Modelling Asynchronous Streams in Haskell*

H. John Reekie†
School of Electrical Engineering
University of Technology, Sydney

email: johnr@ee.uts.edu.au

Abstract

This paper explores the use of Haskell to model discrete-time signals, or streams, with an emphasis on application to signal processing. Because of the deterministic nature of functional programs, asynchronous streams must be modelled by explicitly including time. Two types of asynchronous stream are examined: hiatonic streams, and time-stamped streams. A number of utility functions for manipulating asynchronous streams are presented, and their use demonstrated with examples.

1 Introduction

Lazy functional languages such as Haskell [4] support infinite data structures. Infinite sequences are the most commonly-used of such structures, and are a natural choice for modelling discrete-time signals. These are often called "streams." With streams, it is straight-forward to construct networks of processes [7, 12]. These networks can also be viewed as large-grain dataflow graphs [8], which is more in tune with the viewpoint of signal processing computation. Although lazy lists are often used to model streams, a distinct *Stream* datatype can also be used to more clearly indicate the intended use of the data structure. In this paper, however, I use lazy lists.

If streams are used to model discrete-time signals, the question arises of what the temporal meaning of these streams is—in other words, is the arrival time, or time of production, of stream elements significant, and in what way? There is no inherent notion of time in pure functional languages—indeed, their development is marked by the desire to remove the notion of time from computation.

To illustrate: a problem that occurs in proposed functional operating systems is that of merging two streams [10]. In this context, streams represent channels of communication between processes within the system. If two streams are merged in order of element arrival time, the same two input sequences will not always produce the same output sequence, because the arrival time of elements is in general non-deterministic. This violates the principle of referential transparency, and therefore cannot be modelled directly in a lazy functional language. The solution adopted is to move the non-determinism outside the language.

*Published as: Research Report 94.3, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, PO Box 123, Broadway NSW 2007, Australia, May 1994.

†This work was supported by an Australian Postgraduate Award.

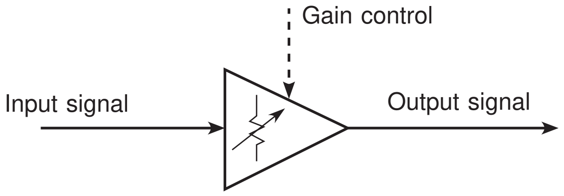

Figure 1: Digital gain control

In other applications, however, it is easy to remove the non-determinism associated with the "arrival time" of stream elements. If, for example, a stream represents a uniformly-sampled, discrete-time signal, then elements of the stream are taken to occur at precisely-spaced instants. On this basis, it is simple to construct process networks that model real-time signal processing computations [9].

Although many signal processing applications can be constructed solely with uniformly-sampled signals, it is useful also to be able to model non-uniformly sampled signals. Streams representing signals of this type are called *asynchronous* streams. For signal processing, it is necessary to be able to specify the timing of asynchronous streams relative to synchronous streams. For example, consider the "digital gain control" shown in Figure 1. The gain control signal has a value only when a person changes the position of a gain control knob. In order to produce an appropriate output signal, we must be able represent the times of occurrence of gain control values relative to the input signal.

The remainder of this paper explores the representation and use of asynchronous streams in Haskell. Because of the deterministic nature of functional programs, time information is explicitly encoded into these streams. After describing two types of asynchronous stream, I give a number of utility functions on them. A number of examples illustrate the use of these functions, and I conclude with some thoughts on implementation in a real-time context.

1.1 A note on notation

The code in this paper is executable by the Gofer [5] interpreter. Gofer is very similar to Haskell, and the code will require only one or two minor changes to be compiled by a Haskell compiler. The source of this paper is a literate Gofer script—that is, it can be executed by the Gofer interpreter as-is. Lines beginning with ">" are program code.

2 Stream Basics

In this paper, a stream is represented by a lazy list, written e.g. $[1, 2, 3, 4, 5, 6, 7]$. (Although this is a finite list, lists can be infinite.) A list containing elements of type a has the type $[a]$.

