– – *DRAFT, extended abstract submitted to IFL 2014* – –

Stream Processing for Embedded Domain Specific Languages

Markus Aronsson Emil Axelsson Mary Sheeran
Chalmers University of Technology
mararon@student.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 without any overhead associated with the high-level programming model. 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.

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 [16]. Mobile communications in particular have been growing at a phenomenal rate, which can be largely attributed to the rising popularity of mobile terminals.

For telecommunications infrastructure, the consequences of such a rapid growth rate have been a dramatic increase in the demand for 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 the the recent diversification in available architecture. These factors, while positively influencing the available computational power, have also significantly increased the complexity of developing new solutions for telecommunication systems.

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 (EDSL), Feldspar [3], 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) [19], which offers natural, high-level descriptions of streaming algorithms, while still permitting the generation of efficient code. Feldspar does have a library for programming with synchronous streams, but that library is quite low-level and tedious to use.

This paper describes a library for extending an existing Haskell EDSL 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. We are 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.

This paper makes the following contributions:

- We present a simple EDSL for synchronous dataflow programming in Haskell. Practically, the result is a useful addition to Feldspar.
- We make use of observable sharing [7] to achieve a deep embedding without relying on combinators to express sharing or cycles. This technique has long been used in the hardware description EDSL Lava [6, 11], but our work now permits the combination not just of simple gates but of arbitrarily complex EDSL programs.
- We abstract away from the underlying expression language by establishing an interface for the underlying expression compiler.
- We show how to generate C code from our library, and evaluate the results on examples.

1.1 Synchronous dataflow programming

Dataflow programming is a paradigm which internally models an application as a directed graph [17, 21], similar to a dataflow diagram. Nodes in the graph are then executable blocks, representing the different components of an application: they receive input, apply some transformation, and forward 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.

A later extension to dataflow programming is the introduction of *synchronous* dataflow. 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. Restricting the dataflow model in this way has the advantage that it can be statically scheduled [18], which, in turn, allows for generation of efficient code.

Lucid Synchrone [8, 20] is a member of the family of synchronous languages and is designed to model reactive systems. It was introduced as an extension of LUSTRE [13], 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 form of higher-order functions was added. However, Lucid Synchrone is a standalone language which cannot easily be integrated with EDSLs such as Feldspar. For this reason, we chose to implement a library, partly inspired by Lucid Synchrone, that brings an SDF programming model to existing EDSLs, such as Feldspar.

2. Signal

This library is based on the concept of a *signal*, which represents an infinite sequence of values in some pure expression language. Signals are constructed by the following interface:

```
map    :: (exp a → exp b)
          → Signal exp a
          → Signal exp b

repeat :: exp a → Signal exp a

zip    :: Signal exp a
          → Signal exp b
          → Signal exp (a, b)

fst    :: Signal exp (a, b)
          → Signal exp a
```

where `exp` is the pure expression language. The `map` function promotes a pure function to operate over signals; `repeat` makes a constant-valued signal; `zip` and `fst` are used to make nodes with multiple incoming or outgoing signals.

Sequential operations are supported through the following functions, which manage a signal's phase and frequency:

```
delay  :: exp a
          → Signal exp a
          → Signal exp a

sample :: exp Int
          → Signal exp a
          → Signal exp a
```

While few in number, these sequential functions are quite general and allow for arbitrary feedback networks to be expressed.

The need to implement particular signal functions may place demands on the underlying expression language, in that support for common data types or functionality may be required. For instance, in order to implement a signal version of Haskell's `zipWith` function, the expression language needs to support tuples:

```
class TupExp exp
  where
    tup :: exp a → exp b → exp c
    fst :: exp (a,b) → exp a
    snd :: exp (a,b) → exp b

zipWith :: (TupExp exp, Signal exp ~ sig)
           ⇒ (exp a → exp b → exp c)
           →  sig a → sig b → sig c
zipWith f s u = map (λp → f (fst p) (snd p))
              $ zip s u
```

Classes such as `TupExp` provide a suitable interface with the expression language, but without forcing a particular language to be hardwired into the system.

