint32_t)(e1));
  (* out) = 0;
  for(uint32_t v2 = 0; v2 < len0; v2 += 1)
  {
    v3 = ((* out) + (v1 | ((int32_t)(v2))));
    (* out) = v3;
  }
}
```

The compiled code is similar in structure to that of its core language, with the core's `forLoop` translated into the flattened version above; its conditional statement and initial value have been moved outside of the loop while the function makes up the loop's body. While it is certainly possible to write programs using Feldspar's core language, it is a cumbersome process and the produced code can be rather opaque. One should instead take advantage of the many abstractions and optimizations provided by Feldspar's libraries. The core language does however provide finer control and gives opportunity for manual optimisations.

2.3.1 Co-iteration and Streams

In the framework of dataflow programming, co-iteration has been a central concept for reasoning about and optimising stream systems for quite some time [34]. It consists of associating a transition function and initial state to each stream, where the transition function takes a state and produces a new state and an output value. Feldspar contains a modified version of these streams, where the initial state and output are wrapped in monads, and has the following type:

```
Stream a :: Syntax state ⇒ (state → M a) → M state → Stream a
```

`M` is a monad for mutable state in Feldspar, and the Syntax constraint implies that the type supports conversion into an abstract syntax tree [35]. An interesting property of the above definition is that it allows us to handle infinite streams in a strict and efficient manner, instead of having to deal with them in a lazy way – as in the arrow paper [20, p. 87].

Streams can be used to represent the continuous behaviours of a system, similar to how signals are used in Yampa, but they can also be used to model discrete event occurrences when given an appropriate choice of type parameter. Signals in Yampa were used in conjunction with the Event data type to model event sources. Feldspar has a corresponding type called Option, which either carries some value or is empty, and has the following definition:

```
data Option a = Option { isSome :: Data Bool, fromSome :: a }
```

A stream, when given the above data type as its type parameter, will act in a similar manner as event sources from Yampa: at each step it either yields nothing or some event carrying a given type. Options can be constructed using the some and none functions, which are semantically similar to the Just and Nothing constructors for Haskell's Maybe type.

Given the above types for signals and events, coupled with the assumption of stream functions as simple mappings from stream to stream, we can implement some simple stream generators and transformers. For instance, we can define the following operators

```
repeat :: (Syntax a) ⇒ a → Stream a
repeat a = Stream return (return a)

fby :: (Syntax a) ⇒ a → Stream a → Stream a
fby a (Stream next init) = Stream newNext newInit
  where
    newInit         = init ‘with‘ (newRef a)
    newNext (st, r) = do v ← getRef vr
                         setRef vr =<< next st
                         return v

map :: (Syntax a, Syntax b) ⇒ (a → b) → Stream a → Stream b
map f (Stream next init) = Stream (next >=> return . f) init

zip :: (Syntax a, Syntax b) ⇒ Stream a → Stream b → Stream (a, b)
zip (Stream next1 init1) (Stream next2 init2) = Stream next init
  where
    init            = init1     ‘with‘ init2
    next (st1, st2) = next1 st1 ‘with‘ next2 st2

with :: Monad m ⇒ m a → m b → m (a, b)
with = liftM2 (,)
```

`repeat` is an example of a simple signal generator, as it creates a stream which outputs the same value indefinitely. The `fby` operator, inspired by Lucid Synchrone, is slightly more complicated as it requires the use of references to store values between iterations. Feldspar's mutable references are similar to Haskell's `IORef`, and are used to create a persistent state which can be altered during execution of the stream. Feldspar provides functions for managing such references through, for example, `newRef`, for creating new references, `getRef`, which fetches their contents, and `setRef`, for updating them. `map` and `zip` are the stream version of their corresponding functions in Haskell. Using these stream functions, we can define, for example, an edge detector:

```
edge :: Stream (Data Bool) → Stream (Data Bool)
edge str = map (uncurry (==)) $ zip str $ False ‘fby‘ str
```

3

Comparison of Approaches

In the previous chapter we introduced the two languages Yampa and Lucid Synchrone, which we selected as representatives for the functional reactive and synchronous dataflow paradigms, respectively. It is now our intent to further study these two languages through the implementation of a common set of examples.

The previous examples, power and bouncing balls, captured two important aspects of DSP, namely control oriented systems and clocks. Including them in our comparisons will allow us to study how each language tackles these common scenarios, and more importantly: how their approaches differ. The aim of these comparisons will be to discern any favorable attributes one approach may posses over the others, especially when considering their application in the DSP domain.

Following these comparisons, we will investigate the current state of Feldspar's stream library. In doing so we hope to determine the feasibility of introducing any promising features, discovered during the earlier comparisons, to Feldspar. We also intend to evaluate Feldspar's current streaming library. In order to do so, we try and implement a subset of Yampa's operators in Feldspar using its current stream library, allowing us to evaluate whether the concepts of signals and events can be supported using streams.

Another common concept in DSP is that of feedback networks, that is, streams defined using either earlier input or output, or a combination of the two. Feldspar's stream library currently exports a number of recurrence equations for modeling recursively defined streams so we implement some basic feedback networks in order to examine these operators.

3.1 Yampa and Lucid Synchrone

The power and bouncing balls function were implemented using Lucid Synchrone and Yampa, respectively, in the previous chapter. Therefore, in order to compare the two languages, we implement the same examples again, this time using the opposite language instead. We start out with the power example in Yampa, followed by the bouncing balls example in Lucid Synchrone.

3.1.1 Power in Yampa

When defining the power function in Yampa, one might incorrectly start out by writing something with a type similar to:

```
power :: Num a ⇒ SF a a
power = ?
```

which fails to capture the behaviour of the function, as it forces us to return a value during each time instant. Rather than a constant stream, the power function's discrete behaviour is closer to that of an event source in Yampa.

Redefining the power function, to instead make use of events, gives us the ability to return either nothing or some value at each step, that is, we are free to not return a value until one has been calculated. This discrete behaviour, coupled with the use of sampling and recursively defined signals, allows us to define the power function in a similar way as to how it was defined using Lucid Synchrone:

```
power :: Num a ⇒ SF a (Event a)
power = sample 5 >>> recur $ proc x → do
  i ← hold    1 −≺ x
  o ← loopPre 1 (arr $ λ(i,c) → (c, i * c)) −≺ i
  y ← sample  5 −≺ o
  returnA −≺ y
```

While a smaller definition could be achieved by using the recursive arrow notation instead of an explicit loop, its similarity to the corresponding version in Lucid Synchrone makes it easier to compare the two. As before, the output is constructed using three smaller streams; where the first stream, `i`, holds the current value of `x` and the second stream, `o`, incrementally computes the correct power. This second stream is recursively defined, as it keeps track over previous output, and is realised using the `loopPre` operator:

```
loopPre :: c → SF (a, c) (b, c) → SF a b
```

The operator expresses computations in which an output value is fed back as input, this internal state is initialised by its first argument. recur is then used to flush the state of the loop each time an output event is produced, restarting any internal computations in a way similar to Lucid Synchrone's sampling operator.

Even though a number of sampling operators were used in the above power example, the actual notion of time is still handled implicitly by Yampa, that is, the function's type does not reflect the orderly behaviour of the system. By inspection it becomes apparent that the function operates at a pace five times slower than that of its input, but the Event type hides this information. Losing such information might not be much of a concern when programming in Yampa; in Feldspar however, discarding such information is a bad idea, as it could be used to generate a static schedule for the slower streams [25].

3.1.2 Bouncing Balls in Lucid Synchrone

As Lucid Synchrone lacks the switch operators of Yampa, state machines are instead used to describe control dominated systems. A state machine, or automaton, is simply a collection of states and transitions, where a state is made of a set of equations and can be parametrised. To implement the bouncing balls example in Lucid Synchrone, we need to define such an automaton with two states; as these two states will represent the control structure imposed by the switch operator. The first, and initial state, will simply initialise the automaton's output and then immediately transition into the second state. This second state will then handle all the logic associated with bouncing a ball: this includes monitoring the moving ball and continuously outputting a stream of its position and velocity, until the height reaches zero, upon which it transitions into the same state with its velocity negated.

```
let static t = 1.0 // sampling frequency
let node integr x0 dx = let rec x = x0 → pre x +. t *. dx in x

let node fallingBall (y0,v0) = (y,v) where
  rec v = integr v0 (-9.18)
  and y = integr y0 v

let node bouncingBall y0 = o where
  rec automaton
    Init →
      do o = (y0, 0.0) then Bounce(o)
    Bounce((yI,vI)) →
      let (y,v) = fallingBall (yI,vI)
      do o = (y,v) until (y ≤ 0) then Bounce((y,-v))
  end
```

As Lucid Synchrone does not have any built in integral function, we defined our own using delays and the lifted versions of OCaml's standard floating-point operators. Then we defined the two functions, `fallingBall` and `bouncingBall`, and gave them same behaviour as their corresponding functions in the Yampa version.

The use of automata in the above example certainly yields compilation into efficient code, as it can be converted to a loop with a simple switch-statement inside; avoiding interference between states would however require a limitation on the effects of streams, which currently is not present in Yampa. It should also be possible for a clever compiler to unroll the main loop, taking advantage of the knowledge that the first state is only run once. However, the family of switch operators provided by Yampa offers far more modularity than automata, as a specialised automaton needs to be designed for each unique control structure. Even though automata provide some desirable attributes, such as performance, one would ideally combine them with a syntax closer to the switch operators of Yampa.

3.2 FRP and SDF in Feldspar

While the expressive power of both languages seems relatively even, the FRP paradigm and Yampa seems to be the preferable choice, as its semantics is closer to Feldspar than that of SDF and Lucid Synchrone. It is also possible to implement automata in Feldspar or Yampa, as one could implement a simple Mealy-styled automaton [36] using tuples and streams; the converse is however not true, as one cannot define a state machine in Lucid Synchrone with an infinite number of states. An infinite number of states might not however have many practical applications, unless one intends to use formal models for checking side-effects.

Ideally, one would include clock types in our extensions as well, since they give access to optimisations such as static scheduling. The method of associating each function with an additional clock type [37], as done in Lucid Synchrone, cannot however be easily ported to Feldspar: functions in Haskell are only associated with a single type and we cannot simply add another. This could be solved by merging the function- and clock-type, which would require the use of type level programming. However, as Feldspar already consists of multiple levels we would prefer to avoid introducing yet another one. Clocks could instead be associated with stream functions themselves, and used to identify whenever changes in frequency occur between two composed transformers.

Now, armed with the knowledge gained during our comparisons, we examine to which extent the concepts and ideas we developed can be supported using streams.

3.2.1 Streams in Feldspar

As we saw in the previous chapter, Feldspar's stream library could already support the notion of signals and events from FRP. It is now our intent to further explore to what extent the concepts from Yampa can be realised using streams, and in order to do so, we implement a subset of Yampa in Feldspar, including a modified version of the switch operator.

Given our earlier assumption of signal functions, it turns out that Feldspar's stream library is expressive enough for most of Yampa's operators to be realised using it. For example, the following functions from Yampa's libraries

```
identity   :: SF a a
constant   :: b → SF a b
sscan      :: (b → a → b) → b → SF a b
never      :: SF a (Event b)
now        :: b → SF a (Event b)
after      :: Time → b → SF a (Event b)
repeatedly :: Time → b → SF a (Event b)
```

can be given corresponding implementations in Feldspar with streams:

```
identity :: Stream a → Stream a
identity = id

constant :: (Syntax b) ⇒ Stream a → Stream b
constant b = λ _ → repeat b

sscan :: (Syntax b) ⇒ (b → a → b) → b → Stream a → Stream b
sscan f b = mapAccum (λ b a → let x = f b a in (x, x)) b

never :: (Syntax a) ⇒ Stream (Option a)
never = repeat none

now :: (Syntax b) ⇒ b → Stream a → Stream (Option b)
now b = λ _ → some b ‘fby‘ never

after :: (Syntax a) ⇒ Data Length → a → Stream (Option a)
after l a = afterEach $ indexed (1 :: Data WordN) (const (l,a))

repeatedly :: (Syntax a) ⇒ Data Length → a → Stream (Option a)
repeatedly l a = mapNth (const (some a)) (l + 1) l $ never
```

These functions successfully mimic the behaviour of their Yampa versions, and most of them could be realised using functions provided by Feldspar's stream library. Those signal functions that could not be supported – at least not using the existing stream functions – were those with an irregular behaviour. For instance, the `after` function generates a single event after a variable amount of time. In order to support the `after` function's irregular behaviour, and others like it, we defined a function that generates events according to some schedule. For simplicity, this schedule is represented as a vector of pairs, where each pair contains an event and the time until the event should be produced:

```
afterEach :: (Syntax a) ⇒ Vector (Data Length, a) → Stream (Option a)
afterEach xs = Stream next init
  where
    init =
      do vr ← newRef xs
         ir ← newRef (0 :: Data Index)
         tr ← newRef (0 :: Data Index)
         return (vr, ir, tr)

    next (vr, ir, tr) =
      do v ← getRef vr
         i ← getRef ir
         t ← getRef tr
         ifM (length v ≥ t)
           (do let (l, x) = v ! i
               ifM (i == l)
                 (do setRef ir 0
                     modifyRef tr (+1)
                     return $ some x)
                 (do modifyRef ir (+1)
                     return none))
           (return none)
```

While `afterEach` does make use of vector operations, we avoid going into how Feldspar handles vectors; conceptually, we can think of vectors as immutable lists, and its operations are semantically similar to those on lists as well. The function itself behaves as a counter, since it continuously counts down until the next events should occur. When an event finally is produced, the function moves on to the next item in the schedule and starts over.

As we do not yet have access to a switch operator, implementing the bouncing balls example for streams is still infeasible. However, we can implement some simple streams; for example, clocks could be represented as streams by:

