Stream Processing for Embedded Domain Specific Languages

Markus Aronsson Emil Axelsson Mary Sheeran
Chalmers University of Technology
mararon@chalmers.se, emax@chalmers.se, ms@chalmers.se

Abstract

We present a library for expressing digital signal processing (DSP) algorithms using a deeply embedded domain-specific language (EDSL) in Haskell. The library supports definitions in functional programming style, reducing the gap between the mathematical description of streaming algorithms and their implementation. The deep embedding makes it possible to generate efficient C code. The signal processing library is intended to be an extension of the Feldspar EDSL which, until now, has had a rather low-level interface for dealing with synchronous streams. However, the presented library is independent of the underlying expression language, and can be used to extend any pure EDSL for which a C code generator exists with efficient stream processing capabilities. The library is evaluated using example implementations of common DSP algorithms and the generated code is compared to its handwritten counterpart.

Categories and Subject Descriptors D.3.2 [*Language Classification*]: Applicative (functional) languages—Dataflow languages; C.3 [*Computer Systems Organization*]: Signal processing systems

General Terms Languages, Design

Keywords Digital Signal Processing, Stream processing, Embedded domain specific language, dataflow, synchronous programming, observable sharing, code generation

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 2015 [20]. For telecommunications infrastructure, the consequence of such a rapid growth rate has been a dramatic increase in the demand for network capacity and computational power [1].

IFL '14, October 1–3, 2014, Boston, MA, USA.

http://dx.doi.org/10.1145/978-1-4503-3284-2

Today, digital signal processing software is typically implemented in low level C, which forces designers to focus on low-level implementation details rather than the mathematical specification of the algorithms. Our group is developing an embedded domain-specific language, Feldspar [5], that aims to raise the abstraction level of signal processing software by expressing algorithms as pure functional programs.

However, signal processing is more than just pure computations – it is also about how to connect those functions in a network that operates on streaming data. A suitable programming model for reactive systems that process streams of data is *synchronous dataflow* (SDF) [21], which offers natural, high-level descriptions of streaming algorithms, while still permitting the generation of efficient code. There already exist a number of SDF languages, such as Lustre [10], but we are interested in adding an SDF programming model to Feldspar rather than using an existing language. The reason is that Feldspar is an ecosystem of different programming models which together offer much more flexibility than a typical SDF language.

Feldspar does have a library for programming with synchronous streams, but they are implemented in such a way that we cannot analyze them or detect sharing – a common problem in embedded languages. As a result of this, we cannot prevent code duplication and instead need to resort to using low-level combinators, which are tedious to use and do not scale well. Our solution is not to abandon such streams altogether – instead, we wrap them in a layer that remedies these problems.

This paper describes a library for extending an existing Haskell EDSL, like Feldspar, with support for SDF. The underlying EDSL is used to represent pure functions (which, of course, can be arbitrarily complicated), and our library gives a means to connect such functions using an SDF programming model. If the underlying EDSL provides a C code generator with a given interface, our library is capable of emitting C code for SDF programs. While we are interested in using Feldspar as the expression language, the library is not dependent on Feldspar, and so may be of interest to other EDSL developers.

This paper makes the following contributions:

- We present an EDSL for synchronous dataflow programming in Haskell (Section 2). By allowing the dataflow nodes to run arbitrary code, our EDSL greatly increased the expressive power compared to previous dataflow EDSLs in Haskell, such as Lava [11, 16]. Practically, the result is a useful addition to Feldspar.
- We show how to abstract away from the underlying expression language by establishing an interface for the un-

derlying expression compiler and interpreter (Sections 3 and 4.6). As a result, we ensure that our EDSL and the expression language are loosely coupled, giving a clear separation of concerns and an interchangeable expression language.

- We show a method for code generation from our EDSL via two lower-level representations: streams and imperative programs (Section 4). Our representation of streams breaks away from the classical one [9], as streams instead take on a monadic form using imperative programs. This allows us to generate efficient code from streams.

Section 5 gives an evaluation of the method on some simple examples.

2. Signals

Our library[1] is based on the concept of *signals*: possibly infinite sequences of values in some pure expression language. The notion of *time* used is discrete and non-negative, as we require signals to be causal: output may only depend on current or previous input. If the output of a signal, or signal function, does not depend on previous input, it is called *combinatorial*. Otherwise it is *sequential*.

Conceptually, signals can be thought of as infinite lists, similar to those found in most lazy functional languages:

```
Sig a ≈ [a]
```

Unlike lists, `Sig` is, however, not a first-class value, as nesting of signals is disallowed.

Programming of signals is done compositionally; a rich set of operators supports the composition of new signals from existing ones. Signal programs are a collection of mutually recursive signal functions, each built from static values or other signals. For instance, some of the supplied combinatorial functions include:[2]

```
repeat :: Typeable a ⇒ Expr a → Sig a

map :: (Typeable a, Typeable b)
    ⇒ (Expr a → Expr b)
    → Sig a → Sig b

zipWith
    :: (Typeable a, Typeable b, Typeable c)
    ⇒ (Expr a → Expr b → Expr c)
    → Sig a → Sig b → Sig c
```

where `repeat` constructs a signal by repeating some value, `map` promotes a function to operate element-wise over signals, and `zipWith` joins two signals. These functions represent different kinds of nodes in the signal graph.

A fixed expression language, `Expr`, has been used in the above interface in order to improve readability. Later on, in section 4.6, this interface will be generalized to arbitrary expression types. Also, as a result of using type casting internally, values are required to be `Typeable`. Such a constraint is not harmful in practice, as most types satisfy the constraint.

We make use of existing type classes where possible, as they provide a familiar user interface when composing signals. For instance, numerical operations can be supported for signals by instantiating Haskell's `Num` class:

```
instance (Num a, Typeable a) ⇒ Num (Sig a)
  where
    fromInteger = repeat . fromInteger
    (+)         = zipWith (+)
    (-)         = zipWith (-)
    ...
```

The general idea is that every ground type, as permitted by the expression language, is lifted to operate element-wise over signals. This can either be done by redefining standard Haskell functions, as with the previous `repeat` and `map` functions, or through type classes.

For classes that cannot be instantiated in this way, we instead provide custom classes that override the default ones. For instance, as the `Eq` class has methods with non-overloaded types, making it incompatible with `Sig`, a customized version of that class is provided:

```
class EEq a
  where
    (==:) :: Expr a → Expr a → Expr Bool
    (/=:) :: Expr a → Expr a → Expr Bool
```

Using standard classes and function names in this way simplifies the construction of signals by providing a homogeneous user interface; complex networks can be defined using standard Haskell functions.

So far, only combinatorial functions have been considered. Most interesting examples of signal programs do, however, carry some form of state. For this reason, a sequential operator is provided:

```
delay :: Typeable a
      ⇒ Expr a → Sig a → Sig a
```

`delay` prepends a value to a signal, effectively delaying the input signal by one instant. Note that `delay` introduces the notion of a *next time step*, making time enumerable.