2.1 Example: FIR Filter

Consider the mathematical definition of a finite impulse response filter of rank N:

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

This description is convenient for software realization, as it can be deconstructed into three main components: a number of successive unit delays, multiplication with coefficients and a summation. We can represent the decomposed filter graphically, as in Figure 1.

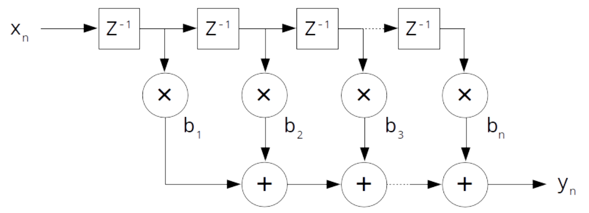

Figure 1. A direct form discrete-time FIR filter of order N

Support for such numerical operations over signals is implemented by instantiating their corresponding classes in Haskell:

```
instance (TupleExp exp, Num (exp a)) ⇒
    Num (Signal exp a)
  where
    fromInteger = repeat . fromInteger
    (+)         = zipWith (+)
    (-)         = zipWith (-)
    ...
```

where similar instance declarations can be made for fractional and floating point arithmetic.

Using Haskell's standard classes in this way simplifies the construction of signals by providing a homogeneous user interface. Furthermore, as the pure Haskell code is separated from our signal library in this way, it introduces a meta-level of computation, helping us to reason about program correctness. 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 further reducing the complexity of developing new networks.

Given this support for numerical operations, we now create helper functions, modeling the three main components of the FIR filter: summation and multiplication of signals and successive delaying. Using Haskell's standard library functions, summation can be neatly expressed as a single fold operation:

```
import qualified Prelude as P

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

Similarly, both of the remaining components can be expressed using standard Haskell functions:

```
muls :: (TupExp exp, Num (exp a))
        ⇒ [exp a]
        → [Signal exp a]
        → [Signal exp a]
muls = P.zipWith (λc s → repeat c * s)

delays :: [exp a]
       → Signal exp a
       → [Signal exp a]
delays as s = P.tail
            $ P.scanl (P.flip delay) s as
```

The FIR filter can now be neatly expressed as:

```
fir :: (TupleExp exp, Num (exp Float))
        ⇒ [exp Float]
        → Signal exp Float
        → Signal exp Float
fir bs = sums . muls bs . delays ds
  where
    ds = P.replicate (P.length bs) 0
```

This description is close to the filter's graphical representation, a beneficial attribute since domain experts in DSP tend to be comfortable with composing sub-components in this way.

2.2 Example: IIR Filter

Infinite impulse response (IIR) filters are digital filters with an infinite impulse response and, unlike FIR filters, contain feedback. These filters will therefore 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 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, and a_j and b_i are the filter coefficients. We can represent the decomposed filter graphically, as in Figure 2.

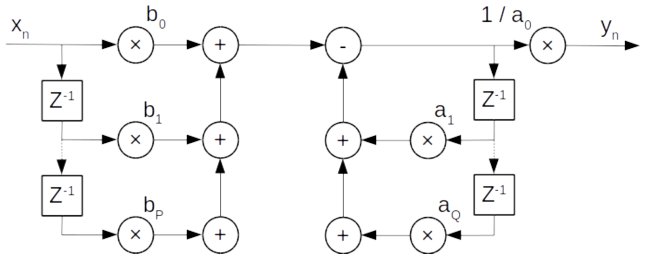

Figure 2. A direct form discrete-time IIR filter of order P and Q

This description, besides the subtraction and division of signals, is quite similar to the previous FIR filter when deconstructed. This similarity seems to imply that they share computational components. As it turns out, the previously defined helper functions can indeed be reused to implement the IIR filter as well:

```
iir :: ( TupleExp exp
       , Num (exp Float)
       , Fractional (exp Float))
       ⇒ [exp Float]
       → [exp Float]
       → Signal exp Float
       → Signal exp Float
iir (a:as) bs s = repeat (1 / a) * (l - r)
  where
    l = sums $ muls bs $ delays (inits bs) s
    r = sums $ muls as $ delays (inits as) r

    inits = P.flip P.replicate 0 . P.length
```

where the rightmost summation, here called `r`, is defined in terms of itself rather than the input signal. Recursive definitions like this are made possible by the lazy nature of the delay operator. The general idea is that any recursion expressed using the signal library introduces feedback, while recursion introduced by pure Haskell code produces repetitive code instead.

3. Implementation

Signals are implemented on top of the following `Stream` data type:

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

The `Prog exp` monad is a deep embedding of an imperative programming language that uses `exp` to represent pure expressions. In the above definition, the outer monad is used to initialize the stream, and the inner action is used to retrieve the next value of the stream. For example, the `head` function, which retrieves the first element of a stream is defined as follows:

```
head :: Stream exp a → Prog exp a
head (Stream init) = do
    next ← init
    next
```

The first line in the `do` block initializes the `next` action, and the second line uses `next` to get the first element.

Our model of streams is essentially the same as in Feldspar's `Stream` library, except that the `Prog exp` monad used here is a standalone monad that adds imperative programming on top of any pure expression EDSL.

The problem with `Stream` is that is quite low-level and cumbersome to program with. For example, in order to define a filter that refers to previous values of some signal, one has to manually create a mutable buffer and update it on every iteration. Feldspar exports a few combinators that hide the details of creating such networks, but these combinators are rather ad hoc, and can only handle a few predefined cases.

Our `Signal` type can be seen as a front end to `Stream` that offers a much more convenient interface. In the end, functions on signals are compiled to functions on streams, as seen in the type of the compile function:

```
compile
  :: (Typeable a, Typeable b)
  ⇒ (Signal exp a → Signal exp b)
  → IO (Stream exp a → Stream exp b)
```

The `IO` in the result type comes from the `data-reify` package that performs observable sharing [10].

The basic way to create nodes in a signal network is by lifting a stream function to a signal function:

```
lift :: (Stream exp a → Stream exp b)
        → Signal exp a
        → Signal exp b
```

We can, for example, use `lift` to define `map` from Section 2:

```
map f = lift (Stream.map f)
```

Lifting is however not enough to define generators, as those are supposed to create signals from nothing. Another signal construct is therefore introduced, modeling the signals identity morphism:

```
bot :: Signal exp a
```

which allows us to define `repeat` as:

```
repeat e = lift (const $ Stream.repeat e) bot
```

Here, `Stream.map` and `Stream.repeat` are the corresponding functions defined for streams.

In this extended abstract, we will not show the definition of `Signal`, `lift` and `compile`, but here is an outline of how it all works:

- `Signal` is a tree type, whose nodes consist of lifted stream functions, `delay` and `sample`. Observable sharing is used to turn this tree into a DAG.
- The compiler assigns a mutable reference (supported by the imperative `Prog` monad) for each node in the graph and creates a program that executes all nodes in sequence, reading and writing data to the corresponding references.
- By analyzing the graph, certain references can be eliminated, and chains of `delay` nodes can be turned into efficient cyclic buffers.

While the stream type is kept abstract in signals, basing them on the co-iterative approach [5] allows us to ensure that no unused computations are performed. Co-iteration is a concept for reasoning about infinite streams and allows one to handle such streams in a strict manner. This strictness in turn enables stream transformers to pick and remove parts of streams as they please – ensuring that no unsued part of a stream is ever computed. This property is kept for signals, as lifted nodes are fused during compilation.

Furthermore, signals offer optimization for a common concept in DSP: *feedback networks*. As the recursively defined streams in feedback networks may reference a number of previous values, memory efficient buffers are introduced for storing the delayed values in these signals. These buffers have memory proportional in size to the number of delays and are used to minimize the number of read/write operations used during execution. Detecting such feedback in signal networks is made possible by using observable sharing [7], which allows us to reify signal networks into graphs were the back-edges between nodes are visible.

4. Evaluation

In order to evaluate the signal library we will look at both its expressiveness and the code it generates. As the actual library is being finished at the time of writing, we delay most of the evaluation until the final report and only include a comparison between the generated code of an example and its hand-written counterpart. Note, too, that the compiler has some notable limitations: it currently only handles pure signal functions, that is, we can only compile types of `Signal exp a → Signal exp b`. Also, while signals support the notion of buffers, the compiler does not. The following example will therefore not make use of circular arrays, but will do so in the final version.

Disregarding the current state of our compiler, the produced code has room for several potential improvements. Firstly, too many references are used during the compilation process, which then spills into the compiled code and produces a number of unused variables. There are also some leftover pairs found in the produced code, remnants from the zipping constructor. Both the numerous variables and the pairs can however be optimized away, and should not be present in the final report. There is also room for more subtle improvements; for example, a dedicated initialization function could remove the need for some if-statements. Another interesting improvement is to make use of C specific memory management functions, such as `memset` or `memcopy`, for cases when the data is neatly ordered – as it is in the case of the FIR filter, for example.

For the actual comparison, we generated code from the previous FIR filter and compared it to a hand-written version. However, as the compiler only accepts signal functions, we feed the FIR filter a one-element list of ones before compiling it – effectively making it a rank-1 FIR filter. This produces the following code:

```
typedef struct {
  float first;
  float second;
} pair;

int main()
{
  FILE *v0;
  v0 = fopen("test", "r");
  FILE *v1;
  v1 = fopen("test2", "w");

  int i = 0;
  while (i < 3)
  {
    float v3;
    float v4;
    fscanf(v0,"%f",v4);
    float v5 = v4;
    float v6;
    pair v7;
    float v8;
    float v9;
    bool v10 = true;
    float v11 = v5;
    bool v12;
    if (v12 = v10)
    {
      v10 = false;
      v8  = 0.0;
    }
    else
    {
      float v13 = v9;
      v8 = v13;
    }
    v9 = v11;
    float v14;
    v14 = 1.0;
    float v15 = v14;
    float v16 = v8;
    v7.first = v15;
    v7.second = v16;
    pair v17 = v7;
    v6 = v17.first * v17.second;
    float v18 = v6;
    v3 = v18;
    float v19 = v3;
    fprintf(v1, "%f", v19);
    i++;
  }

  return 0;
}
```

While the hand-written code is for a more general FIR filter, where the number of coefficients hasn't been fixed yet, the comparison should however still be valid as the underlying ideas have not changed.

```
double insamp[ ... ];

void firInit( void )
{
    memset( insamp, 0, sizeof( insamp ) );
}

void fir( double *coeffs,
       double *input, double *output,
       int length, int filterLength )
{
    double acc;     // accumulator for MACs
    double *coeffp; // pointer to coefficients
    double *inputp; // pointer to input samples
    int n;
    int k;

    // put the new samples at the high
    // end of the buffer
    memcpy( &insamp[filterLength - 1], input,
            length * sizeof(double) );

    // apply the filter to each input sample
    for ( n = 0; n < length; n++ ) {
        // calculate output n
        coeffp = coeffs;
        inputp = &insamp[filterLength - 1 + n];
        acc = 0;
        for ( k = 0; k < filterLength; k++ ) {
            acc += (*coeffp++) * (*inputp--);
        }
        output[n] = acc;
    }

    // shift input samples back in time
    // for next time
    memmove( &insamp[0], &insamp[length],
            (filterLength - 1) * sizeof(double) );