```
minutes :: Stream Bool
minutes = repeatedly 60 true

hours :: Stream Bool
hours = sscan countdown 0 minutes
  where
    countdown n True  = (n + 1 ‘mod‘ 60, n == 60)
    countdown n False = (n, false)
```

3.2.2 Switching by Streams

Even though Yampa's signal functions could be realised straightforwardly as streams, it proves difficult to implement its family of switch operators as problems arise when considering their possible definitions in Feldspar and how existential types are used in the construction of streams.

The first problem encountered when implementing switches is the fact that a stream is partly static: a stream's state can be updated, but not the transition function. This means that the stream's behaviour cannot be dynamically modified while running. Instead, a new stream will be constructed whenever an event occurs, which will in turn require typecasting of the created stream's state – as their types will not necessarily be equal. Typecasting does however imply that we know beforehand which type we are casting to. Since the type of a stream's state is existential, as shown in chapter 2.3.1, and hence only known after construction, we are forced to construct an initial stream without access to any events. It is possible to use undefined values for this purpose, which are represented using `undef` in Feldspar, and by writing:

```
switch :: ∀a c. ... ⇒ Stream (a, Option c)
                         → (c → Stream a)
                         → Stream a
switch (Stream next (init :: M state1)) f
  | Stream _ (_ :: M state2) ← f undef = ...
```

we can define a switch operator that type-checks. It will however allow for the construction of ill-typed streams, since we cannot restrict the type of a created stream's existential type, and therefore risks breaking the correctness of our typecasting.

A better approach to switching in Feldspar is to implement a fixed-point-esque switch operator instead, where the initial state is given as an argument – thus avoiding any initialisation with undefined values. We can define such a switch operator by, for example, writing:

```
switch :: ∀a c. (Syntax a, Syntax c) ⇒ c → (c → Stream (a, Option c)) → Stream a
switch initC f | Stream _ (init :: M state) ← f initC =
  let
    newInit :: M (Ref state, Ref c)
    newInit = (init >>= newRef) ‘with‘ newRef initC

    newNext :: (Ref state, Ref c) → M a
    newNext (rs, rc) =
      do c ← getRef rc
         case f c of
           (Stream next _) → do
             st      ← getRef rs
             (x, ev) ← next (unsafeCoerce st)
             optionM
               (return x)
               (λ c’ → case f c’ of
                 (Stream next init’) → do
                   st’    ← init’
                   (x’,_) ← next st’
                   setRef rc c’
                   setRef rs (unsafeCoerce st’)
                   return x’)
               ev
  in Stream newNext newInit
```

The internal state consists of the current state used, coupled with the last event which occurred; initially, the internal state will contain the given event and the state constructed from it. During execution, the internal state is then used to recreate the streams and extract their produced elements. Each time an event occurs, we simply update the internal state; we did choose to ignore the first event produced after a switch, as it would otherwise have required a, possibly infinite, loop unrolling.

Unsafe type casting is still required in the above version, since the created streams' states are bound in the case alternative rather than at a top level; as we cannot access the streams' types, and thereby constrain them, we also cannot assert that their types will be equal. This is however a minor inconvenience, since the switching function, `f` above, needs to inspect its argument in order to break casting, which is abuse of Feldspar. Also, while it may seem inefficient to reconstruct a stream for each iteration, remember that these programs are simple generators and streams are abstractions which will be removed during compilation.

Using the switch operator defined above, the bouncing balls example can finally be implemented in Feldspar as well. Mimicking the Yampa version, we implement the example by writing:

```
type DPair a = (Data a, Data a)

falling_ball :: (Ord a, Numeric a, Syntax a) ⇒ DPair a → Stream (DPair a)
falling_ball (initV, initY) = zip currV currY
  where
    currV = iterate (9 ‘addNum‘) initV
    currY = scan (λacc v → acc ‘subNum'‘ v) initY currV
      where subNum' a b = (a ≤ b) ? (0, a ‘subNum‘ b)

falling_ball_ev :: (Ord a, Numeric a, Syntax a)
                 ⇒ DPair a → Stream (DPair a, Option (DPair a))
falling_ball_ev start = zipWith mergeF currPair currOpt
  where
    currPair   = falling_ball start
    currOpt    = edge $ map ((≤ 0) . snd) currPair
    mergeF a b = option (a, none) (λ_ → (a, some a)) b

bouncing_ball :: (Ord a, Numeric a, Syntax a) ⇒ Data a → Stream (DPair a)
bouncing_ball initY = loop (0, initY)
  where
    loop start = switch start $ λ(v,y) → falling_ball_ev (negate v, y)
```

`iterate` iteratively applies a function to a starting element, the successive results are then used to create a stream. `scan` produces a stream by successively applying a function to each element of the input stream and the previous element of the output stream. `zipWith` pairs together two streams using a function to combine the corresponding elements at each time instant. `option` constructs elements of Feldspar's `Option` type, and behaves as `maybe` does in Haskell for its `Maybe` type.

3.2.3 General Recurrence Equations

As we saw in the two previous chapters, the better part of Yampa's signal functions could already be realised in Feldspar as stream functions. Its family of switching operators could however not be implemented in a satisfactory manner – as they required the use of unsafe type casting. A better approach to switching in Feldspar, as it turned out, was to instead employ fixed-point styled combinators. Such recurrence equations, where a stream's output is determined by a combination of previous input and output values, is a necessary component in any network where feedback is present. Feldspar's current stream library therefore offers support for such recurrence equations, or feedback loops, through the following combinators.

```
recurrenceO  :: (Type a) ⇒
                    Vector1 a
                 → (Vector1 a → Data a)
                 → Stream (Data a)

recurrenceI  :: (Type a, Type b) ⇒
                    Vector1 a
                 → Stream (Data a)
                 → (Vector1 a → Data b)
                 → Stream (Data b)

recurrenceIO :: (Type a, Type b) ⇒
                    Vector1 a
                 → Stream (Data a)
                 → Vector1 b
                 → (Vector1 a → Vector1 b → Data b)
                 → Stream (Data b)
```

where the type `Vector1` represents a non-nested vector.

The first of the three combinators above, `recurrenceO`, takes a vector, containing the initial values of the stream, and a function for computing the stream's new output values. Its function may refer to previous outputs of the stream, but only as many as the length of the input vector. Restricting the access of previous values in this way enables efficient memory management, as the combinators need only use memory proportional in size to the input vector. Each time a new value is produced, the internal vector is updated to replace the oldest value with the new one.

The `recurrenceI` combinator operates in a similar manner as `recurrenceO` does. They do however differ through the addition of an input stream and its function restriction, as the function may now only refer to previous input values instead. The input stream of `recurrenceI` is then used in combination with the previous values stored to compute its next output. Lastly, `recurrenceIO` is simply a combination of `recurrenceO` and `recurrenceI`: it has an input stream, vectors for storing both previous inputs and outputs, and a function which may refer to both previously computed input and output values.

Given these feedback combinators, it is possible to express some simple recursively defined streams. For example, the Fibonacci sequence and a moving average filter can be defined as:

```
import qualified Feldspar.Vector as V

fib :: Stream (Data WordN)
fib = recurrenceO (V.vector [0,1]) (λ fib → fib!0 + fib!1)

slidingAvg :: Data WordN → Stream (Data WordN) → Stream (Data WordN)
slidingAvg n str = recurrenceI (V.replicate n 0) str $
                   λ input → V.sum input ‘quot‘ n
```

Vectors are used internally in both examples for storing values. However, as their semantics are equivalent to that of lists, we once again refrain from explaining them in detail; interested readers could instead refer to the Feldspar tutorial [38].

It is also possible to define finite and infinite impulse response filters using recurrence equations:

```
fir :: Vector1 Float → Stream (Data Float) → Stream (Data Float)
fir a inp = recurrenceI (V.replicate (length a) 0) inp
                        (V.scalarProd a)

iir :: Data Float → Vector1 Float → Vector1 Float
                  → Stream (Data Float)
                  → Stream (Data Float)
iir a0 a b inp = recurrenceIO (V.replicate (length b) 0) inp
                              (V.replicate (length a) 0)
                                 (λ i o → 1 / a0 ∗ ( scalarProd b i
                                                   - scalarProd a o))
```

where `scalarProd` calculates the scalar product of two vectors, and `replicate` creates a vector of the specified length and each element value.

While the stream's recurrence equation allows for feedback loops in circuits to be described, they are somewhat unintuitive to work with. This approach also quickly gets cumbersome as each recurrence equation only allows for a fixed number of buffer vectors; that is, a unique combinator is required for each conceivable use case.

4

Extending Feldspar

In the previous chapters, we discussed benefits and limitations of some current approaches for modelling streaming computations, which paradigms those were implemented in, and the current state of Feldspar's stream library. Based on the results of these studies, we now develop a model for expressing various streaming computations.

Since the current stream library is already capable of modeling the core concepts of FRP, we build our extension on top of the current stream library. Ideas from SDF that we found to be beneficial, such as clocks, are then incorporated into our extension as well. The functionality provided by our extension will however be closer to that of Yampa, whilst the ideas taken from Lucid Synchrone will be applied mainly during the compilation phase. We focus mainly on Yampa since we are lacking the ability to introduce some of the special syntax rules used in Lucid Synchrone; Lucid Synchrone relies on its syntax to make the design of automata easier and implementing them without it is unwieldy at best.

As Yampa adopted the use of arrows for its design of signal transformers, we start out by investigating the possibility of using them in our extensions as well. Afterwards, our own model of signal transformers is developed, combining ideas from FRP and SDF to achieve a healthy mix of performance and usability. Once the model has been introduced, we develop a means to compile signals into monadic programs, which can be further translated by Feldspar's compiler.

4.1 Arrows

Arrows are a generalisation of monads, as every monad produces an arrow, but not all arrows can be turned into monads. In particular, they allow notions of computation that may be partially static or take multiple inputs, while still retaining a disciplined style of composition similar to monads.

There are several reasons to prefer our language extension design to be based on arrows over, for example, an approach such as the one currently used in Feldspar's stream library. For instance, arrows are modular and introduce a meta-level of computation, helping us reason about program correctness. Another, and equally important, property of arrows is that they grant us access to the special arrow syntax. For example, consider the mathematical definition for describing the height of a free-falling ball

$$height = heigth_0 + \int velocity * \mathrm{d}t$$

In Yampa, we defined this using arrow syntax as

```
f y0 = proc v → do
  y ← (y0 +) ^<< integral ⤙ v
  returnA ⤙ y
```

Even if one is not familiar with the arrow syntax, the close correspondence between the formula's mathematical definition and the Yampa program should be clear. In Feldspar, as is common in most high-level language designs, this is the primary motivation for developing an embedded language: reducing the gap between an algorithm's specification and its corresponding program. Arrows also have a strong theoretical foundation in category theory, as they represent an alternative, and more general, formulation of Freyd categories [39].

In Haskell, arrows are given the following type class declaration

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c → cat a b → cat a c

class Category a ⇒ Arrow a where
  arr   :: (b → c) → a b c
  first :: a b c → a (b, d) (c, d)
```

Operations from the `Category` type class provide generalised versions of Haskell's identity and function composition operators, that is, categories define things connecting two types in a particular direction. The `Arrow` type class extends `Category`, but the underlying theory is the same: arrows are things that have an identity constructor and can be composed. The additional operators provided by the `Arrow` class define a way to lift arbitrary functions into arrows and how arrows can be promoted to operate on tuples, respectively. The general idea behind the second function, called `first`, is to feed the first component through the argument arrow, where the second part of the input is fed directly to the output.

Suppose that we would like to support the arrow class for our signal functions. A possible definition would then look something like:

```
data SF a b
  where
    SFarr   :: (Stream a → Stream b) → SF a b
    SFfirst :: SF a b → SF (a, c) (b, c)
    SFcomp  :: SF a b → SF b c → SF a c
    ...
```

with the intention that SF allows one to build networks whose nodes are stream transformers. Support for the arrow interface can now be implemented in a straightforward manner as:

```
import qualified Prelude         as P
import qualified Feldspar.Stream as S

instance Category (SF a b) where
  id  = SFarr P.id
  (.) = flip SFcomp

instance Arrow (SF a b) where
  arr f = SFarr (S.map f)
  first = SFfirst
```

This approach does however require us to relax the mapping function for streams, by dropping the syntax requirement; during construction one can do without the `Syntax` restriction, it is however necessary during compilation. This relaxation is necessary due to Haskell's arrow and monad type classes assuming that the guest language is a superset of Haskell's, because every Haskell function can be promoted to a guest language expression: an arrow's arr lifts arbitrary Haskell functions to an arrow and a monad's return lifts into the monad. Unfortunately, first-class functions are not viable in most embedded languages. While having to implement our own version of the monad and arrow type classes is superable, losing their associated syntax is not; as arrow syntax alleviates the cumbersome nature of writing programs in the point-free style demanded by arrows.