While it may appear innocent, the combination of `delay` with feedback allows one to express any kind of sequential network. For example, an edge detector could be implemented in terms of the delay function, checking at each instant if the values have changed:

```
edge :: Sig Bool → Sig Bool
edge s = zipWith (/=) s (delay false s)
```

where `false` creates a boolean value in the expression language.

An edge detector is however quite a small example. Therefore, in order to develop a better understanding of how programming with the signal library works, we will consider some larger examples, including the use of feedback.

2.1 FIR Filter

Finite impulse response (FIR) filters are one of the two primary types of digital filters used in DSP applications [23].

Consider the mathematical definition of a FIR filter of rank N:

$$y_n = \sum_{i=0}^{N} b_i \cdot x_{n-i} \tag{1}$$

While this description is already convenient for software realization, representing the decomposition graphically can yield a better intuition of what constitutes the filter.

The resulting filter is depicted in Figure 1, where the z^{-1} block is a unit delay. Looking at this representation, where

[1] Our library is available as open source [3].

[2] Because we are reusing names from Haskell's `Prelude`, these names must be hidden when using our library. This can be done by beginning each program with `import qualified Prelude`.

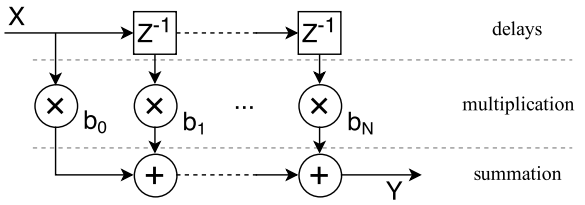

Figure 1. A direct form discrete-time FIR filter of order N

each column represents a term in the expanded summation, it is clear that the filter consists of three components: a number of successive unit delays, multiplication with coefficients and a summation.

Numerical operations are already supported for signals, since they instantiate Haskell's `Num` class. Consequently, in order to implement a FIR filter for signals, we only require a couple of helper functions to model each of its main components. These components are simply repeated applications of existing signal functions, and can therefore be modeled using pure Haskell functions:

```
import qualified Prelude as P

sums :: (Num a, Typeable a) ⇒ [Sig a] → Sig a
sums = P.foldr1 (+)

muls :: (Num a, Typeable a)
     ⇒ [Expr a] → [Sig a] → [Sig a]
muls as = P.zipWith (*) (P.map repeat as)

dels :: Typeable a ⇒ Expr a → Sig a → [Sig a]
dels a = P.iterate (delay a)
```

Here, addition and multiplication are overloaded operators from the `Num` class, and functions prefixed with `P` are standard library functions in Haskell.

Given these functions, a FIR filter can be expressed as:

```
fir :: (Num a, Typeable a)
    ⇒ [Expr a] → Sig a → Sig a
fir as = sums . muls as . dels 0
```

which is quite close to the filter's graphical representation. Domain experts in digital signal processing tend to be comfortable with composing sub-components in this way.

A FIR filter is still a small example, and, more importantly, it lacks feedback, a necessary component in many signal processing applications. We will therefore present one more example, which includes the use of feedback.

2.2 IIR Filter

Infinite impulse response (IIR) filters comprise the second primary type of digital filters used in DSP, and, unlike FIR filters, they contain feedback.

An IIR filter is typically described and implemented in terms of a difference equation, which defines how the output signal is related to the input signal:

$$y_n = \frac{1}{a_0} \cdot \left(\sum_{i=0}^{P} b_i \cdot x_{n-i} - \sum_{j=1}^{Q} a_j \cdot y_{n-j} \right) \qquad (2)$$

P and Q are the feedforward and feedback filter orders, respectively, and a_j and b_i are the filter coefficients. Nevertheless, representing the filter graphically can once again yield a better understanding of what constitutes the filter.

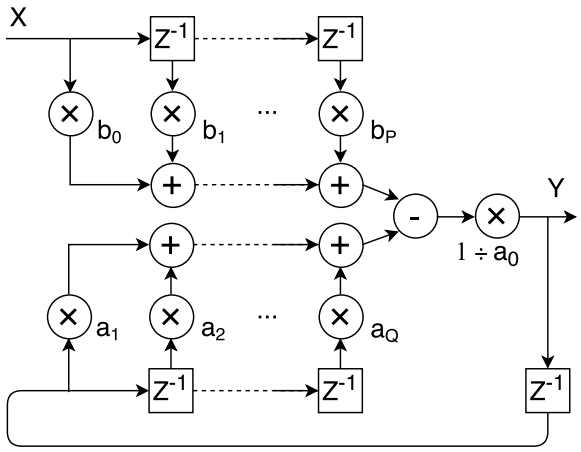

Figure 2. A direct form discrete-time IIR filter of order P and Q

Examining Figure 2, we see that this IIR filter loosely consists of two FIR filters, where the bottom one has an extra delay and is recursively defined. Also, the filter contains an extra subtraction and multiplication. This leads directly to the following code:

```
iir :: (Fractional a, Typeable a)
    ⇒ Expr a → [Expr a] → [Expr a]
    → Sig a → Sig a
iir a0 as bs x = y
  where
    y = (1 / repeat a0) * (upper x - lower y)
    upper = fir bs
    lower = fir as . delay 0
```

Circular definitions like this are possible because of the `delay` operator which makes sure that each output only depends on previous inputs, thus ensuring a productive network. In contrast, when defining our helper functions, `sums`, `muls` and `dels`, a different kind of recursion was used, leading to unfolding of the signal graph instead of feedback.

In general we have that recursively defined signals introduce feedback, while recursion over other Haskell values can be used to build graphs instead.

3. Embedding Programs

Before going into the implementation of signals, we will show how to embed an imperative programming language in Haskell. The choice of an imperative model is largely based on our interest in generating C code, but it could just as well have been some other sequential model. This embedded language will then form the basis for our implementation of signals.

The imperative language is implemented using the Operational package [2], which provides a monad, `Program cmd`, parameterized on its primitive instructions `cmd`. The idea behind Operational is that monadic programs can be viewed as sequences of instructions to be executed by some machine, so the task of implementing a monad is then the same as writing an interpreter for its instructions. It is possible to give multiple interpretations of the same program, and in

particular, it is possible to implement code generation just as a special program interpreter.

Our approach to code generation is similar to previously published methods [26, 27]. In these, code generation is implemented as a recursive function over a monadic program's graph representation, where each node corresponds to a monadic bind operation. Although Operational provides a similar functionality, we have chosen a different, and in some respects simpler approach – namely to use the following function:

```
interpretWithMonad :: Monad m
                   ⇒ (∀a.     cmd a → m a)
                   → (Program cmd b → m b)
```

This function lifts a monadic interpretation of the primitive instructions, which may be of varying types, to a monadic interpretation of the whole program. By using different types for the monad, `m`, one can implement different "back ends" for the programs. For example, interpretation in the `IO` monad gives a way to run the programs in Haskell, and interpretation in a code generation monad can be used to make a compiler to another language.