}
```

While benchmarking would clearly be of interest here, we delay it to the final paper.

5. Related Work

Lava is a family of simple hardware description languages embedded in Haskell [6, 11]. What look like circuit descriptions are actually circuit generators that are run to produce internal representations, which can be used to produce further useful artifacts such as netlists or formulas for use in formal verification. Later versions of Lava have used observable sharing to provide a more user-friendly approach to capturing feedback in these circuit descriptions. There is a close link between this approach to hardware description and SDF languages like LUSTRE. One way to view the present work is as a beefed up Lava in which the building blocks are general data-processing functions rather than just Boolean gates.

Lucid Synchrone is another functional language for SDF and is hosted in OCaml, importing every ground type from the host language and lifting them to corresponding stream versions [8, 20]. Sequential operations over the imported stream are also offered, similar to those from our signal library. Lucid Syncrhone incorporates several type systems (including clock inference) that guarantee safety properties of the generated code. In addition, special syntax for defining automata is provided. Our work is inspired by Lucid Syncrhone and we will investigate the inclusion of these features in our signal library.

Functional reactive programming is another common paradigm for modeling continuous signals, and its libraries are typically implemented using Haskell's arrow or applicative classes, see for example Yampa [9, 12], reactive-banana [2] or Sodium [4]. Although we cannot support the promotion of abstract functions required by these otherwise attractive interfaces, we have still drawn inspiration from the lifting functions of FRP.

The use of pure Haskell for modeling signals [22] has also been investigated, demonstrating that functional programming and lazy evaluation can directly model common signal problems quite satisfactorily. Other related work includes Atom [14], Ivory and Tower [15], but we delay the discussion of these to the final paper.

6. Discussion

Functional programming encourages a style of programming in which combinators or higher order functions capture common patterns of computation. It is when we combine SDF with a sufficiently general value type that the question of how to design an appropriate set of combinators becomes an interesting one. For instance, the ability to pass arrays, or any similar data type, as values over signals means that the programmer is concerned with processing chunks of data rather than just individual values. This change of perspective is necessary if we are to implement key functions like FFT on signals. This generality does however come at a cost, as ill-defined signals could be expressed and type-checked due to a lack of constraints on sequential signals

The benefits of a general lifting constructor come into full effect when the underlying expression language has powerful features of its own. In the case of Feldspar, an expression language which already supports a plethora of different algorithms, several complex signal functions can often be obtained by simply lifting the regular ones. For instance, as Feldspar already contains an FFT algorithm over vectors, a signal version can be obtained by simply lifting the existing one.

The simplicity of the current signal library does mean that some ill-defined signals can be expressed. Sequential operations in particular are quite susceptible to grammatical errors, as one can easily create signals with an undefined behavior when using delay/sample. For instance, consider the following function:

```
f :: Signal exp a → Signal exp a
f sig = sample 2 sig + sig
```

The clocks of these two streams are obviously not equal, but there are at present no constraints in place to keep such signals for being defined. Recursively defined signals suffer from a similar problem: there is no constraint in place to check whether the recursive signal has been delayed or not before it is used. Signals with undefined initial values can therefore be expressed by, for example, writing:

```
g :: Num (exp a) ⇒ Signal exp a
g = o where o = map (+1) o
```

Such ill-defined signals could however be identified during the compilation process, by inspecting the signal's reified syntax tree.

Our plan is to develop this library futher, based on ideas from Lucid Synchrone and FRP.

Acknowledgments

This research was funded by the Swedish Foundation for Strategic Research (in the RAW FP project) and by the Swedish Research Agency (Vetenskapsrådet).

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. Reactive-banana. *Haskell library available at http://www. haskell. org/haskellwiki/Reactive-banana*, 2012.

[3] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, 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. .

[4] S. Blackheath. Sodium reactive programming (frp) system, March 2014. URL https://hackage.haskell.org/package/sodium.

[5] 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. {CMCS} '98, First Workshop on Coalgebraic Methods in Computer Science.

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

[7] 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.

[8] 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, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. . URL http://doi.acm.org/10.1145/1017753.1017792.

[9] 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, New York, NY, USA, 2003. ACM. ISBN 1-58113-758-3. . URL http://doi.acm.org/10.1145/871895.871897.

[10] A. Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. . URL http://doi.acm.org/10.1145/1596638.1596653.

[11] 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.

[12] G. Giorgidze and H. Nilsson. Switched-on yampa. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative*

Languages, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-77441-9. . URL http://dx.doi.org/10.1007/978-3-540-77442-6_19.

[13] N. Halbwachs. *Synchronous programming of reactive systems.* Number 215. Springer, 1992.

[14] T. Hawkins. Controlling hybrid vehicles with haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming.* ACM, 2008.

[15] 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*, pages 3–9. ACM, 2014.

[16] 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.

[17] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/1013208.1013209.

[18] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987. ISSN 0018-9340. .

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

[20] M. Pouzet. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

[21] T. B. Sousa. Dataflow programming: Concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, 2012.

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