To model recursion, i.e. allowing cycles in the computational graph, we introduce another of the arrow type classes: `ArrowLoop`, which defines a generalised loop combinator for arrows. In Haskell, it has the following definition

```
class Arrow a ⇒ ArrowLoop a where
  loop :: a (b, d) (c, d) → a b c
```

The loop operator expresses computations in which an output value is fed back to become part of its own input in the next iteration, with the intent that any actual computation only occurs once. For our signal functions, this type class is once again too general to support: in order to feed back values between iterations they need to be stored. Storing values in Feldspar requires them to support the Syntax type class. Given that the type admits `Syntax`, a possible implementation of a loop combinator for streams would be

```
loop :: ∀a b c. (Syntax c) ⇒ (Stream (a,c) → Stream (b,c))
                           → Stream a
                           → Stream b
loop f s = let r  = newRef (undef :: c)
               ss = f $ S.zip s $ Stream getRef r
           in case ss of
             (Stream next init) → Stream next' (init ‘with‘ r)
               where next' (st, r) = do (b, c) ← next st
                                        setRef r c
                                        return b
```

where the value to feed back is stored in a reference. There is however no apparent way of feeding back values without the `Syntax` requirement, the `ArrowLoop` type class can therefore not be supported by our signal functions. Access to the arrow syntax for loops is therefore lost, and any recursively defined streams have to be manually specified, rather than by using the `rec` keyword.

Another important feature, as required by signal processing, is the ability to describe control dominated systems. In the arrow framework, this is known as conditional statements, and is described using the `ArrowChoice` class

```
class Arrow a ⇒ ArrowChoice a where
  left :: a b c → a (Either b d) (Either c d)
```

Unfortunately, we are once again hindered by the class's restrictive definition: Haskell's `Either` type cannot be made an instance of `Syntax`, since when we want to convert it to an internal representation there is no way of knowing, at least until the program is run, whether the value is constructed using `Left` or `Right`.

An interesting property of arrows can be observed when both choice and feedback are supported: the way control structures are described moves closer to the state machines of Lucid Synchrone, rather than the switching combinators of Yampa. A Mealy-styled automaton, as mentioned earlier, is basically a collection of states and transitions. As the arrows' choice combinators allow for control dominated structures to be described, it would be possible to manage the transitions of an automaton using them.

One important feature of state machines is however that only one set of equations is executed during one time instant, a property that is not enforced by the current choice operators. Additional constraints on the effects of streams are therefore required, in order to avoid interference when using the choice operators. For example, a mechanism for storing the current state of a stream for later retrieval is required, in order to ensure that no side-effects occurred. Finally, the threading of state, both local and global, in the automaton could be handled by arrows' recursion operator.

Haskell's arrows present many compelling arguments for their use in streaming computations, as well as the small gap between an algorithms description and its implementation using arrows. We are however unable to support arbitrary functions being promoted into Feldspar, as is demanded by arrows. Instead, we turn to language extensions to further examine the possibility of adopting arrows for use in our stream extensions.

4.1.1 Generalised and Rebindable Arrows

Like Haskell's arrows, generalised arrows [40] provide a means for meta-programming, that is, they enables one to write programs which generate other programs. Unlike Haskell's arrows, they allow for heterogeneous meta-programming. They achieve this by removing the assumption that any Haskell function can be promoted to an expression in the guest language.

Generalised arrows are a subclass of the Category class, and obey its laws of associativity and neutrality – as required of any category. The new arrow class defines a `ga_first` and `ga_second` function, almost identical in type and behaviour to their corresponding standard arrow functions. The use of tuples is however abstracted out, as a second type parameter of kind $(* \to * \to *)$ is used instead. This type parameter is called the tensor of a generalised arrow, and its role is analogous to that of tuples in Haskell's arrows. Missing from the class interface is the arr function, used for lifting arbitrary functions into the arrow. In its place are instead a number of functions for rearranging the inputs and outputs of a generalised arrow. Conceptually, as the arr function was used in describing the laws and behaviour of arrows, these functions are required for upholding those laws. The type class itself is given the following definition:

```
class Category g ⇒ GArrow g (**) u
  where
    ga_first     :: g x y → g (x**z) (y**z)
    ga_second    :: g x y → g (z**x) (z**y)
    ga_cancell   :: g (u**x) x
    ga_cancelr   :: g (x**u) x
    ga_uncancell :: g x (u**x)
    ga_uncancelr :: g x (x**u)
    ga_assoc     :: g ((x**y)**z) (x**(y**z))
    ga_unassoc   :: g (x**(y**z)) ((x**y)**z)
```

Subclasses are also provided, accounting for languages with, for example, recursion:

```
class GArrow g (**) u ⇒ GArrowLoop g (**) u
  where
    ga_loop :: g (x**z) (y**z) → g x y
```

Generalised arrows solve some of the problems we encountered while trying to support arrows for our stream transformers. For instance, we no longer need to support promotion of arbitrary functions, and the strict types are substituted with a type parameter. These restrictions are however not enough, as we also require the contexts to only accept types admitting Syntax. There is in fact a recent language extension to GHC which allows us pass constraints as type arguments, allowing us to arbitrarily restrict the types that arrows accept. A version of arrows that fits our requirements could then be defined as:

```
{-# LANGUAGE ConstraintKinds #-}

class Category a ⇒ Arrow a (**) ctx
  where
    arr  :: (ctx x, ctx y) ⇒ (x → y) → a x y
    ...

class Arrow a (**) ctx ⇒ ArrowLoop a (**) ctx
  where
    loop :: (ctx z) ⇒ a (x**z) (y**z) → a x y

class Arrow a (**) ctx ⇒ ArrowChoice a (**) ctx (++)
  where
    left :: (ctx z) ⇒ a x y → a (x++z) (y++z)
```

Lifting is possible since we only permit a possibly restricted set of types.

Sadly, however, by writing our own versions of arrows we are prohibited from using their associated syntax. While GHC allows most of its built-in syntax to be rebound by the user, through the rebindable syntax language extension, it is too restrictive for our proposed version, as the types of the new arrow functions must match the original types very closely.

4.2 Signals

As Haskell's arrows proved to be incompatible with Feldspar's restricted language, we instead focus on developing our extensions through pure Haskell code. Arrows, and the meta- programming they offered, did however introduce a nice way of expressing streaming computations, which we will aim to mimic in our model.

As we build our extension on top of Feldspar's stream library, we require a way to promote stream functions into our language – similar to how arrows provides the arr operator for lifting arbitrary functions into arrows. Unlike arrow's lift operator, it is necessary to restrict the types for which promotion is possible. Failure to do so would enable the construction of streams without any internal representation in Feldspar, that is, it would create streams which are impossible to compile. This reasoning leads us to the following definition of `lift`:

```
lift :: (Syntax a, Syntax b) ⇒ (Stream a → Stream b)
                             → Signal a
                             → Signal b
```

as `lift` represents our way of promoting stream functions into signals. We can define our first constructor for `Signal`, and, based on the type of `lift`, we give the initial definition of:

```
data Signal a
  where
    Arr :: (Syntax a, Syntax b)
        ⇒ (Stream a → Stream b)
        → Signal a
        → Signal b
```

Like arrows, the need for an underlying Category class becomes evident when implementing signal generators. Consider, for example, the `repeat` generator, which produces a constant stream of some value:

```
repeat :: (Syntax a) ⇒ a → Signal a
repeat a = Arr (S.repeat a ‘asTypeOf‘) $ ...
```

Currently, we have no input signal to feed the `Arr` constructor with, and neither would we want to add one as it would defeat the purpose of having generators in the first place. The introduction of another construct is therefore necessary, modeling the identity morphism of categories:

```
data Signal a
  where
    ...
    Bot :: Signal a
```

This construct will be hidden in the final library, effectively stopping users from creating their own signal generators and the potential memory issues those may incur.

Defining signals, and `map`, in this way introduces a subtle ambiguity to signals: how to distinguish between pairs of values and a pair of two independently constructed types. This difference is perhaps of lesser importance when considering streams as seen from the user's perspective, as they are semantically equivalent. During the complation process, the ability to distinguish between the two becomes significantly more important. For instance, while a tuple of values is bound to be evaluated simultaneous, the product of two independently created signals can be computed in parallel and later joined when both are needed. Furthermore, given a way to distinguish between the two types of pairs, it is possible to optimise and balance operations which only affect one element by, for example, rearranging combinatorial functions.

For deeply embedded data types, such as our `Signal` type, library operations only build an interim representation of the data structure that reflects the expression tree, that is, programs written using signals produce a tree over the network, which we can later inspect and optimise. Tracing a network's graph would reveal whether the pair is a result from a use of the `map` function or pair of combined signals, originating from a certain constructor. Introducing a constructor for pairing signals allows us to detect opportunities for parallel evaluation. It is however not enough to allow transformations on pairs to be rearranged in some favourable order. Consider, for example, the mapping over a signal of pairs using `map`:

```
fst :: (...) ⇒ (Stream (a,b) → Stream (a,c))
             → Signal (a,b)
             → Signal (a,c)
fst = map
```

While the type may appear to be appropriate, it does not guarantee that the function's effect is contained to only the tuple's second element. For example, consider the following function

```
bad :: (...) ⇒ Signal (a, a) → Signal (a, a)
bad = fst $ S.map $ λ(x, y) → (y, x)
```

The addition of another constructor is therefore necessary, and we introduce the following signal transformers:

```
zip :: (Syntax a, Syntax b) ⇒ Signal a → Signal b → Signal (a, b)

fst :: (Syntax a, Syntax b) ⇒ Signal (a, b) → Signal a
```

accompanied by a corresponding extension to our `Signal` type

```
data Signal a
  where
    ...
    Zip :: (Syntax a, Syntax b) ⇒ Signal a → Signal b → Signal (a, b)
    Fst :: (Syntax a, Syntax b) ⇒ Signal (a, b) → Signal b
```

Using the `Arr` constructor it is possible to express arbitrary combinatorial circuits, as it gives us access to Feldspar's stream library.

In Feldspar's stream library, sequential circuits are currently expressed using the recurrence equations and sampling operator. However, for reasons discussed in earlier chapters, we would prefer to avoid using them entirely. Dedicated constructors for sequential operations are therefore introduced, one for delay and another for sampling.

```
delay  :: (Syntax a) ⇒ a → Signal a → Signal a

sample :: (Syntax a) ⇒ Data Length → Signal a → Signal a
```

These are added with the intent to enable the use of memory efficient buffers in feedback networks, where the size and access patterns for each buffer would be determined by inspection of the program's graph. Both functions are inspired by their corresponding versions in Lucid Synchrone, where `delay` produces a signal initialised to some value, similar to the `fby` operator, and `sample` simply samples a signal at a specified rate. These transformers are once again accompanied by a corresponding extension of the `Signal` type:

```
data Signal a
  where
    ...
    Delay  :: (Syntax a) ⇒ a → Signal a → Signal a
    Sample :: (Syntax a) ⇒ Data Length → Signal a → Signal a
```

which gives us the final definition of our Signal type:

```
data Signal a
  where
    Arr    :: Stream b → Stream a) → Signal b → Signal a
    Zip    :: Signal a → Signal b → Signal (a, b)
    Fst    :: Signal (a, b) → Signal a
    Delay  :: a → Signal a → Signal a
    Sample :: Data Length → Signal a → Signal a
    Bot    :: Signal a
```

where constraints have been elided in order to improve readability.

4.2.1 Meta-Programming using Haskell

When we defined our signals any pure Haskell code was carefully separated from our signal functions. Having such a division between the languages, coupled with the distinction between Feldspar types and Haskell types, has the nice property that we can use pure Haskell code as a form of meta-programming language for our signal programs.

The general idea is that any recursion created by using functions from our signal library results in feedback, while recursion created by using pure Haskell code produces repetitive code instead. For example, consider the following sequential function

```
f :: (Syntax a, Num a) ⇒ Signal a → Signal a
f sig = sig + delay 0 sig
```

Recursion is present here through the delay operator, creating a function whose output depends upon previous input values. This is reflected in the graph representation, and by observing the sharing present, the graph can be viewed in Figure 4.1.

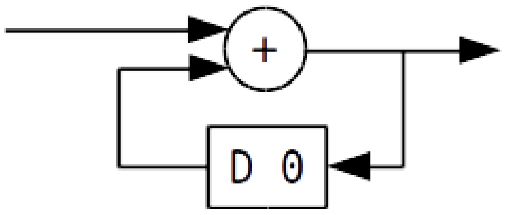

Figure 4.1: Small feedback network with delay

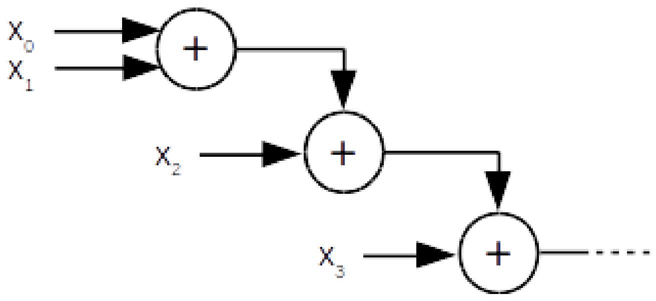

Figure 4.2: Summation of signals

Conversely, by defining a pure Haskell function for summing the outputs from a list of signals as, for example:

```
import qualified Prelude as P

sums :: (Syntax a, Num a) ⇒ [Signal a] → Signal a
sums = P.foldr1 (+)
```

the recursion is instead introduced through the folding operator. This repetitive application of the plus operator is reflected in the network generated, which takes the structure shown in Figure 4.2. Pure Haskell code can hence be used to create repetitive signal programs, enabling it to be used as meta-programming language for signals.