We will demonstrate our approach to code generation using a simple example: reading and writing numbers through IO. First, we define a concrete expression language to use:

```
data Expr a where
  Var :: String → Expr a
  Lit :: Integral a ⇒ a → Expr a
  Add :: Integral a ⇒ Expr a → Expr a
                             → Expr a
  deriving (Typeable)
```

These constructors represent numeric literals, variables and additions. `Expr` is indexed on the result type of the expression, and by using a generalized algebraic data type (GADT) we make sure that only well-typed expressions can be represented (barring the fact that variables can have any type, independent of the environment). As the instruction set, the following type is used:

```
data CMD a where
  Get :: CMD (Expr Int)
  Put :: Expr Int → CMD ()
```

where `CMD` is an instruction type with two commands: `Get` for reading from `stdin` and `Put` for writing to `stdout`.

By using `CMD` as the parameter to `Program`, a specialized monad for the imperative EDSL is made:

```
type Prog a = Program CMD a
```

which also hides the instruction set used from the end user.

For simple code generation to C, one needs a combination of the `State` and `Writer` monads, the former to thread a fresh name supply and the latter to collect a sequence of statements; C statements are represented as strings for simplicity:

```
type Code = StateT Int (Writer [String])
```

Two helper functions are also needed, one for generating fresh names and another for compiling pure expressions; again, C expressions are represented as strings:

```
gensym :: Code String
gensym = do v ← get; put (v + 1);
            return ("v" ++ show v)

compExpr :: Expr a → String
compExpr (Lit a) = show $ toInteger a
compExpr (Var v) = v
compExpr (Add a b)
    = "(" ++ compExpr a ++ " + "
          ++ compExpr b ++ ")"
```

Compilation of primitive instructions is then given by the functional `compCMD`:

```
compCMD :: CMD a → Code a
compCMD Get = do
    v ← gensym
    tell [v ++ " = getchar();"]
    return (Var v)
compCMD (Put a) =
    tell ["putchar(" ++ compExpr a ++ ");"]
```

The `compCMD` function generates code for a single instruction, and is lifted for compilation of whole programs:

```
compile' :: Prog a → Code a
compile' = interpretWithMonad compCMD

compile :: Prog a → String
compile = unlines . execWriter
        . flip execStateT 0 . compile'
```

As an example of what the generated code looks like, consider the following small program:

```
gett = singleton Get   :: Prog (Expr Int)
putt = singleton . Put :: Expr Int → Prog ()

prg :: Prog ()
prg = do a ← gett; b ← gett;
         putt (a + b + 1)
```

It gets two numbers and then puts their sum plus one, where `gett` and `putt` use of Operational's `singleton` function to lift their instructions into actions. The expression `a + b + 1` makes use of the following instance:

```
instance Integral a ⇒ Num (Expr a) where
    fromInteger = Lit . fromInteger
    (+)         = Add
    ...
```

The code generated from `prg` looks as follows:

```
*Main> putStr $ compile prg
v0 = getchar();
v1 = getchar();
putchar(((v0 + v1) + 1));
```

3.1 Abstracting away from the Expression Language

Note that the only thing we need to know about the `Expr` type in order to implement `compCMD` is that it has a `Var` constructor and a compilation function `compExpr`. So, to abstract away from the `Expr` type, we simply put those functions behind an abstract interface:

```
class CompExpr exp where
  type VarPred exp :: * → Constraint
  varExpr  :: VarPred exp a ⇒ String → exp a
  compExpr :: exp a → String
```

To account for the fact that different languages may have different constraints on the variable constructor, the asso-

ciated `VarPred` type allows us to use different constraints in each instance.[3] For example, in the instance for `Expr`, `VarPred Expr` maps to `Integral` due to the `Integral` constraint on the `Var` constructor:

```
instance CompExpr Expr where
  type VarPred Expr = Integral
  varExpr           = Var
  -- compExpr as before
```

We can now parameterize on the expression type used in `CMD`, in order to show that the technique described in this paper is independent of the expression language used.

```
data CMD exp a where
  Get :: CMD exp (exp Int)
  Put :: Expr Int → CMD exp ()
```

where the first parameter to `CMD` is the representation of pure expressions. Reimplementing `compCMD` using the abstract interface yields a compilation function with the following type:

```
compile
    :: (CompExpr exp, VarPred exp Int)
    ⇒ Program (CMD exp) a → String
```

As the type says, this version of `compileProg` is capable of compiling programs over *any* expression language, as long as it implements the `CompExpr` interface. Note the constraint `VarPred exp Int`, coming from the fact that `compCMD` needs to return a variable of type `exp Int` in the case for `Get`.

3.2 Running Programs

As previously noted, it is possible to run embedded programs in Haskell by providing an interpretation in the `IO` monad. Naturally, we would prefer to run such programs without constraining the expression language unnecessarily. A type class reminiscent of `CompExpr` is therefore introduced:

```
class EvalExpr exp where
  type LitPred exp :: * → Constraint
  litExpr  :: LitPred exp a ⇒ a → exp a
  evalExpr :: exp a → a
```

An instance for `Expr` is also made[4]:

```
instance EvalExpr Expr where
  type LitPred Expr  = Integral
  litExpr            = Lit
  evalExpr (Lit a)   = a
  evalExpr (Add a b) = evalExpr a
                     + evalExpr b
```

Based on `EvalExpr`, one can now conveniently define the behaviour of each primitive instruction and then lift that to get a run function for programs:

```
runCMD :: (EvalExpr exp, LitPred exp Int)
       ⇒ CMD exp a → IO a
runCMD Get     = fmap (litExpr . fromEnum)
               $ getChar
runCMD (Put a) = putChar $ toEnum
               $ evalExpr a

runProg :: (EvalExpr exp, LitPred exp Int)
        ⇒ Program (CMD exp) a → IO a
runProg = interpretWithMonad runCMD
```

Now that we have an embedding of imperative programs, we move on to describe the implementation of the signal library. By building it on top of the imperative embedding, we essentially get code generation for free.

4. Implementation

This section presents how signals are implemented and compiled to C code. While we stated earlier that signals represent sequences of values, the underlying implementation is a bit more involved: a signal consists of three different layers. At the bottom of these layers is the program layer, described in section 3. On top of that is the stream layer, a shallow implementation of co-iterative streams. Then, at the very top, is the signal layer, which extends the stream layer with support for observable sharing and rewriting.

4.1 Program Layer

Although our implementation is based on the same principle as the example in Section 3, it uses a more powerful expression language with support for many numerical and logical operations. The instruction set, `CMD`, has been extended as well and contains many more instructions – including ones with nested programs, such as this if statement whose branches are themselves programs:

```
data CMD exp a where
  ...
  If :: exp Bool
     → Program (CMD exp) a  -- true branch
     → Program (CMD exp) a  -- false branch
     → CMD exp a
```

While these extra instructions are useful for constructing more interesting examples, signals themselves only require basic functionality from the embedded language. More specifically, signals make use of mutable references – similar to `IORef` in Haskell. The interface to mutable references is given by the following functions:

```
newRef :: Typeable a
       ⇒ Expr a → Prog (Ref (Expr a))
getRef :: Typeable a
       ⇒ Ref (Expr a) → Prog (Expr a)
setRef :: Typeable a
       ⇒ Ref (Expr a) → Expr a → Prog ()
```

where `newRef` creates a new reference, `getRef` reads from a reference and `setRef` writes to one.

4.2 Stream Layer

The stream layer is an implementation of co-iterative streams, as described, for instance, by Caspi and Pouzet [9]. Co-iteration consists of associating to each stream a transition function from old state to a pair of a value and a new state, and an initial state. Thus, in Haskell we can model co-iterative streams as follows:

```
data Stream a where
       Stream :: (state → (a, state))
              → state
              → Stream a
```

[3] `VarPred` requires the `ConstraintKinds` extension in GHC.

[4] Note that we assume that evaluated expressions do not contain any variables. That is true for the code in this paper, as variables are only introduced during code generation, but there is nothing preventing us from violating this assumption in general.

This approach allows us to handle infinite data types, like streams, in a strict and efficient way, instead of having to deal with them using lazy data structures such as lists. As a result of this, stream transformers are free to pick apart the input streams in any way they see fit. Fusion of streams is achieved by construction.

A slightly modified version of these co-iterative streams is used in our library, where the state is represented implicitly by the `Program` monad:

```
data Stream a = Stream (Prog (Prog a))
```

In this definition, the outer monad is used to initialize the stream, and the result of the initialization is a monadic action that can be executed repeatedly to produce the stream of values. Since the program layer supports stateful computations, this representation allows for the state to be updated while running. Since we often have streams with `Expr` in the result type, we define a convenient shorthand:

```
type Str a = Stream (Expr a)
```

We can use this type to define stream versions of the `repeat` functions from Section 2:

```
repeatStr :: Expr a → Str a
repeatStr a = Stream $ return $ return a
```

Since `repeatStr` does not carry any state, it has an empty initialization and always produces the same value, `a`. In a similar manner, we can also define a stream version of the `map` function:

```
mapStr :: (Expr a → Expr b) → Str a → Str b
mapStr f (Stream init) = Stream $ do
  next ← init
  return $ do
    a ← next
    return (f a)
```

Unlike `repeatStr`, `mapStr` carries state, or at least the state of its argument. As a result, `init` is run to initialize the stream and access the argument's transition function `next`. Output is then produced by running `next`, transforming each value it produces using `f` before returning it.

These are the combinatorial functions we can implement using streams, but as the `Program` type also supports mutable state, it is also possible to have sequential, state-carrying, nodes. For instance, we can implement a function corresponding to `delay` from Section 2:

```
delayStr :: Typeable a ⇒
            Expr a → Str a → Str a
delayStr a (Stream init) = Stream $ do
  next ← init
  r    ← newRef a
  return $ do
    v ← next
    b ← getRef r
    setRef r v
    return b
```

This definition is similar to `mapStr`, but instead of transforming values, it stores them to be returned in the next iteration. Note how the local state is created in the initialization stage, after intializing the input stream.

By defining a `zipWithStr` fucntion, analogously to `mapStr`, we can reimplement all functions from Section 2 for streams. However, this leads to problems: because streams are implemented as a shallow embedding on top of programs, we lose the ability to analyze and optimize the dataflow networks. The problem can be seen, for example, in the definition of `mapStr`, where the code for the output stream includes the whole code for the input stream. So the result of `mapStr` is code where we can no longer see where nodes begin and end.

This problem is more severe than it may seem at first:

- Nodes that are shared in the network will lead to duplication in the generated code.
- Related to the above, feedback loops will lead to code being duplicated indefinitely.

Additionally, there are many optimizations that can be performed if the network can be analyzed. For example, it is more efficient to implement delay lines, as generated by the earlier `dels` function, using circular buffers instead of individual delay elements. We are interested in doing such rewriting optimizations automatically.

4.2.1 Feldspar's Solution

It is possible to work around the above problems by defining combinators for creating specific types of networks. This is what was done earlier in Feldspar. Feldspar has a stream library[5] implemented roughly as described in this section.

One of Feldspar's network combinators is `recurrenceIO`, defining a feedback network with one input and one output:

```
recurrenceIO :: (...)
     ⇒ Vector1 a        -- initial in buffer
     → Stream (Data a)  -- input stream
     → Vector1 b        -- initial out buffer
     → (Vector1 a → Vector1 b → Data b)
     → Stream (Data b)  -- output stream
```

The fourth argument represents the actual computation in the network: a function that, given vectors of previous inputs and outputs, computes the next output. The sizes of the initial buffers determine the number of previous values that are remembered. Restricting access to previous values in this way enables efficient memory management, and the vectors are implemented as circular buffers internally.

Given this `recurrenceIO` function, it is possible to express recursively defined streams. For instance, Feldspar's stream library includes the following definition of an IIR filter:

```
iir :: Data Float → Vector1 Float
     → Vector1 Float → Stream (Data Float)
     → Stream (Data Float)
iir a0 as bs x =
   recurrenceIO (replicate (length bs) 0) x
                (replicate (length as) 0)
     (λi o → 1 / a0 * ( scalarProd bs i
                      - scalarProd as o))
```

where `scalarProd` is the scalar product on Feldspar's vectors.

While functions such as `recurrenceIO` allow efficient networks to be expressed, they do not provide a very satisfactory solution to programming with streams. First of all, they are quite unintuitive to work with. Worse still, each combinator only captures a specific kind of network. For this reason, Feldspar provides a number of recurrence combinators differing only in the number of input and output streams they handle. All in all, the approach does not scale very well.

However, the solution to the above problems is not to abandon streams altogether – instead, we wrap streams in

[5] http://hackage.haskell.org/package/feldspar-language-0.7/docs/Feldspar-Stream.html

a layer using a deep embedding. This new layer will then remedy the problems inherent in a shallow model while still having access to streams and the nice properties they have for code generation.