The ability to use pure Haskell in this way presents several benefits, as it improves the syntax and ease of programming signals significantly. For instance, in order to define complex networks, the user is only required to be versed in Haskell's standard library operators, thereby reducing the complexity of developing new networks. An additional benefit is that delays can now be applied to any expression, unlike the previous recurrence equation. The effect of these benefits can be observed when we, for example, redefine the FIR filter to use our new signals instead. Before we do that however, we define two additional helper functions:

```
delays :: (Syntax a) ⇒ [a] → Signal a → [Signal a]
delays (a:as) sig = P.init $ P.scanl (flip delay) (delay a sig) as

muls :: (Syntax a, Num a) ⇒ [Signal a] → [a] → [Signal a]
muls = P.zipWith $ λsig c → sig ∗ repeat c
```

`delays` creates a list of successively delayed signals, and `muls` applies point-wise multiplication to a list of signals. These functions allow us to express the FIR filter in a rather eloquent manner as:

```
fir :: Int → [Data Float] → Signal (Data Float) → Signal (Data Float)
fir bs sig = sums $ muls bs $ delays ds sig
  where
    ds = P.replicate (length bs) 0
```

Even though we have now managed to define a FIR filter using signals, we have yet to implement any support for its compilation. In fact, trying to naively compile the data structures created by any of the above examples would most likely produce very inefficient code. This is due to the fact that we cannot observe any sharing that is taking place, and any piece of shared code will hence be run once for each invocation. For instance, in the above example of a FIR filter, each delayed signal in the list created by delays would contain an unique copy of the entire input signal. To circumvent this, we employ the techniques of type-safe observable sharing.

4.3 Type-Safe Observable Sharing of Signals

A common problem in pure and lazy functional programming languages, and especially so when manipulating data types representing embedded languages, is observing any sharing present in the syntax trees. While an embedded language is commonly implemented as finite graphs over its base types, viewing it as an algebraic data type makes it indistinguishable from an infinite tree.

Observable sharing in trees extracted from some embedded language, is a well- established problem: how to allow trees extracted from some deeply embedded language to have observable back-edges; such sharing is currently invisible to any function which traverses the tree. One possible solution to the problem, as proposed by [41], is the use of explicit labels. By associating every component in the language with a unique tag it is possible to recover an expression's graph structure, as we can keep track of visited nodes when traversing the tree. This does however obfuscate the original data structure and introduces the problem of supplying the source of unique names.

Monads present another solution to the problem of allowing observable sharing, where some underlying machinery is used to guarantee unique tags. This is the solution used in early implementations of Chalmers Lava [42], where a changing piece of state is threaded through the computations and used to generate unique tags. Introducing monads – or any other categorical structure – does however impact the types of the primitive components; it implies that they become monadic, that is, functions are restricted to returning monadic types. Restricting the functions in such a way affects the manner in which one writes expressions. For instance, when using monadic types it is no longer possible to use normal recursion or function abstraction, one is instead forced to use their monadic variants, such as `mdo` and `bind`.

Another approach to observable sharing, as proposed by [43], relies on the GHC-specifics of stable names to provide an `IO` function capable of observing sharing directly. Stable names are used in order to identify shared nodes in a data structure, as they provide pointer equality for Haskell objects. Supporting observable sharing only requires us to provide a way to reference into a specific type and map over its dereferenced internal constructs. This is done by instantiating the following class for all related data types:

```
class MuRef a where
    type DeRef a :: ∗ → ∗

    mapDeRef :: (Applicative f) ⇒
          (∀b . (MuRef b, DeRef a ~ DeRef b) ⇒ b → f u)
                   → a
                   → f (DeRef a u)
```

mapDeRef takes as parameter a higher-ranked function and an abstract syntax tree of the original data type, from which it returns a value of the associated type family where each recursive value is mapped through the function to produce a materialised version of the graph. This process is commonly known as reification: to take something abstract and regard it as material instead; a reified type is simply a value that represents a type.

Given the mapDeRef function, it is possible to reify any syntax tree admitting MuRef through the use of the reifyGraph function, which is given the following type:

```
reifyGraph :: (MuRef t) ⇒ t → IO (Graph (DeRef t))
```

reifyGraph takes a syntax tree and returns a graph representation of the dereferenced node's, where a nodes children are abstract references rather than recursive values. The actual graph is represented as a sort of linked list, and has the following type:

```
type Unique  = Int
data Graph e = Graph ([(Unique, e Unique)], Unique)
```

Employing observable sharing does in general imply a recursive data type is used to represent the embedded language, together with a mirror type, where the points of recursion have been replaced by abstract references. For example, consider the following example of a small embedded language for arithmetic operations, modeled after the one used in [44]:

```
data Exp
  where
    EAdd :: Exp → Exp → Exp
    ENeg :: Exp → Exp
    EInt :: Int → Exp
```

Constructing the mirror type is then as simple as writing:

```
data Tree ref
  where
    TAdd    :: ref → ref → Tree ref
    TNeg    :: ref → Tree ref
    TInt    :: Int → Tree ref
```

If we then want to enable reification of our small language, we need only implement support for the above MuRef class:

```
import Control.Applicative

instance (...) ⇒ MuRef Exp
  where
    type DeRef Exp = Tree
    mapDeRef f node = case node of
      (EAdd x y) → TAdd <$> f x <*> f y
      (ENeg x)   → TNeg <$> f x
      (EInt i)   → pure $ TInt i
```

While it may seem as though `mapDeRef` simply traverses the syntax tree, translating nodes as it goes, it also manages to detect any sharing present during its traversal. It does so by wrapping each node in the applicative, `f`, which is then used to keep track of the translated nodes. Conceptually, this process is similar to using a lookup table, where nodes that have already been referenced are fetched from the table, rather than being translated again.

Given these types and instances, and a suitable instance of Haskell's `Num` class for `Exp`, we can finally reify expressions in our little language. For instance, the following expression can be successfully reified into its graph representation:

```
reifyGraph (let i = (TInt 2) in negate (i + i) :: Exp)

> let [(1, TNeg 2), (2, TAdd 3 3), (3, TInt 2)] in 1
```

where the references are represented as integers. While it certainly is neat to detect sharing in simple expressions, we also require observable functions in order to observe the sharing present in signal transformers. Luckily, it turns out that observing functions is possible using the same machinery as for simple expressions.

Traditionally, functions have been observed by applying them to a dummy argument, and observing where the argument occurs inside the resulting expression. This was the approach used by Kamin and Elliott [44, 45], where they adopted the host language's functions and simply extended their base types with support for variables[1]. Using variables to support observable functions, as done in Elliott's paper [44], requires threading a namespace to all points where functions are examined. With observable sharing, we can observe the sharing present in functions without needing to introduce unique names, by instead observing the sharing created by let bindings inside the observed functions. This idea is captured by the following class and function:

[1]While using exceptions as dummy arguments seems to be a simpler solution, they could potentially yield expressions which have been evaluated in an unsound way.

```
import Data.Dynamic

class NewVar a
  where
    mkVar :: Dynamic → a

capture :: (Typeable a, Typeable b, NewVar a) ⇒ (a → b) → (a, b)
capture f = let a = mkVar (toDyn f) in (a, f a)
```

where `capture` takes a function and returns the function's argument and result; `Dynamic` is used to create unique labels for input signals. While these labels do not admit equality, they are used internally during reification to check whether any two variable expressions can be declared comparable. In order to support observable functions for our earlier little language, `Exp`, we simply need to extend the data type with a variable constructor. For its graph representation, `Tree`, we add nodes for both variables and lambda terms:

```
data Exp
  where
    EVar :: Dynamic → Exp
    ...

data Tree ref
  where
    TLambda :: ref → ref → Tree ref
    TVar    :: Tree ref
    ...
```

Observing functions is then simply a matter of supporting the required type classes:

```
instance NewVar Exp
  where
    mkVar = EVar

instance (...) ⇒ MuRef Exp
  where
    type DeRef Exp = Tree
    mapDeRef f node = case node of
      (EVar _) → pure $ TVar
      ...

instance (...) ⇒ MuRef (a → b)
    where
    type DeRef (a → b) = Tree
    mapDeRef f fn = let (v, g) = capture fn in TLambda <$> f v <*> f g
```

We can now observe the sharing present in functions:

```
reifyGraph (λx → negate (x + x) :: Exp)

> let [(1, TLambda 2 3), (2, TVar), (3, TNeg 4), (4, TAdd 2 2)] in 1
```

4.3.1 Sharing in Signals

In order to adapt the techniques of observable sharing for use with our signal type, the introduction of an additional data type is required. This will be `signal`'s version of the mirror type, as it is used during the reification process to represent our signals as a directed graph over signal constructors. Also, for the sake of detecting sharing present in single functions, our signal type will require an additional constructor for dynamic types to support the `NewVar` class.

Introducing a new variable constructor to signals turns out to be a simple enough endeavour. As none of the signal transformers attempts to inspect, i.e. pattern match, on its inputs signals, an additional constructor will not affect the current implementation. `Signal` can thus be safely extended in the following manner:

```
data Signal a
  where
    ...
    Var :: Dynamic → Signal a
```

The matter of supporting an instance declaration of `NewVar` for `Signal` is then solved by a simple use of the variable constructor:

```
instance NewVar (Signal a)
  where
    mkVar = Var
```

Now that our `Signal` type supports variables, the next step to enable observable sharing is to introduce a mirror type. Constructing such a mirror type from `Signal` is easy enough, as each one of `Signal`'s constructors is simply translated into node form by replacing all the recursive types with an abstract reference. Streams functions are kept as leaves in the mirror type rather than having them translated as well, since keeping them around allows us to later fuse them by joining nodes of consecutive `Arr`s. We then define our mirror type from these translated nodes, with additional ones added for lambda abstractions and variables. This results in the following type:

```
data Tree ref
  where
     -- Notating functions
    TLambda :: ref → ref → Tree ref
    TVar    ::               Tree ref

     -- Signal specific
    TArr    :: (Stream a → Stream b) → ref → Tree ref
    TZip    :: ref → ref → Tree ref
    TFirst  :: ref → Tree ref
    TDelay  :: a → Tree ref
    TSample :: Data Length → Tree ref
    TBot    :: Tree ref
```

Defining a `MuRef` instance for `Signal` is then done in essentially the same way as for our little arithmetic language from the previous chapter: each constructor is translated into its corresponding version in the mirror type, applying the given function to each recursive type parameter.

```
instance (Syntax a) ⇒ MuRef (Signal a)
  where
    type DeRef (Signal a) = Tree
    mapDeRef f signal = case signal of
      (Arr    g sig) → TArr    g <$> f sig
      (Delay  a sig) → TDelay  a <$> f sig
      (Sample i sig) → TSample i <$> f sig
      (Zip x y)      → TZip <$> f x <*> f y
      (Fst x)        → TFst <$> f x
      (Bot  )        → pure $ TBot
      (Var _)        → pure $ TVar

instance ( MuRef a, Typeable a, NewVar a
                 , MuRef b, Typeable b
                 , DeRef a ~ DeRef (a → b)
                 , DeRef b ~ DeRef (a → b)) ⇒ MuRef (a → b)
  where
    type DeRef (a → b) = Tree
    mapDeRef f fn = let (v, g) = capture fn in TLambda <$> f v <*> f g
```

The constraints are required in order to assert that our signals provide the required functionality demanded by, for example, the `capture` function. Our data types are therefore extended with Syntax and Typeable constraints on their quantified types, wherever such constraints are necessary.

Once the correct constraints are in place, we are finally able to reify any signal expression or function, and observe their resulting graph structure. For example, consider the following sequential signal function:

```
fir1 :: Signal (Data Float) → Signal (Data Float)
fir1 sig = sig + delay 0 sig
```

Reification of this function is certainly possible, since `Data Float` admits all the necessary type constraints, and produces the following graph representation:

```
reifyGraph fir1

> let [ (1, TLambda 2 3)
> , (2, TVar)
> , (3, TArr _ 4)
> , (4, TZip 2 5)
> , (5, TDelay 0 2)
> ] in 1
```

Relying on non-recoverable sharing can however be quite fragile, as the original network is lost after reification and should therefore in general be used with care when detecting additional sharing may introduce errors. In the case of signals, the effects of introducing more sharing would not observably change the evaluation of signals and is therefore an acceptable tool. It is however possible to construct signal transformers where additional sharing would negatively affect performance. For instance, in the following example, unrolling the computationally heavy function into both the left and right signals means it will be computed twice – hurting its performance.

```
f :: Signal (Data Float) → Signal (Data Float)
f sig = zip left right
  where sig'  = compute_meaning_of_life sig
        left  = sig' + 1
        right = sig' - 1
```

4.4 Compiling Signals to Streams

Compiling our signal transformers is the act of transforming them into monadic expressions within Feldspar, where the expressions represent the computations necessary to retrieve a transformed version of the input data which, in most cases, comes from another compiled signal. Nevertheless, the compilation process amounts to a traversal of the signal transformers' reified graph structure, where nodes are recursively computed and then connected together.

Relying solely on Feldspar's monad during compilation is however a fragile approach, as the monad makes no guarantees for how expressions are shared. One solution to this problem, which we chose to pursue, is to store the compiled expressions in references. As references allow us to name shared expression, they give a finer control over how values are accessed and stored. However, as the return types may differ between nodes, this requires the use of dynamic values.

Haskell's `Dynamic` type, previously used in the observable sharing section, provides a potential solution to generalising the types of compiled nodes. However, observing the types of references and dynamic values reveals a shortcoming of this approach: their type constraints do not align. For instance, consider the functions for creating and reading the values in references and dynamic types, respectively.

```
newRef :: Syntax a ⇒ a → M (Ref a)
getRef :: Syntax a ⇒ Ref a → M a

toDyn       :: Typeable a ⇒ a → Dynamic
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a
```

References require that their type argument admit Feldspar's `Syntax` class, while dynamic types require types admitting `Typeable`. Sadly, references do not instantiate `Typeable`, and dynamic types certainly do not instantiate `Syntax`. It is therefore necessary to introduce a new data type, modeling dynamic references:

```
data Dyn
  where
    Dyn :: Typeable a ⇒ Ref a → Dyn
```

We build these around the `Typeable` class, and provide the following functions for constructing new and dynamically casting `Dyn` types:

```
toDyn :: Typeable a ⇒ Ref a → Dyn
toDyn = Dyn

fromDyn :: Typeable a ⇒ Dyn → Maybe (Ref a)
fromDyn (Dyn a) = gcast a
```

The casting function, `fromDyn`, uses the generalised casting operator `gcast`, from Haskell's `Dynamic` library, in order to cast a type inside a constructor – a reference in this case.

Dynamic references provide us with the generalised type we needed for unifying return types of compiled nodes, which in turn allows us to sketch the skeleton of a recursive compiler:

```
import qualified Data.Map as Map

type Node = (Unique, Tree Unique)

compGraph :: (...) ⇒ Graph Tree → M a → M b
compGraph (Graph gnodes groot) input =
fromJust $ fromDyn $ compNode (find groot) empty
  where
    compNode :: Node → Map Unique Dyn → M Dyn
    compNode (i, node) nodes
      | Just dyn ← lookup i nodes = return dyn
      | otherwise = case node of
        ...

    insert :: Unique → x → Map Unique x → Map Unique x
    insert = Map.insertWith (λ _ x → x)

    lookup :: Unique → Map Unique x → Maybe x
    lookup = Map.lookup

    find   :: Unique → Node
    find   = ... // Find node in graph, assume that it exists
```

The intention is that `compGraph` starts the compilation process by calling `compNode` on the root node, after which `compNode` recursively compiles the entire graph. As different nodes may refer to the same sub-nodes, due to sharing being observed, all nodes are compiled once and then placed in a look-up table. Further references to already compiled nodes will then be redirected to use those stored in the table instead.

The compilation process of each individual node follows the same general pattern: firstly, the node fetches its input, it then computes an output value, and lastly a reference to said output is returned. For instance, variables, which simply return a reference to the input, are compiled in the following manner:

```
compNode ... = case node of
  (TVar) → input >>= newRef >>= return . toDyn
```

where `input` refers to the outer input method, brought into scope by the `compGraph` function. While other nodes may contain function application, several inputs, etc., the general compilation strategy remains the same. For example, nodes for lambda and function abstraction are compiled in the following manner:

```
compNode ... = case node of
  (TLambda var fun) →
    do var' ← compNode (find var) nodes
       fun' ← compNode (find fun) (insert var var' nodes)
       return fun'
  (TArr fun sig) →
    do sig' ← compNode (find sig) nodes
       let x = fromJust $ fromDyn $ sig'
           y = ... // Apply fun to a stream of references into x
                   // and return the stream's next function
        r ← y >>= newRef
        return $ toDyn r
  ...
```

We can now reify signals into their graph representations and turn those graphs into monadic expressions; furthermore, Feldspar's compiler allows us to turn those expressions into compiled code. The last step required, in order to support compilation for our signal transformers, is therefore to combine the above techniques. We start out by combining the first two of these techniques into a single function, giving it the following type:

```
comp :: (...) ⇒ (Signal (Data a) → Signal (Data b))
             → IO (Data [a] → Data [b])
```

`comp` receives a signal transformer as input and produces as result a monadic function. While the produced function represents the signal transformation, it operates over chunks of data rather than a continuous stream. The use of chunks, or lists, is required due to a constraint in the current version of Feldspar's compiler, which we use in combination with our `comp` function to generate compiled code:

```
import Feldspar.Compiler

compSF :: (...) ⇒ (Signal (Data a) → Signal (Data a)) → IO ()
compSF sf = comp sf >>= icompile
```

Since chunks represent a new kind of abstraction, we will introduce the concept of mutable arrays before implementing `comp`.

Mutable arrays are conceptually similar to references, only they contain multiple values instead of one. The type constructor for mutable arrays is `MArr`, and operations which create, update and query these arrays all belong to Feldspar's `M` monad:

```
newArr :: Type a ⇒ Data Length → Data a → M (Data (MArr a))
setArr :: Type a ⇒ Data (MArr a) → Data Index → Data a → M ()
getArr :: Type a ⇒ Data (MArr a) → Data Index → M (Data a)
```

There are also operations that convert between mutable and immutable arrays of the same type, namely

```
freezeArray :: Type a ⇒ Data (MArr a) → M (Data [a])
thawArray   :: Type a ⇒ Data [a] → M (Data (MArr a))
```

They are however still inside the `M` monad. In order to escape the mutable monad, Feldspar provides the following run function:

```
runMutableArray :: Type a ⇒ M (Data (MArr a)) → Data [a]
```

Using mutable arrays, we can represent the chunks required by our `comp` function and implement it in the following way:

```
comp sfun = do
    graph ← reifyGraph sfun
    return $ λ input → runMutableArray $
      do let inarr = arrify    input
             len   = getLength input
             fun   = compileGraph graph inarr

         // As required by Feldspars compiler,
         // we process the input in chunks
         out ← newArr_ len
         forM len $ λ ix → fun >>= setArr out ix
         return out
  where
    arrify :: (...) ⇒ Data [a] → M (Data a)
```

Compilation consists of three stages: reification of the input graph, constructing a monadic expression from the reified graph and finally transforming each input chunk by applying the transformer to it.

4.4.1 Pre-processing the Graph

Even though we managed to implement a compiler for our `Signal` class, the efficiency of the compiled code is still unsatisfactory. In order to fix this, it is important that we address the problem of creating memory efficient buffers for use with delayed signals, which we lost when replacing recurrence equations with the more general `Delay` constructor.

Using individual delays for each single value does allow for efficient memory management, as the combined size of each delay element is proportional in size to those of the buffers used by Feldspar's recurrence equations. The main problem with storing delayed values in such a decentralised manner is that each element has to be updated each time a new input value is read. As delays typically appear in chains, a better approach is to instead associate each delayed signal with a circular buffer – when using buffers, we only update one value and its counter. These buffers could then be given individual sizes, each one proportional to the number of delays applied to their respective signals. The intent is that we put new values into the buffer whenever input is fetched, while delayed values are accessed by simply indexing backwards in the buffer. Buffers are given the following generalised type:

```
data Buffer a = Buffer {
      getBuf :: Data Index → M a
    , putBuf :: a → M ()
    }
```

Given this type, we can create circular arrays by, for example, writing:

```
newBuffer :: Syntax a ⇒ Data Length → a → M (Buffer a)
newBuffer l init = do
  buf ← newArr l $ desugar init
  ir  ← newRef (0 :: Data Index)
  let get j = do i ← getRef ir
                 fmap sugar $ getArr buf ((l+i-j-1) ‘mod‘ l)
      put a = do i ← getRef ir
                 setRef ir ((i+1) ‘mod‘ l)
                 setArr buf i $ desugar a
  return (Buffer get put)
```

where `sugar` and `desugar` translate types between Feldspar's internal representation and the frontend types commonly used in programs. The circular behaviour is achieved by using the `mod` operator, which is used to traverse the internal array.

Even though circular arrays allow for memory efficient storage, forcing users to write delays with circular buffers in mind would clutter the algorithm descriptions. We would much rather stick to using delays as before, and have the creation of buffers taken care of during a stage before compilation. A pre-processing stage is therefore added, executed before compilation, with the intent to transform and optimise a reified signal's graph before it is compiled. All the necessary buffers are created during this stage, and all leaf nodes referring to delayed signals are translated into pointers to such buffers.

Variable nodes still reads input as usual, but they will now also put the newly read values into their associated buffers as well; referencing variable nodes is then the same as reading the head value of its associated buffer. Delayed nodes are replaced with references to the delayed signal's buffer, along with a number corresponding to their position in the delay chain. This number is then used to index into the correctly delayed value of the buffer.

The Tree type is extended with two constructors in order to support these new buffer nodes:

```
data Tree ref
  where
    ...
    TBVar   :: ref → Tree ref
    TBDelay :: ref → Data Index → Tree ref
```

where `TBVar` and `TBDelay` are the translated version of `TVar` and `TDelay`, respectively.

The process of translating nodes of a graph, into their buffered versions, is performed in three steps: first we detect and mark chains of delays, all referencing some common variable node; marked nodes are then translated into their pointer versions, with buffers created for each chain; the new nodes are then substituted into the graph and the buffer is filled with its initial values. We extend our previous compiler to use buffers in the following way:

```
compGraph :: (...) ⇒ Graph Tree
             → Map Unique (Buffer a)
             → M a
             → M b
compGraph (Graph gnodes groot) buffers input = ...
```

We also add the following cases to the recursive compiler:

```
compNode ... = case node of
  (TBVar r)   →
    do let buf = fromJust $ lookup r buffers
        v  ← input
        putBuffer v buf
        r  ← getBuffer 0 buf >>= newRef
        return $ toDyn r
  (TBDelay r i) →
    do let buf = fromJust $ lookup r buffers
       r ← getBuffer i buf >>= newRef
       return $ toDyn r
  ...
```

Armed with this new compiler, it is now possible to define signal transformers in a true DSL style and have them generated as target specific code with similar efficiency to Feldspar's stream library. For instance, the earlier FIR filter, as defined in chapter 4.2.1, can now be compiled into C code when given an appropriate input list of coefficients. One such use case of the filter, coupled with a call to the compiler, could be:

```
comp (fir [1, 2, 3 :: Data Float]) >>= icompile
```

Which yields the following C code when run, where parts not relevant to the filter have been elided in order to improve readability:

```
void fir(struct array * v0, struct array * out)
{
    ...
    v53 = getLength(v0);

    // buffer, stored at 'e1', is filled with its initial values
    e2 = ((e2 + 1) % 3);
    at(float,&e1,e2) = 3.0f;
    e2 = ((e2 + 1) % 3);
    at(float,&e1,e2) = 2.0f;
    e2 = ((e2 + 1) % 3);
    at(float,&e1,e2) = 1.0f;

    // output array is allocated and filled with zeroes.
    initArray(&e9, sizeof(float), v53);
    for(uint32_t i = 0; i < v53; i += 1)
    {
            at(float,&e9,i) = 0.0f;
    }

    for(uint32_t v14 = 0; v14 < v53; v14 += 1)
    {
        // set index pointers
        e0 = ((e0 + 1) % v53);
        e2 = ((e2 + 1) % 3);

        // index into the buffer
        e14 = at(float,&e1,(((e2 + 3) - 1) % 3));
        e15 = e14;
        e17 = at(float,&e1,((((e2 + 3) - 1) - 1) % 3));
        e18 = e17;

        // perform the transformation
        e19 = 1.0f;
        e20.member1 = e18;
        e20.member2 = e19;
        e21 = (e20.member1 * e20.member2);
        e23 = at(float,&e1,((((e2 + 3) - 2) - 1) % 3));
        e24 = e23;
        e25 = 2.0f;
        e26.member1 = e24;
        e26.member2 = e25;
        e27 = (e26.member1 * e26.member2);
        e29 = at(float,&e1,((((e2 + 3) - 3) - 1) % 3));
        e30 = e29;
        e31 = 3.0f;
        e32.member1 = e30;
        e32.member2 = e31;
        e33 = (e32.member1 * e32.member2);
        e34.member1 = e27;
        e34.member2 = e33;
        e35 = (e34.member1 + e34.member2);
        e36.member1 = e21;
        e36.member2 = e35;
        e37 = (e36.member1 + e36.member2);

        // set output value
        at(float,&e9,v14) = e37;
    }

    initArray(out, sizeof(float), getLength(&e9));
    copyArray(out, &e9);
    freeArray(&e1);
    freeArray(&e9);
}
```

The generated code is concise in its content and makes use of a circular buffer for efficient memory storage. There are however some variables having constant values assigned to them inside the for-loop, which should ideally be put before the loop – any standard C compiler should however be capable of solving this. Even the initialisation of some arrays might be beneficial to have performed outside of the actual function, given that we know that the function will be called often, and the array will probably live throughout the program's entire lifetime. An improvement would therefore be have the initialisation, transformation, and termination stages of a signal transformer separated. This extension would possibly result in three new functions: `fir_init`, `fir`, and `fir_term`, separating the signal transformer's algorithm from its memory management.