4.3 Signal Layer

The last layer, called the signal layer, will act as our wrapper layer for the underlying stream model. As such, it provides a way to promote streams and stream functions into a corresponding signal version:

```
data Signal a where
  Const :: Typeable a
        ⇒ Str a → Signal (Expr a)

  Lift  :: (Typeable a, Typeable b)
        ⇒ (Str a → Str b)
        → (Signal (Expr a) → Signal (Expr b))


type Sig a = Signal (Expr a)
```

`Const` is used for lifting constant streams, while `Lift` lifts a stream transformer into a signal transformer. Like `Str`, `Sig` defines a shorthand for signals that return expressions.

Most of the signal library's functionality comes from lifting stream functions; for instance, the previous `map` and `repeat` functions are both implemented in this way:

```
repeat :: Typeable a ⇒ Expr a → Sig a
repeat a = Const (repeatStr a)

map :: (Typeable a, Typeable b)
    ⇒ (Expr a → Expr b) → Sig a → Sig b
map f = Lift (mapStr f)
```

Defining signal functions in this way introduces a ambiguity to signals: how to distinguish between pairs of signals and signals of pairs. Additional constructors are therefore included, for managing the flow of a signal and helping to differentiate between the two kinds of signals:

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

Manual zipping and unzipping can however get tiresome. Signals are therefore extended with a generalized constructor, `Map`, which allows one to have signal functions which take and produce arbitrary trees of signals. These trees are represented using the `Struct` type.

```
data Signal a where
  ...
  Map :: (Typeable a, Typeable b)
      ⇒ (Struct a → Struct b)
      → (Signal a → Signal b)


data Struct a where
  Leaf :: Typeable a
       ⇒ Expr a → Struct (Expr a)
  Pair :: Struct a → Struct b
       → Struct (a, b)
```

Armed with these two, it is possible to implement a more general zipping function `zipWith`, which joins the two signals element-wise using the given function:

```
zipWith
    :: (Typeable a, Typeable b, Typeable c)
    ⇒ (Expr a → Expr b → Expr c)
    → Sig a → Sig b → Sig c
zipWith f a b =
    Map (λ(Pair (Leaf a) (Leaf b))
              → Leaf (f a b)) $ Zip a b
```

Defining functions in this way is not as elegant as one would like it to be. It is possible, however, to automate this process, and rewrite the previous definition of `zipWith`:

```
zipWith f = curry $ lift $ uncurry f
```

What the `lift` does is to transform Haskell tuples into signals, using `Zip`, apply the function, and then transform them back again.

Sequential networks, as shown in section 2.2, can be expressed using recurrence equations. However, due to their limitations, we prefer to avoid using them entirely and instead program feedback using the more general `delay` function shown in section 2. A dedicated `Delay` constructor is therefore included in the `Signal` type:

```
data Signal a where
  ...
  Delay :: Typeable a
        ⇒ a → Signal a → Signal a


delay :: Typeable a
      ⇒ Expr a → Sig a → Sig a
delay = Delay
```

Explicitly describing feedback networks that include delays is much more user friendly than imposing a fixed set of combinators for recurrence equations.

4.4 Running signals

Now that we have defined all the layers, we would like to be able to peel them off. As signals are a wrapper for the underlying stream model, we want them, and the overhead associated with them, to disappear entirely during the code generation process.

This brings us up against a common problem in manipulating data types representing embedded languages: how to observe sharing and cycles in trees generated from a deeply embedded language. Programs in the signal library represent data flow graphs, which can contain shared nodes and cycles. But the `Signal` type is a tree. In order to view the `Signal` type as a graph, we need to examine how this tree is represented in memory (because even trees are represented as graphs internally by GHC).

One approach to observable sharing, as proposed by Gill [15], relies on GHC-specifics to provide an `IO` function capable of observing sharing directly. Given such a function, it is possible to define a function that strips away the signal layer and gives back a stream function. We call this the `compile` function:

```
compile :: (Typeable a, Typeable b)
    ⇒    (Sig a → Sig b)
    → IO (Str a → Str b)
compile f = do
  (Graph nodes root) ← reifyGraph f
```

```
let links = linker nodes
    order = sorter root nodes
    cycle = cycles root nodes

return $ if cycle
  then error "Cycle found"
  else compiler nodes links order
```

The function `reifyGraph` is the one that turns a signal function into its representation as a directed acyclic 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. Reification of signals through observable sharing therefore implies that a recursive data type is used to represent the embedded language, like `Signal`, coupled with a mirror type where all points of recursion have been replaced by abstract references:

```
data TSignal r where
  TConst :: (Typeable a) ⇒ Str a → TSignal r
  TLift  :: (Typeable a, Typeable b)
         ⇒ (Str a → Str b) → r → TSignal r
  TZip :: r → r → TSignal r
  TFst :: r     → TSignal r
  ...

type Node = TSignal Int
```

Typing information is however lost after reification, as nodes are no longer connected by types. Typecasting is therefore used internally to connect these reified nodes, hence the need for `Typeable`.

As a means to recover some of the lost information, two accessory functions are used in the compiler: `linker` and `sorter`. As their names suggest, `linker` reconnects nodes with their expected input and output sources, while `sorter` sorts the nodes in topological order. Furthermore, as we cannot compile un-delayed feedback loops, `cycles` is used to check for valid signal networks before they are compiled.

Sorting and cycle detection of graphs are both well-studied problems and efficient solutions exist for both. Our `sorter` and `cycles` are implemented as follows: the graph is traversed in a depth-first manner, making sure to avoid loops by tagging nodes as they are visited. These two functions are given the following type signatures:

```
sorter :: Int → [(Int, Node)] → Map Int Int

cycles :: Int → [(Int, Node)] → Bool
```

Worth mentioning is the treatment of delays while detecting cycles, as cycles should only be rejected if they do not contain any delays. The solution we chose was to simply ignore all outgoing edges from a delay node. This breaks all "good" cycles and leaves the "bad" ones for us to detect. Also, as a result of edges being removed, we might end up splitting the graph into subgraphs but we can safely process these one at a time.

Lastly, we have the `linker` function, which reconnects nodes with their input sources, while also recovering parts of the typing lost during reification. More specifically, `linker` provides a typed `Struct` where each value has been replaced by a reference to the node that produces it.

Implementing `linker` as depth-first traversal of the graph – in the same manner as `sorter` or `cycles` – does however turn out to be unsatisfactory: `linker` must account for both the ordering of nodes and avoid cycles. As a result of this, its logic will be hidden behind all the necessary bookkeeping.

Instead, `linker` can be better expressed using a circular programming technique. We employ the same technique as Axelsson [4] to express circular knot-tying programs using a monad. This allows us to both declare connection constraints and read their solution at the same time, without having to worry about doing so in the correct order.