There is also a number of C-structures, used here for modeling tuples, appearing throughout the generated code, but there is no mention of tuples ever being used in the function's Haskell code. Tuples are however used during the compilation process to join together two references in a zip node, and, unfortunately, they remain in the generated code. The use of tuples are in no way required and should ideally be substituted for some data structure which is only used during compilation. A possible solution is to use an abstract data type instead, which models the virtual pairs of references used during compilation:

```
data VRef a
  where
    Leaf :: Ref  a → VRef a
    Tup  :: VRef a → VRef b → VRef (a, b)
```

These virtual references will then be unpacked each time a node needs access to the actual references inside. The unpacking and packing process of virtual references will however be done during compilation, which means the generated code will be free of unecessary constructs.

Furthermore, the program contains a number of renaming operations, that is, it contains empty operations where new variables are created with the single purpose of having an already computed value assigned to them. This is due to Feldspar's compiler interpreting each monadic statement as a executable program, with an associated variable assignment. Hence, whenever a line of Haskell code occurs in a monadic expression, which will eventually be flattened out by the compiler – like reading a reference or pattern matching using a case-statement – it results in a new variable. These superfluous variables are however also removed by an C compiler later on, as most compilers are capable of reducing expressions through simple copy propagation.

5

Using Signals

In the previous chapters we have looked at some of this project's related work, compared different approaches to find the one which fit Feldspar's requirements, and then finally implemented our own data type for modeling signals. It is now our intention to illustrate how the new data type can be used for writing various signal processing algorithms. We will do so by introducing some of the signal library's core functionality and show how to derive complex signal transformers from them. A number of examples from the previous sections will be implemented using our new library as well.

By implementing previously used examples, such as bouncing balls and the power function, we can compare their implementation using our signals to their corresponding versions in related languages. The examples will also serve as a means to analyse the current signal library and its abilities to handle irregular streams – which is necessary for modeling the changing behaviour of the bouncing balls example.

In Yampa and Lucid Synchrone, signals with irregular behaviour were model by special operators: Yampa used a family of switching functions and Lucid Synchrone used functions dedicated to merging streams. Feldspar's stream library does however lack support for irregular streams, as we saw in chapter 3.2.1, where the introduction of a new function was necessary in order to model such streams. We will show that these examples can be implemented in the signals library without the need to introduce new functions, as they can be modeled by using recursively defined streams and a multiplexer instead.

Furthermore, we introduce some of the related languages which weren't mentioned during the background discussion. These are all languages that served as inspiration during the development of our library, but were omitted from the earlier discussion in order to keep it concise. They are therefore presented at the end of this chapter instead, where we discuss any similarities they share with our library and their intersting language features.

5.1 The Signal Library

Before we begin implementing the aforementioned examples, we introduce the signal library – as viewed by a user – and derive some useful combinators from its core constructs.

The signal library is mostly based around the same concepts of FRP as, for example, Yampa is: Signals represents a mapping from time to values of some type and are based on the reactive approach, that is, signals are only run when their results are needed. All of FRP's concepts are however not directly present in the signal library. For example, the notion of events have been treated slightly differently and are based on ideas from SDF instead. Event sources are therefore not the continuous streams of, possibly empty, values as they are in Yampa. Instead, they make use of variable clocks to determine when an output is produced.

Much of the core functionality provided by the signal library comes from the functions we introduced during the creation of the `Signal` type, and have been given the following type declarations:

```
lift   :: ( Syntax a, Typeable a
          , Syntax b, Typeable b
          ) ⇒ (Stream a → Stream b)
            → Signal a
            → Signal b

zip    :: (Syntax a, Syntax b) ⇒ Signal a → Signal b → Signal (a, b)
fst    :: (Syntax a, Syntax b) ⇒ Signal (a, b) → Signal a

delay  :: (Syntax a, Typeable a) ⇒ a → Signal a → Signal a
sample :: (Syntax a, Typeable a) ⇒ Data Length → Signal a → Signal a
bot    :: (Syntax a, Typeable a) ⇒ Signal a
```

While these functions are no more than short-hand constructors for `Signal`, they are quite general and allow arbitrary streaming computations to be expressed by simply lifting them into signals; most of the signal library's functionality does actually come from simply lifting functions from Feldspar's stream library into signals. For example, the following functions are implemented almost entirely by lifting stream functions:

```
repeat :: (Syntax a, Typeable a) ⇒ a → Signal a
repeat a = lift (S.repeat a ‘asTypeOf‘) bot

iterate :: (Syntax a, Typeable a) ⇒ (a → a) → a → Signal a
iterate f a = lift (S.iterate f a ‘asTypeOf‘) bot

snd :: (Syntax a, Typeable a, Syntax b, Typeable b) ⇒ Signal (a, b) → Signal b
snd = fst . map (λ(a, b) → (b, a))

map :: (Syntax a, Typeable a, Syntax b, Typeable b) ⇒ (a → b) → Signal a → Signal b
map f = lift $ S.map f
```

```
scan :: (Syntax a, Typeable a, Syntax b, Typeable b)
       ⇒ (a → b → a) → a → Signal b → Signal a
scan f a = lift $ S.scan f a

zipWith :: ( Syntax a, Typeable a
           , Syntax b, Typeable b
           , Syntax c, Typeable c
           ) ⇒ (a → b → c)
             → Signal a
             → Signal b
             → Signal c
zipWith f = merge $ S.zipWith f

merge :: ( Syntax a, Typeable a
         , Syntax b, Typeable b
         , Syntax c, Typeable c
         ) ⇒ (Stream a → Stream b → Stream c)
           →  Signal a → Signal b → Signal c
merge f as bs = lift (uncurry f . S.unzip) $ zip as bs
```

The fact that it is possible to lift the entire stream library into signals is a testament to the benefits of having a lifting operator.

In fact, the usefulness of lifting is in no way limited to functions from Feldspar's stream library, but can be used in conjunction with other libraries as well. For instance, the fast Fourier transform (FFT) algorithm can be implemented as a signal transformer by simply lifting the existing FFT function from Feldspar's algorithm library:

```
import qualified Feldspar.Algorithm.FFT as F (fft)
import qualified Feldspar.Vector        as V

deriving instance Typeable1 V.Vector

fft :: Signal (V.Vector1 (Complex Float)) → Signal (V.Vector1 (Complex Float))
fft = lift $ S.map F.fft
```

The ability to reuse existing Feldspar functions in this way significantly reduces the complexity of developing new signal processing programs.

While the above functions are often enough to let users to define the DSP algorithms they want, a number of commonly used signal transformers are also included in the signal library. For example, an edge detector and a multiplexer are provided with the following implementations:

```
edge :: Signal (Data Bool) → Signal (Data Bool)
edge sig = zipWith (/=) sig (delay false sig)

mux :: (Syntax a, Typeable a) ⇒ Signal (Data Bool) → Signal a → Signal a → Signal a
mux = zipWith3 (λ b s1 s2 →  b ? (s1, s2))
```

The current implementation of the multiplexer does however suffer from the interference and efficiency problems discussed in chapter 3. Introducing a conditional construct, similar to the common if-statements, could potentially alleviate these problems, and the idea is discussed further in chapter 6.

5.2 An Example: IIR Filter

Infinite impulse response (IIR) filters are digital filters with an infinite impulse response and, unlike FIR filters, contain feedback. They are therefore known as recursive digital filters, as they contain a recursively defined parts. These filters will serve as an example of how the signal library handles recursively defined signals, that is, signals whose output depends on a combination of previous input and output values.

The IIR filter, or at least its digital version, is often described and implemented in terms of a difference equation, which defines how the output signal is related to the input signal:

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i * x_{n-j} - \sum_{j=1}^{Q} a_j * y_{n-j}\right)$$

where P and Q are the feedforward and feedback filter orders, respectively; a_j and b_i are the filter coefficients.

This description is convenient for software realisation, as it can easily be broken down into a couple of main components: a number of unit delays, multiplications with some coefficients, a summation of the two amplified signals, and a singel subtraction and division. We can represent the decomposed filter graphically, as in Figure 5.1.

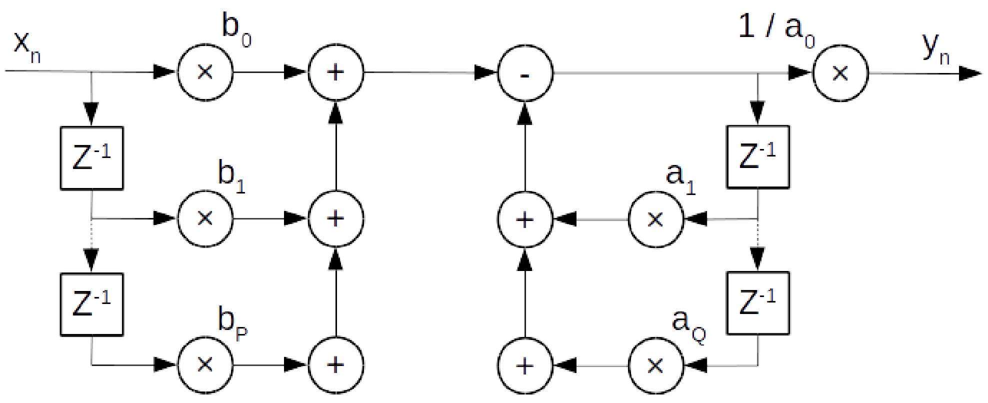

Figure 5.1: A direct form discrete-time IIR filter of order P and Q

Besides the subtraction and division, the deconstructed variants of the IIR and FIR filters are quite similar. This similarity seem to imply that the IIR filter could be expressed in a similar manner as the FIR filter was. As it turns out, the same helper functions used to implement the FIR filter can be used to implement the IIR filter as well; the one major difference between the two kinds of filters is the use of a recursively defined signal, as the rightmost sum is defined in terms of previous output values:

```
iir :: [Data Float] → [Data Float] → Signal (Data Float) → Signal (Data Float)
iir as@(a:_) bs sig = repeat (1 ‘divFrac‘ a) * (left - right)
  where
    left :: Signal (Data Float)
    left = sums $ muls bs $ delays (inits bs) sig

    right :: Signal (Data Float)
    right = sums $ muls as $ delays (inits as) right

    inits l = (P.replicate (P.length l) 0)
```

Due to the lazy nature of the delay operator, the recursive component of the filter can be expressed in the same way as one would do with an input signal.

5.3 An Example: Bouncing Balls

The task of modeling a bouncing ball is an example in how the different languages handle behavioural switches in a signal, as the ball's velocity changes direction each time it touches the ground. Since the signal library does not offer support for Lucid Synchrone's state machines or Yampa's family of switch operators, we are instead required to solve the problem in a different manner.

While it is certainly possible to implement a skeleton of the example, with similar functions as those used in the previous implementations of the problem, any such attempt would get stuck on the bouncing logic. For example, if we begin implementing the example as:

```
type Pos = Data Float
type Vel = Data Float
type Ev  = Data Bool

falling_ball :: (Pos, Vel) → Signal (Pos, Vel)
falling_ball (initY, initV) = zip currV currY
  where
    currV = iterate (9 ‘addNum‘) initV
    currY = scan (λ acc v → acc ‘subNum’‘ v) initY currV
      where subNum’ a b = (a ≤ b) ? (0, a ‘subNum‘ b)

falling_ball_ev :: (Pos, Vel) → Signal ((Pos, Vel), Ev)
falling_ball_ev init = zip currP currE
  where
    currP = falling_ball init
    currE = edge $ map ((≤0) . snd) currP

bouncing_ball :: Signal (Pos, Vel)
bouncing_ball = ...
```

it would be difficult to fill in the `bouncing_ball` function by using the current signal library's functions. The signal library does offer a multiplexer for these kinds of situations, which one can use to control a signal's behaviour. However, we cannot control the signals behaviour from the `bouncing_ball` function, since the multiplexer doesn't restart a stream like Yampa's switch operators does. The problem therefore needs to be solved in a different way.

A normal course of action in these circumstances is to replace the logic with arithmetic expressions instead, where a function is created in order to mimic the bouncing behaviour. In the case of the bouncing balls example, a simple step function is sufficient:

```
bouncing_ball :: Signal (Pos, Vel)
bouncing_ball = iterate fall (100, 0)
  where
    fall :: (Pos, Vel) → (Pos, Vel)
    fall (p,v) = (p',v')
      where v' = (p == 0)  ? (negate v, v ‘addNum‘ 9)
            p' = (p ≤ v') ? (0, p ‘subNum‘ v')
```

checking at each step whether the ball has reached the floor, negating its velocity whenever it does.

5.4 An Example: Power

The power function required the use of sampling, as the output stream produces events at a slower pace then its input. Simply using the signal library's `sample` operator to sample the input stream, as was done in the Yampa implementation, will not have the desired effect: the internal computations, which depend on the sampled signal, will then also run at a slow pace. To remedy this, a stuttering function could be introduced. The function would then be used in order to increase the clock rate of the internal computations by repeating the sampled streams value. This would however require additional compilation checks to assert that the clock speed never goes above the global limit.

A simpler solution to the sampling problem is to instead solve the problem in a similar way as Lucid Synchrone did, using one of its merging operators. The signal library does not offer any such merging operator, but it does have a multiplexer and a lifting operator, with which one can create any such merging operator with. For instance, Figure 5.2 illustrates the kind of merging function needed in order to express the power function. Where the multiplexer is used to sample the input stream and the counting is done modulo the power rank.

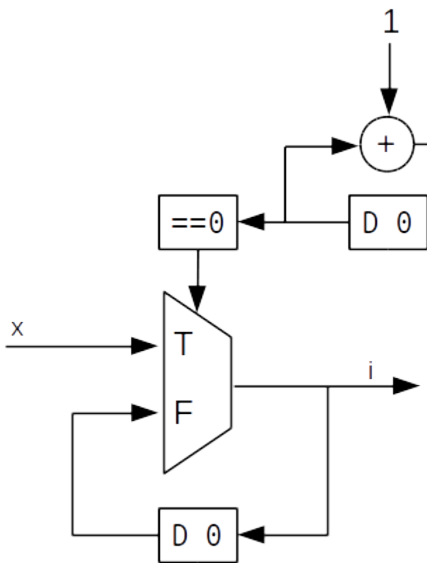

Figure 5.2: Circuit representation of a merging function

Given this circuit representation of the merging function, an implementation of the power function can be realised in a straightforward manner. First, we introduce a couple of helper functions in order to model the counter, and give them the following declarations:

```
count :: Data WordN → Data WordN → Signal (Data WordN)
count start max = o
  where o = delay start $ map (λc → c + 1 ‘mod‘ max) o

timer :: Data WordN → Data WordN → Signal (Data Bool)
timer start max = map (==0) $ count start max
```

We can then express the power function as:

```
power :: Data Length → Signal (Data Float) → Signal (Data Float)
power pow sig = y
  where
    i :: Signal (Data Float)
    i = mux (0 ‘timer‘ pow) sig (delay 0 i)

    o :: Signal (Data Float)
    o = delay 1 $ (i ∗) $ mux (0 ‘timer‘ pow) i o

    y :: Signal (Data Float)
    y = sample pow o
```

which is quite similar to its corresponding version in Lucid Synchrone. The general concept is the same as well: a stream, `i`, holds the current value while another stream, `o`, performs the incremental multiplications. The output stream is then created by simply sampling the internal streams at the correct intervals.

5.5 Related Work

In this section we present some of the related work which was left out from the earlier background chapter but still served as inspiration for our design of signals. These languages will only be briefly explained, and we instead focus on comparing any similarities they may share with our own signals or presenting their interesting language features.

5.5.1 Chalmers Lava

Lava is a family of Haskell EDSLs designed for expressing hardware descriptions [42, 46], and is commonly a design pattern for constructing EDSLs that try to capture some common concerns of hardware design. The general idea in Lava languages is that one can describe circuits as Haskell functions, which gives us a nice way of expressing how larger circuits can be composed from a number of sub-components.

Chalmers Lava [47] is an experimental tool designed to assist circuit designers with hardware design, specification and verification. The language itself is embedded in Haskell and allows hardware circuits to be described by simply writing ordinary Haskell functions.The general idea is then that these descriptions can be analysed in a number of different ways. To illustrate this, consider the following description of a half and a full adder:

```
import Lava

type Bit = Signal Bool

half_add :: (Bit, Bit) → (Bit, Bit)
half_add (a, b) = (s, co)
  where
    s  = xor2 (a, b)
    co = and2 (a, b)

full_add :: (Bit, Bit, Bit) → (Bit, Bit)
full_add (ci, a, b) = (s, co)
  where
    (s1, c1) = half_add (a,  b)
    (s,  c2) = half_add (ci, s1)
    co       = xor2 (c1, c2)
```

where the `Signal` type represents an infinite stream of symbols. Input wires are grouped together into tuples by convention, so that circuits always have a single output and input value. Given suitable input data, we can simulate the full adder:

```
simulate full_add (high, low, high)

> (high, low)
```

The approach used in our definition of `Signal` for embedding data flow using observable sharing is very similar to that of Chalmers Lava. `Signal` is however more general when considering the kinds of operators one can express in the two languages: Chalmers Lava does not permit lifting of arbitrary streaming computations into the language; functions are instead constricted to use a set of primitive gates. Observable sharing is also handled differently by Chalmers Lava, as they introduce their own references and use the following interface to manually handle references:

```
type Ref a = Ref (IORef [(IORef (), Dyn)]) a

instance Eq (Ref a) where
    Ref r1 _ == Ref r2 _ = r1 == r2

ref   :: a → Ref a
deref :: Ref a → a
```

Each signal is then wrapped in these references and we can create new references, compare them for equality and de-reference them. For instance, in the below example we create one reference to an undefined value and compare it with itself, which yields true.

```
let x = "Hello Reader"
    r = ref x
 in r == r

> True
```

Sequential operations are also expressed differently using `Signal`, since Lava relies on circuit transformations to slow down signals instead of providing a specific sampling function. Furthermore, `Signal`'s make use of some compilation tricks in order to optimise a program's graph – which Lava does not. This is mainly due to that fact that Lava targets gate-level hardware, where one can afford more computations because they are all done in parallel.

5.5.2 Kansas Lava

Kansas Lava [46] is another member in the Lava family of hardware description languages. It is designed to express hardware-oriented descriptions of computations, from which it can generate VHDL code. The language itself is an EDSL hosted in Haskell, and its programs are simple descriptions of hardware components – as is standard in Lava languages.

A `Signal` in Kansas Lava represent an infinite sequence of values over time, and encodes the typical representation of a signal in VHDL: a vector of wires. It is semantically modeled as

```
Signal a :: Time → a
```

where `Time` is a clock cycle count – note the similarity to the semantics of signals in FRP. One noteworthy aspect of `Signal` in Kansas Lava, which differs from how circuits are usually built using gate-level operations in Lava, is the way its `Signal` type embeds both the synthesizable circuit and its model, that is, Kansas Lava's `Signal` serves as a representation for both the shallow and deep embeddings of circuits. It is realised in the form of a pair, operating in the clock domain `clk`:

```
data Signal (clk :: *) a = Signal (Stream (X a)) (D a)
```

Where a phantom type [48] is used to express the clock domain; phantom types were considered during the implementation of `Signal` for Feldspar, but the idea was abandoned in order to avoid type-level programming. The clocks are however global in the current version of Kansas Lava[1], restricting their use to single-clock systems.

The deep embedding of `Signal`, named `D`, is a typed entity which usually corresponds to some real entity in VHDL. An Entity is a globally scoped name representing a specific logical function, some type information about the entity, and a set of input ports and their respective drivers; a driver is a representation of a wire, originating from a input pad, some constant, or another entity. The shallow portion of a `Signal` is represented as a `Stream`, operating over a sequence of unknown values; conceptually, these unknown values, `X`, work in a similar way as Haskell's `Maybe` does. `Stream` is given the following implementation in Kansas Lava:

```
data Stream a = Cons !a (Maybe (Stream a))
```

and is defined as a infinite sequence of values, since if the tail is empty, then the last value is repeated. Elements of the stream are made strict, in order to prevent space leaks.

While programming in Kansas Lava means a shallow and deep embedding of the circuit is built in parallel, the actual construction is hidden by its operators. Its syntax is therefore quite similar to other Lava languages, for instance, consider the half and full adders we previously defined in Chalmers Lava. These can be implemented in Kansas Lava as well by writing:

```
half_add :: (Clock clk, sig ~ Signal clk Bool) ⇒ (sig, sig) → (sig, sig)
half_add (a, b) = (s, co)
  where
    s  = a ‘xor‘ b
    co = a  .&.  b

full_add :: (Clock clk, sig ~ Signal clk Bool) ⇒ (sig, sig, sig) → (sig, sig)
full_add (ci, a, b) = (s, co)
  where
    (s1, c1) = half_add (a,  b)
    (s,  c2) = half_add (ci, s1)
    co       = c1 ‘xor‘ c2
```

which is quite similar to their corresponding Chalmers Lava version. The major differences between the two Lava implementations of these adders are their type signatures and the two, slightly differently named, logical operators.

[1]The version of Kansas Lava that was used during this report was the one available at Github [46] in June - 2014

The two languages do however lose their close resemblance when considering their internal representations of circuits. For example, Kansas Lava makses use of type-safe observable sharing – as our own signal library does – to support feedback networks, rather than explicit references such as those used in Chalmers Lava.

5.5.3 CAL

The Cal Actor Language [49] is a high-level programming language for defining data-flow actors, where actors are stateful operators which transform their input and forwards it to the output stream – similar to the nodes of a DFP graph. Actors perform their transformation in steps, as they first may consume some values from their input streams, modify their internal states, and lastly, produce values at their output streams.

Defining actors is therefore done by describing their interface to the outside world, their input and output streams, the structure of any internal state they may use, and the transformation they perform on their inputs. For example, a two input adder can be defined as:

```
actor Add[T] () T in1, T in2 ⟹ T out1
:
  action in1:[a], in2:[b] ⟹ out1:[a + b] end
end
```

The actor line at the top defines the port signature of the `Add` actor with a generic type parameter `T` and no value parameters. There are two input ports, `in1` and `in2`, each of type `T`, and an output port, `out1`, also of type `T`. Each action construct defines a, possibly partial, pattern of input values and a consequent response at some outputs whenever the input pattern appears. More complicated actors may involve state functions or pattern guards on their actions, it is also possible to have different actions consume or produce different amounts of tokens in the same actor. For instance, the following definition is a valid actor in CAL:

```
actor Sum () in =⇒ out
:
  sum := 0;

  action in:[a, b] =⇒ out:[sum] guard a ≥ sum
  do
    sum := sum + a + b;
  end

  action in:[a] =⇒ out:[sum, a] guard a ≤ sum
  do
    sum := sum - a;
  end
end
```

where actions work at different rates and a local state, called `sum`, is shared between the two. There is also non-determinism in this actor since both actions could fire at the same time if two input tokens are available and both guards are true – and the output will look different depending on which action fires. CAL does not offer a definitive rule for how such non-determinism should be handled; it is instead up to the actor's author to implement a scheduling for the activities in the actor network.

5.5.4 Matlab

Matlab [50] is an interactive and matrix-based tool for scientific and engineering number computations, analysis and visualisation. Its strength lies in the fact that it can express rather complex numerical problems relatively easily and then visualize the results, all while using only a fraction of the effort required with a more complicated programmning language like C. It is also powerful in the sense that it may behave as both a calculator and a programming language, which means that it can be easily extended with new functions.

Digital signal processing in Matlab is supported by its 'DSP Systems Toolbox' package [51], which provides algorithms for designing and simulating signal processing systems. The toolbox also includes various design methods for specialised FIR and IIR filters, FFTs, and other DSP techniques for processing streaming data. Tools for spectral analysis and visualisation of signals are provided as well, enabling users to analyse a systems behaviour and performance.

Filter design in Matlab can be conceptually simplified to a process consisting of four steps, where an optional step can be added in order to specialise the filter further with implementation details. The first four steps of the filter's design process all relate to its specification: a filter response is selected, this is needed to initiate the filter and allows one to specify its type to, for example, a bandpass filter; a filter specification is chosen, by setting a number of design parameters for the filter; an algorithm for the filter is set, these algorithms are chosen from a predefined set and affect the filter's implementation details; lastly there is an optional step for filter customisation. If any of the last two steps are skipped, then Matlab is kind enough to select to optimal settings for the given filter type.

The following example shows how to design low pass FIR filters in Matlab, using only a few lines of code.

```
Fc = 0.4;   % filter cutoff frequency
N  = 100;   % FIR filter order

% step 1: selecting a filter response
d = fdesign.bandpass

% step 2: set the specification paramaters
set (d, 'specification', 'N,Fc', N, Fc)

% step 3: selecting the filter algorithms
f1 = design(d, 'window', 'window', @hamming,      'SystemObject', true);
f2 = design(d, 'window', 'window', {@chebwin,50}, 'SystemObject', true);
```

The two filters, `f1` and `f2`, are created using the Dolph-Chebyshev and Hamming window functions, respectively; the fourth step was omitted in order to have Matlab optimise the settings for us. We can then analyse the filters using the tools provided. For instance, we can visualise the two window functions by writing

```
fp = fvtool(f1, f2, 'Color', 'White');
legend(fp, 'Hamming window design', 'Dolph-Chebyshev window design')
```

which results in Figure 5.3.

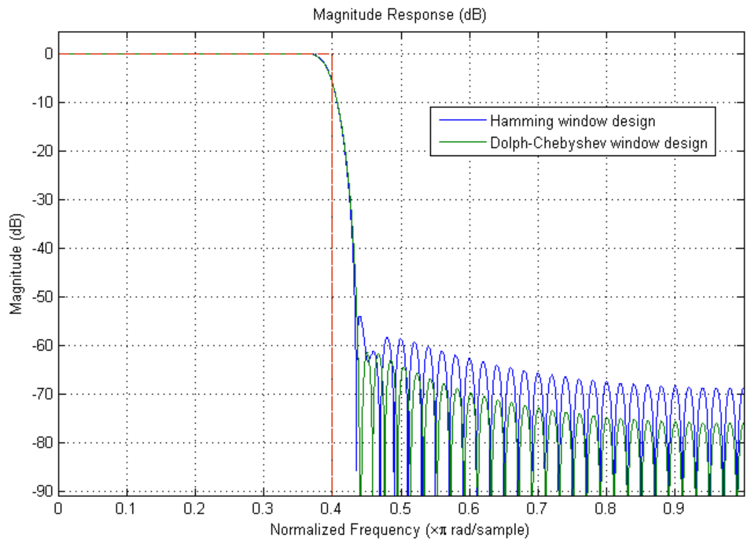

Figure 5.3: Analysis of two FIR filers by graph visualisation

Filter design in Matlab is, simply stated, a process of selecting a pre-existing filter and tweaking its parameters. This means that relatively specialised filters can be defined quickly and in a straightforward manner. It does however lack the expressive power of the more complicate programming languages, the produced filters are also slower in most cases since Matlab is an interpreted language.