The general idea behind knot-tying is to make use of Haskell's laziness and a combination of the `Reader` and `Writer` monads:

```
import Control.Monad.Reader
import Control.Monad.Writer

type Knot r c m = ReaderT r (WriterT [c] m)
```

Constraints can now be declared through `Writer` and solutions are read using `Reader`. All that's left is to have Haskell's laziness tie the constraints and solution together:

```
tie :: MonadFix m ⇒ ([c] → r)
                  → Knot r c m a
                  → m (a, r)
tie solver knot = mdo
  (a, cs) ← runWriterT $ runReaderT knot r
  let r = solver cs
  return (a, r)
```

where `mdo` is notation for recursive monadic programs. The first argument to `tie` is the solver, which computes a solution from a list of constraints.

Applying the `Knot` monad to our problem of linking means we need to come up with a model for our constraints and solutions, which we would then write in bulk and have `tie` sort out their correct ordering. For this reason, a new type, called `TStruct`, is introduced to store the connection solutions. `TStruct` has a similar form to `Struct` but contains references at the leaves instead of values:

```
data TStruct a where
  TLeaf :: String → TStruct a
  TPair :: TStruct a → TStruct b
        → TStruct (a, b)
```

The type of these structs will however differ from node to node, which means that they cannot be stored directly in a plain map. Yet another datatype is therefore introduced, modeling existential types:

```
data EStruct where
  Ex :: Typeable a ⇒ TStruct a → EStruct

data Res = Resolution
  { _in  :: Map Int EStruct
  , _out :: Map Int EStruct }
```

Constraints are then simply members of these mappings, represented as:

```
data Cons = In (Int, EStruct) | Out (Int, EStruct)
```

As the knotty details of cycles and orderings are now entirely managed by knots, our `linker` can focus on describing the connections between nodes.

```
linker :: [(Int, Node)] → Res
linker = snd . runIdentity . tie solve
             . sequence    . map link
```

```
solve :: [Cons] → Res
solve cons = Resolution
  { _in  = fromList [i | In  i ← cons]
  , _out = fromList [o | Out o ← cons] }

link :: Monad m ⇒ (Int, Node)
                → Knot Res Cons m ()
link (i,TConst  _) =
  tell [Out (i, Ex $ TLeaf $ show i)]
link (i,TLift _ _) =
  tell [Out (i, Ex $ TLeaf $ show i)]
link (i,TZip l r)  = do
  (Ex u) ← asks ((! l) . _out)
  (Ex v) ← asks ((! r) . _out)
  tell [Out (i, Ex $ TPair u v)]
...
```

Some type casting is required in `link`, but has been omitted to improve readability. Furthermore, nodes like `Zip`, which only forward their input references, can safely be removed from the final mapping as no other nodes will reference them – simplifying networks by rewriting them in a safe manner.

Linking nodes together in this manner works as long as we can reconstruct the expected input and output types. When the necessary information isn't available, we extend the constructor to include the missing pieces. For example, in order to link a `Map` node, we add two trees to its constructor:

```
data TSignal r where
  ...
  TMap :: (Typeable a, Typeable b)
       ⇒ TStruct a → TStruct b
       → -- same as before
```

Given these two pieces of extra information, we can implement `link` for `TMap` nodes:

```
...
link (i, TMap ti to f s) = do
  (Ex (u :: TStruct ti)) ← asks ((! s) . _out)
  tell [In  (i, Ex u)]
  tell [Out (i, Ex $ mark to $ show i)]

mark (TLeaf _)   s = TLeaf s
mark (TPair l r) s = TPair l' r'
  where l' = mark l $ s ++ "_l"
        r' = mark r $ s ++ "_r"
```

Compilation of streams is now straightforward, and our `compiler` function is given the following type:

```
compiler :: (Typeable a, Typeable b)
         ⇒ Map Int Node    -- nodes
         → Map Int EStruct -- links
         → Map Int Int     -- order
         → (Str a → Str b)
```

The general idea is to simply traverse each node in the given order and lift their logic into programs as we go. This behavior is captured by the following pattern:

```
compileNode (id, node) = do
  -- fetch input addresses
  -- apply node logic
  -- store result in variables
  -- write said variables to output addresses
```

where input and output addresses are given by the links mapping. Also, since `linker` connected nodes directly with their input sources, compiling pure flow nodes, like `Zip`, amounts to simply doing nothing at all:

```
compileNode (id, Zip {}) = return ()
```

Signal functions can now be compiled into streams using the previous `compile` function, which acts as an inverse of signal's `Lift`.

Then, as streams are no more than programs in disguise, we can generate code from them using the previously shown techniques. Inspecting the code produced in this way shows that all signal specific constructions, like delays, have indeed been removed by the compiler, leaving behind only arithmetic operations and control structures.

For example, applying the compiler to a rank-2 FIR filter produces the following code:

```
r6 = 0.0; r7 = 0.0; r3 = 0; r19 = 0;
while (r19 ≤ 999) {
    float a20;
    a20 = a0[r3];
    r8 = a20;
    r11 = 1.1;
    r10 = r11 * r8;
    r14 = -0.55;
    r15 = r6;
    r13 = r14 * r15;
    r17 = 0.275;
    r18 = r7;
    r16 = r17 * r18;
    r12 = r13 + r16;
    r9 = r10 + r12;
    r7 = r15;
    r6 = r8;
    a2[r3] = r9;
    r3 = r3 + 1;
    r19 = r19 + 1;
}
```

Here we only show the actual filter loop and omit all variable declarations. The input and output arrays are `a0` and `a2`, respectively. Variables `r14` and `r17` are the filter coefficients.

4.5 Internalized IO

So far, we have seen pure signal functions, such as `repeat`, `map` and `zipWith` and a single stateful function – `delay`. The pure functions are implemented using `Const` and `Lift`, and `delay` is given a specific constructor in the `Sig` type. However, there is in fact nothing stopping us from using `Const`/`Lift` to lift stateful or side-effecting functions to the signal layer. As an example, given a function that gets the current time of day as a number, we can easily make a signal source that gives the current time in each cycle:

```
getTime :: Prog (Expr Double)

time :: Sig Double
time = Const $ Stream $ return getTime
```

In a similar way, it would be possible to make signal functions for reading and writing to files or other resources. One complication then is that the lifted operations must be *non-blocking*, since the dataflow network must be able to keep running even when data is not available.

We have not yet explored the possibility of having networks with internalized IO, but it appears to be very convenient way of programming embedded systems. Rather than having our previous `compile` function with exactly one in-

put and one output signal, one could then instead have a function such as the following:

```
compile :: Typeable a
        ⇒ Sig a → IO (Program ())
```

The argument is a dataflow graph with all external communication built-in and the result is a monolithic program that can be invoked repeatedly to execute the dataflow network.

Although convenient for programming, the disadvantage of using internalized IO is that we can no longer view our networks as simple state machines with a pure transition function from previous to next state. With internalized IO, the "state" can easily reside outside of the program, and the transitions are generally non-deterministic (for example, the `time` source is unlikely to give the same sequence of numbers in two different runs of a system).

4.6 Abstracting away from the Expression Language

As we saw in Section 3.1, the underlying representation of imperative programs abstracts away from the representation of pure expressions. Similarly, while signals and streams have been hard-coded to use `Expr`, our actual implementation uses a more general type:

```
data Stream exp a =
       Stream (Program (CMD exp)
                       (Program (CMD exp) a))

data Signal exp a where
    Const :: Typeable a
          ⇒ Stream exp a → Signal exp a

    Lift  :: (Typeable a, Typeable b)
          ⇒ (Stream exp a → Stream exp b)
          → (Signal exp a → Signal exp b)
    ...
```

In the interface towards the user, we hide this generality and use types that are equivalent to the ones presented in this section:

```
type    Str a = Stream Expr (Expr a)
newtype Sig a = Sig (Signal Expr (Expr a))
```

Note that the code developed in this section is essentially the same when written with more general types.

5. Evaluation

The aim of this work is to be able to generate efficient code that can be used in applications with high performance demands. In order to evaluate the generated code, we have compared it to reference implementations written in C.[6] The referenced C code uses an efficient circular buffer to store previous input values. The IIR filter was implemented by ourselves by simply extending the FIR filter, hopefully without introducing accidental inefficiencies.

Each of our tests applies a filter to a 1000-element vector. In order to reduce the impact of system jitter, each test is repeated a large number of times (10000), and the time reported is the average over those iterations. The measurements were carried out on an Dell Latitude 7420 laptop with an Intel Core i7 processor using GCC version 4.9.1.

[6] The FIR filter's reference implementation was obtained from http://www.iowahills.com/A7ExampleCodePage.html

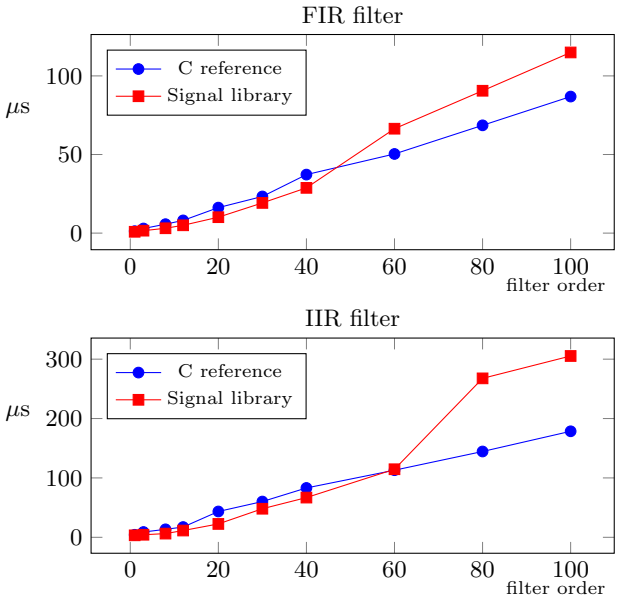

Figure 3. Running time of filters compared to reference C implementations.

The results for filters of varying orders are shown in Figure 3. The code from our library performs slightly better than the reference up to filter orders of around 50. For larger sizes it slows down because our library unrolls the inner loops of the filter, while the reference implementation uses nested loops. So the size of our code grows linearly with the filter order, and after a certain point the loop body becomes too big to be able to run efficiently (e.g. not enough registers to hold intermediate values).

The reason our library unrolls the inner loops is that we make use of recursive Haskell functions, such as `map` and `foldr1` to generate signal graphs. In order to generate nested loops instead, we would need to make use of more complex nodes, where, for example the multiply-accumulate part of the filter is represented as a single node whose input is an array. This in turn would require a more powerful expression language that has arrays and loops.

We are working on integrating the signal library with Feldspar. This will allow us to express filters with nested loops.

6. Related Work

As discussed in section 2, our library is based on the synchronous data flow paradigm and is heavily inspired by related languages from that paradigm, such as the Lucid Synchrone language [13, 24].

Lucid Synchrone is a member of the family of synchronous languages and is designed to model reactive systems. It was introduced as an extension of LUSTRE [17], and demonstrated that the language could be extended with new and powerful features. For instance, automatic clock and type inference were introduced, and a restricted form of higher-order functions was added.

However, Lucid Synchrone is a standalone language that cannot easily be integrated with EDSLs such as Feldspar. For this reason, we chose to implement a new library, partly

inspired by Lucid Synchrone, that brings an SDF programming model to existing EDSLs, such as Feldspar.

Another, rather different approach to modeling signal processing is used by Yampa [14]. Yampa is a member of the functional reactive programming paradigm, used for programming hybrid systems, that is, systems consisting of both continuous and discrete components. 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 [22].

Apart from the different notions of time, the behaviors of Yampa are quite similar to the synchronous concept of signals. Zélus [8], a successor of Lucid Synchrone, has shown that synchronous languages can indeed be extended to model hybrid systems as well. An event is usually a separate entity, modeling the control flow of a system. Some languages do, however, merge the two concepts, by describing events in terms of behaviors. This unification comes at a cost of some elegance, as behaviors can now be discrete, but it also means that it is possible to model such systems with only data flow.

Furthermore, functional reactive programming languages are commonly based on an applicative interface or arrows [19], which makes them a poor fit for a restricted EDSL like Feldspar. This comes as a result of type classes assuming that the underlying language is a super-set of Haskell, since every Haskell function can be promoted to an expression in the language.

Lava is a family of Haskell EDSLs designed for expressing hardware descriptions [7]. Chalmers Lava [11] is an experimental tool designed to assist circuit designers with hardware design, specification and verification.

Chalmers Lava is a structural hardware description language, embedded in Haskell in such a way that the designer actually writes and composes netlist generators. It is an archetypical deeply embedded language, with various backends operating on the data type representing circuits. Its most recent incarnation allows for the description of hardware by providing a polymorphic unit delay element and the ability to describe feedback (observable sharing). However, the other building blocks of ciruits are restricted to be simple gates. Here, in the signal library, we permit the combination of much more interesting building blocks, as our dataflow nodes can run arbitrary code – and indeed the users of the signal library can introduce their own program layer.

Kansas Lava [16] is another member of the Lava family of hardware description languages. It reimplements and extends the original Lava design patterns. One important extension is direct support for including new blocks of existing VHDL libraries as new, well typed primitives. Another noteworthy aspect of Kansas Lava is the use of a dual deep and shallow embedding (for synthesis and simulation) inside its `signal` type. This gives a clear design flow, starting from a Haskell function, adding applicative functors to give a model that understands time (shallow embedding) and then factoring out the applicative functors to give a synthesiable model (deep embedding).

Both of these Lava languages make use of observable sharing, where Chalmers Lava makes use of explicit references [12] and Kansas Lava uses Gill's type-safe version, which is also used in the signal library [15].