6

Conclusions and Future Work

Due to the time constraints, limitations on the scope and depth of this project were required and meant that the final implementation is closer to a proof of concept than a fully-fledged streaming library. Nonetheless, we managed to transform Feldspar's current stream library into a true DSL for signal processing, while still retaining its old efficiency through optimisations of the network graphs.

The currently employed optimisations are however quite far from utilising the full potential of our pre-processing stage, as we only optimise delayed signals. For instance, it is entirely feasible to implement a reordering phase, similar to the one used for causal commutative arrows (CCA) [52], in order to further optimise a network's graph. Several constraints are placed on the effects of arrows in the CCA paper, they do so in order to guarantee that any two arrows will not interfere with each other if both are executed. Given these constrained arrows, it is possible to reorder combinatorial and sequential parts of an arrow in a beneficial way.

In our `Signal` class we already have different representations combinatorial and sequential operators, represented using the `Arr` or `Delay` and `Sample` constructors, respectively; reordering these nodes could potentially open up for fusing more of the combinatorial nodes. Missing is the restriction on effects, as both signals in a pair may refer to some common sub-signal, we cannot therefore execute one and expect the other to be unchanged. This could however be solved in part by inspecting the two signals' graphs, detecting any common sub-signal used by both; replacing those shared nodes with, for example, dummy arguments or local copies could potentially solve the problem.

Even though the automatons from Lucid Synchrone require a special kind of syntax extension, they do present an efficient way of modeling control structures for signals. In order to port these automatons to our signals we would either require special syntax extension, as introduced by the Sugar Haskell library [53], or implement support for conditionals; since we usually prefer to use as pure Haskell code as possible, the second alternative is the preferred one. Using conditionals, we could model the flow control from automatons, switching between states, and the various triggers they make use of. The states can already be represented using recursively defined signals; the introduction of some helper functions, for implicitly passing around states, is however necessary to minimise the overhead.

Another feature currently missing from signal is the ability to use variable clocks. Sampling is currently only possible at regular intervals and should ideally be extended to allow for sampling according to some variable rate, where the rate is specified by some stream of booleans and sampling occurs each time the stream returns true. While variable clocks might not be too common of a requirement, it is still a limitation of the current signal type which we would like to remove. In order to support variable clocks we have to move closer to Lucid Synchrone's concept of sampling according to the `when` and `whennot` functions. These would force us to do an analysis of the networks in order to determine clock frequencies.

Even though the library may have some future work, we feel that it manages to capture the ideas of signal processing in a nice way: FPR's notion of signal behaviour and transformations form the basis of the library, while temporal operators from SDF are used to model sub-signals running at different speeds. We feel that this mixture of ideas have lead to an intuitive model for signal processing. Furthermore, the ability to use pure Haskell as a meta-programming language for the signal library makes rather complex programs easy and intuitive to implement. The fact that our library's functions are semantically similar to Haskell's standard operations on lists further helps users to reason about streams, since they can simply treat them as infinite lists. The lack of proper support for automatons is however regrettable, as they provide a nice and efficent way of describing control dominated systems.

Bibliography

[1] Cisco, Cisco visual networking index: Forecast and methodology, 2012–2017 (May 2012).
URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html

[2] ITU, Global itc development, 2001-2014 (2014).
URL http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2014/stat_page_all_charts_2014.xls

[3] A. Persson, Towards a functional programming language for baseband signal processing.

[4] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, A. Vajda, Feldspar: A domain specific language for digital signal processing algorithms, in: Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on, 2010, pp. 169–178.

[5] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, A. Persson, The design and implementation of feldspar, in: J. Hage, M. Morazán (Eds.), Implementation and Application of Functional Languages, Vol. 6647 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 121–136.
URL http://dx.doi.org/10.1007/978-3-642-24276-2_8

[6] S. L. P. Jones, Haskell 98 language and libraries: the revised report, Cambridge University Press, 2003.

[7] E. Axelsson, A. Persson, J. Svenningsson, Feldspar's streams (2014).
URL https://github.com/Feldspar/feldspar-language/blob/master/src/Feldspar/Stream.hs

[8] P. Caspi, M. Pouzet, A co-iterative characterization of synchronous stream functions (1997).

[9] B. Mulgrew, P. M. Grant, J. Thompson, Digital signal processing: concepts and applications, Macmillan Press, 1999.

[10] A. Oppenheim, Applications of Digital Signal Processing, Prentice-Hall Signal Processing Series, Prentice-Hall, 1978.

[11] E. Lee, D. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.

[12] Z. Wan, P. Hudak, Functional reactive programming from first principles, SIGPLAN Not. 35 (5) (2000) 242–252.
URL http://doi.acm.org/10.1145/358438.349331

[13] C. Elliott, P. Hudak, Functional reactive animation, SIGPLAN Not. 32 (8) (1997) 263–273.
URL http://doi.acm.org/10.1145/258949.258973

[14] H. Nilsson, A. Courtney, J. Peterson, Functional reactive programming, continued, in: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02, ACM, New York, NY, USA, 2002, pp. 51–64.
URL http://doi.acm.org/10.1145/581690.581695

[15] A. Courtney, C. Elliott, Genuinely functional user interfaces, in: Haskell Workshop, 2001, pp. 41–69.

[16] E. Czaplicki, Elm (2011).
URL https://github.com/elm-lang/elm-lang.org

[17] Z. Wan, W. Taha, P. Hudak, Real-time frp, SIGPLAN Not. 36 (10) (2001) 146–156.
URL http://doi.acm.org/10.1145/507546.507654

[18] A. Courtney, H. Nilsson, J. Peterson, The yampa arcade, in: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell, Haskell '03, ACM, New York, NY, USA, 2003, pp. 7–18.
URL http://doi.acm.org/10.1145/871895.871897

[19] G. Giorgidze, H. Nilsson, Switched-on yampa, in: P. Hudak, D. Warren (Eds.), Practical Aspects of Declarative Languages, Vol. 4902 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 282–298.
URL http://dx.doi.org/10.1007/978-3-540-77442-6_19

[20] J. Hughes, Generalising monads to arrows, Science of Computer Programming 37 (1–3) (2000) 67 – 111.
URL http://www.sciencedirect.com/science/article/pii/S0167642399000234

[21] P. Hudak, A. Courtney, H. Nilsson, J. Peterson, Arrows, robots, and functional reactive programming, in: J. Jeuring, S. Jones (Eds.), Advanced Functional Programming, Vol. 2638 of Lecture Notes in Computer Science, Springer Berlin Heidelberg,

2003, pp. 159–187.
URL http://dx.doi.org/10.1007/978-3-540-44833-4_6

[22] W. M. Johnston, J. R. P. Hanna, R. J. Millar, Advances in dataflow programming languages, ACM Comput. Surv. 36 (1) (2004) 1–34.
URL http://doi.acm.org/10.1145/1013208.1013209

[23] T. B. Sousa, Dataflow programming concept, languages and applications, in: Doctoral Symposium on Informatics Engineering, 2012.

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

[25] E. Lee, D. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, Computers, IEEE Transactions on C-36 (1) (1987) 24–35.

[26] P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice, Lustre: A declarative language for real-time programming, in: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '87, ACM, New York, NY, USA, 1987, pp. 178–188.
URL http://doi.acm.org/10.1145/41625.41641

[27] A. Gamatié, Designing embedded systems with the SIGNAL programming language, Springer, 2010.

[28] M. Pouzet, Lucid synchrone, version 3, Tutorial and reference manual. Université Paris-Sud, LRI.

[29] J.-L. Colaço, A. Girault, G. Hamon, M. Pouzet, Towards a higher-order synchronous data-flow language, in: Proceedings of the 4th ACM International Conference on Embedded Software, EMSOFT '04, ACM, New York, NY, USA, 2004, pp. 230–239.
URL http://doi.acm.org/10.1145/1017753.1017792

[30] G. Berry, G. Gonthier, The esterel synchronous programming language: design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87 – 152.
URL http://www.sciencedirect.com/science/article/pii/016764239290005V

[31] N. Halbwachs, Synchronous programming of reactive systems, no. 215, Springer, 1992.

[32] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, Signal–a data flow-oriented language for signal processing, Acoustics, Speech and Signal Processing, IEEE Transactions on 34 (2) (1986) 362–374.

[33] Functional programming group at Chalmers University of Technology, Feldspar-language (May 2014).

[34] P. Caspi, M. Pouzet, A co-iterative characterization of synchronous stream functions, Electronic Notes in Theoretical Computer Science 11 (0) (1998) 1 – 21, {CMCS} '98, First Workshop on Coalgebraic Methods in Computer Science.
URL http://www.sciencedirect.com/science/article/pii/S1571066104000507

[35] A. Persson, E. Axelsson, J. Svenningsson, Generic monadic constructs for embedded languages, in: A. Gill, J. Hage (Eds.), Implementation and Application of Functional Languages, Vol. 7257 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 85–99.
URL http://dx.doi.org/10.1007/978-3-642-34407-7_6

[36] G. H. Mealy, A method for synthesizing sequential circuits, Bell System Technical Journal 34 (5) (1955) 1045–1079.
URL http://dx.doi.org/10.1002/j.1538-7305.1955.tb03788.x

[37] J.-L. Colaço, M. Pouzet, Clocks as first class abstract types, in: R. Alur, I. Lee (Eds.), Embedded Software, Vol. 2855 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 134–155.
URL http://dx.doi.org/10.1007/978-3-540-45212-6_10

[38] E. Axelsson, A. Persson, M. Sheeran, J. Svenningsson, G. Deval, A tutorial on programming in feldspar (2011).

[39] R. Atkey, What is a categorical model of arrows?, Electronic Notes in Theoretical Computer Science 229 (5) (2011) 19 – 37, proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).
URL http://www.sciencedirect.com/science/article/pii/S157106611100051X

[40] A. Megacz, Multi-level languages are generalized arrows, CoRR abs/1007.2885.

[41] J. O'Donnell, Generating netlists from executable circuit specifications in a pure functional language, in: J. Launchbury, P. Sansom (Eds.), Functional Programming, Glasgow 1992, Workshops in Computing, Springer London, 1993, pp. 178–194.
URL http://dx.doi.org/10.1007/978-1-4471-3215-8_16

[42] P. Bjesse, K. Claessen, M. Sheeran, S. Singh, Lava: Hardware design in haskell, SIGPLAN Not. 34 (1) (1998) 174–184.
URL http://doi.acm.org/10.1145/291251.289440

[43] A. Gill, Type-safe observable sharing in haskell, in: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell, Haskell '09, ACM, New York, NY, USA, 2009, pp. 117–128.
URL http://doi.acm.org/10.1145/1596638.1596653

[44] C. Elliott, S. Finne, O. De Moor, Compiling embedded languages, J. Funct. Program. 13 (3) (2003) 455–481.
URL http://dx.doi.org/10.1017/S0956796802004574

[45] S. Kamin, et al., Standard ml as a meta-programming language, Tech. rep., Technical report, University of Illinois at Urbana-Champaign (1996).

[46] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, B. Werling, Introducing kansas lava, in: M. Morazán, S.-B. Scholz (Eds.), Implementation and Application of Functional Languages, Vol. 6041 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 18–35.
URL http://dx.doi.org/10.1007/978-3-642-16478-1_2

[47] K. Claessen, Embedded languages for describing and verifying hardware, Ph.D. thesis, Chalmers University of Technology (2001).

[48] M. Rhiger, A foundation for embedded languages, ACM Trans. Program. Lang. Syst. 25 (3) (2003) 291–315.
URL http://doi.acm.org/10.1145/641909.641910

[49] J. Eker, J. Janneck, Cal language report, University of California at Berkeley, Tech. Rep. UCB/ERL M 3.

[50] MATLAB, version 7.10.0 (R2010a), The MathWorks Inc., Natick, Massachusetts, 2010.

[51] J. N. Little, L. Shure, Signal processing toolbox: for use with MATLAB; user's guide, Math Works, 1992.

[52] H. LIU, E. CHENG, P. HUDAK, Causal commutative arrows, Journal of Functional Programming 21 (2011) 467–496.
URL http://journals.cambridge.org/article_S0956796811000153

[53] S. Erdweg, F. Rieger, T. Rendel, K. Ostermann, Layout-sensitive language extensibility with sugarhaskell, in: Proceedings of the 2012 Haskell Symposium, Haskell '12, ACM, New York, NY, USA, 2012, pp. 149–160.
URL http://doi.acm.org/10.1145/2364506.2364526

[54] H. Thielemann, Audio processing using Haskell, Zentrum für Technomathematik, 2004.