The ForSyDe (Formal System Design) methodology and tools [25] explore methods of developing heterogeneous systems in which different parts of the system may have different models of computation (one of which is the synchronous model). An initial specification is refined into a detailed implementation model by the application of formally justified transformations. One of the implementations of ForSyDe is as an embedded language in Haskell and here, once again, the combination of deep and shallow implementations of a signal API is considered.

The way in which our program, stream and signal layers are layered one above the other is reminiscent of the Ivory and Tower EDSLs being developed at Galois Inc [18].

Ivory is an EDSL for generating safe, embedded C code, and it places particular emphasis on the definition of in-memory data structures and of how they should be manipulated. In our work on Feldspar, that aspect of C programming is covered largely by another "system layer" being developed to target a particular platform (the Adapteva Parallella board), above the Feldspar layer producing individual C functions. Ivory/Tower also has a layer for composing Ivory programs; the Tower layer covers the "glue code" that implements inter-process communication, processor locking, system clock reading etc. The utility and practicality of the layered EDSL approach is very well explained in the Galois experience report on this work [18].

In the work on the signal library described here, we have more and "thinner" layers than in the Ivory/Tower work. We expect to have yet more layers above the signal layer, although we have not yet decided exactly how to combine our various system layers.

7. Discussion

Writing digital signal processing software in C is a potentially tedious and error-prone task, primarily because of the enforced focus on low-level implementation details. We have described a library that alleviates this problem by extending an EDSL modeling pure functions with support for synchronous dataflow operations, allowing many low level details to be handled by the relevant compilers.

By using an EDSL for pure computations and Haskell's type classes, we can retain the elegance and modularity of traditional functional programming for signals. We are mainly interested in using Feldspar as the expression language; however, the library is not dependent on Feldspar, and so may be of interest to other EDSL developers.

We also intend to further build on earlier work on Lucid Synchrone (and LUSTRE) by introducing clocks as types [6]. This will allow us to deal correctly with sampling and oversampling, and to describe and implement multi-clocked networks. We feel that this will result in an interesting and expressive programming language for our envisioned users.

Having made the step to clocks, we are interested in the question of how to generate circuits rather than programs from our dataflow networks.

The intention is to grow the program layer to cover a substantial part of C. In order to support more complex systems, we also need a more powerful expression language. However, rather than extending the `Expr` type, our plan is to use the existing Feldspar EDSL [5] to represent pure expressions. This will require more advanced interleaving between the compilation phases for programs and expressions, as Feldspar expressions may generate complex chunks of C code.

Even if Feldspar ends up as our only expression language, we feel that the idea of establishing an interface towards the expression compiler (i.e. the `CompExpr` and `EvalExpr` classes

from Section 3.1) is a good one, as it allows us to treat the two parts of the compiler separately.

Acknowledgments

This research was funded by the Swedish Foundation for Strategic Research (in the RAW FP project) and by the Swedish Research Agency (Vetenskapsrådet). Anders Persson has provided valuable help on C code generation and benchmarking.

References

[1] Cisco visual networking index: Forecast and methodology, 2012–2017, May 2013. URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf.

[2] H. Apfelmus. The operational package, version 0.2.3.2. URL https://hackage.haskell.org/package/operational.

[3] M. Aronsson. Signal processing for embedded domain specific languages, March 2015. URL https://hackage.haskell.org/package/signals.

[4] E. Axelsson. *Functional programming enabling flexible hardware design at low levels of abstraction.* PhD thesis, Chalmers University of Technology, 2008.

[5] E. Axelsson, K. Claessen, G. Dèvai, Z. Horvàth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and 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*, pages 169–178, July 2010.

[6] D. Biernacki, J.-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2008.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184. ACM, 1998. ISBN 1-58113-024-4. URL http://doi.acm.org/10.1145/289423.289440.

[8] T. Bourke and M. Pouzet. Zélus: A Synchronous Language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Mar. 2013. URL http://www.di.ens.fr/~pouzet/bib/hscc13.pdf.

[9] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11(0):1 – 21, 1998. ISSN 1571-0661. URL http://www.sciencedirect.com/science/article/pii/S1571066104000507.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and 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, pages 178–188. ACM, 1987. ISBN 0-89791-215-2. URL http://doi.acm.org/10.1145/41625.41641.

[11] K. Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Chalmers University of Technology, 2001.

[12] K. Claessen and D. Sands. Observable sharing for functional circuit description. In P. Thiagarajan and R. Yap, editors, *Advances in Computing Science — ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66856-5. URL http://dx.doi.org/10.1007/3-540-46674-6_7.

[13] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 230–239. ACM, 2004. ISBN 1-58113-860-1. URL http://doi.acm.org/10.1145/1017753.1017792.

[14] A. Courtney, H. Nilsson, and J. Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 7–18. ACM, 2003. ISBN 1-58113-758-3. URL http://doi.acm.org/10.1145/871895.871897.

[15] A. Gill. Type-safe Observable Sharing in Haskell. In *Proceedings of the second ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128. ACM, 2009. ISBN 978-1-60558-508-6. URL http://doi.acm.org/10.1145/1596638.1596653.

[16] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. In M. Morazán and S.-B. Scholz, editors, *Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16477-4. URL http://dx.doi.org/10.1007/978-3-642-16478-1_2.

[17] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Number 215 in International Series in Engineering and Computer Science. Springer, 1993.

[18] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded DSLs. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 3–9. ACM, 2014. ISBN 978-1-4503-2873-9. URL http://doi.acm.org/10.1145/2628136.2628146.

[19] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37(1–3):67 – 111, 2000. ISSN 0167-6423. URL http://www.sciencedirect.com/science/article/pii/S0167642399000234.

[20] 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.

[21] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987. ISSN 0018-9219.

[22] H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64. ACM, 2002. ISBN 1-58113-605-6. URL http://doi.acm.org/10.1145/581690.581695.

[23] A. V. Oppenheim, R. W. Schafer, J. R. Buck, et al. *Discrete-time signal processing*, volume 2. Prentice-hall Englewood Cliffs, 1989.

[24] M. Pouzet. Lucid Synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006. URL http://www.di.ens.fr/~pouzet/lucid-synchrone/manual_html/.

[25] I. Sander. *System Modeling and Design Refinement in ForSyDe.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003. URL http://web.it.kth.se/~ingo/Papers/Thesis_Sander_2003.pdf.

[26] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 287–298. ACM, 2013. ISBN 978-1-4503-2326-0. URL http://doi.acm.org/10.1145/2500365.2500602.

[27] J. D. Svenningsson and B. J. Svensson. Simple and Compositional Reification of Monadic Embedded Languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 299–304. ACM, 2013. ISBN 978-1-4503-2326-0. URL http://doi.acm.org/10.1145/2500365.2500611.