Functions on lists are best described by example. Haskell's *map* function applies a given function to every element of its list argument—it is the same as Lisp's *mapcar* and Smalltalk's *collect*:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
```

```
map f (x:xs) = f x : map f xs
```

The first line is a type declaration. It states that, for some types a and b: the first argument is a function from a to b; the second argument is a list of a; and the result is a list of b. This function is defined recursively: the base case (an empty list) is handled by the first clause; the recursive case by the second. Pattern-matching is used to bind the head and tail of the list to the variables x and xs.

map is applied to a list as follows:

$$map\ (+1)\ [1,3,4,7]\ \Rightarrow\ [2,4,5,8]$$

where $\Rightarrow$ means "evaluates to."

The expression *map* $(+1)$ thus represents a process that adds one to each input element. We may choose to define it as a named function:

```
> incproc :: (Num a) => [a] -> [a]
> incproc = map (+1)
```

The type constraint $(Num\ a)$ specifies that a must be a type in the hierarchy of numerical types. See [4] for details.

To construct "point-wise" processes on corresponding elements of two streams, we can use Haskell's *zipWith* function:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs)   = f a b : zipWith f as bs
zipWith _ _      _        = []
```

The underscores indicate formals that match any actual. An example of the use of *zipWith* is:

$$zipWith\ (+)\ [1,3,4,7]\ [1,2,3,4]\ \Rightarrow\ [2,5,7,11]$$

A fourth useful function is called *prefl*[1], for "prefix-from-left."

```
> prefl :: (a -> b -> a) -> a -> [b] -> [a]
> prefl f a []     = []
> prefl f a (x:xs) = q : prefl f q xs where q = f a x
```

Like *map*, *prefl* produces one output value for each input value. However, the function argument f is applied to an input value and the previous output value, instead of just to the input value. For example,

$$prefl\ (+)\ 0\ [1,3,4,7]\ \Rightarrow\ [1,4,8,15]$$

A third function, *filter*, removes items of a list that do not satisfy a predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

For example,

$$filter\ odd\ [1,3,4,7]\ \Rightarrow\ [1,3,7]$$

An introduction to functional programming can be found in [3] or [5].

[1] *prefl* is very similar to Haskell's *scanl*: $prefl\ f\ a\ =\ tail\ \cdot\ scanl\ f\ a$. I use this definition because *scanl* effectively inserts an unwanted delay in a signal.

3 Synchronous Streams

As noted in the Introduction, streams are easily used to represent uniformly-sampled discrete-time signals. Systems containing only uniformly-sampled discrete-time signals are also known as synchronous dataflow (SDF) systems in the literature [8], so I will call these synchronous streams. Non-uniformly-sampled signals will therefore be called *asynchronous* streams. Streams with different sample rates can exist within the same system, provided that the relative rates of streams are consistent with the rates expected by processes that produce and consume them. For example, connecting a stream of rate n and a stream of rate $2n$ to a point-wise addition process results in inconsistent rates—eventually, the system will run out of buffer space.

In a real-world signal processing system, the actual time of occurrence of an element of a stream is not the exact time associated with it as a sample of a discrete-time signal. Processing and communications time, buffering, and finite resources make this impossible. Instead, it is only required that certain hard realtime processes—such as analog-to-digital (A/D) and digital-to-analog (A/D) converters—perform regularly-clocked, realtime processing; execution of other processes can take place in any manner that satisfies the input and output needs of these processes.

When we use functional programs to model a signal processing system, we do not explicitly mark streams as synchronous or asynchronous. A stream becomes synchronous only because it is connected—directly or indirectly—to a synchronous realtime process. If, for example, the input of an *incproc* process is connected to the output of an A/D convertor process, then the output stream of the *incproc* process is also synchronous, because *incproc* produces one output item for every input item.

Processes that do not have fixed ratios between input and output data rates break the synchronisation between parts of the system. For example, if a process such as $filter\ odd$ has its input connected to the output of an A/D converter process, then the output stream from $filter\ odd$ is not synchronous.

The propagation of synchrony is rather like type inference. If the sample rate of a stream is considered to be part of its type, then connection of two streams with different rates to say, a point-wise addition process, is a type error. Note that this does not exclude different sample rates within the one system, since the data rates of process inputs and outputs can be in any fixed ratio to each other. It would be interesting to attempt to extend the Haskell type-checker to infer stream sample rates.

There is little that can usefully be said about an asynchronous stream produced by a process such as $filter\ odd$. Although we can still operate on such streams, we have lost all timing information. If, for example, we applied a standard finite-impulse-response (FIR) filter[2] to the output of $filter\ odd$, we would certainly get a signal, but it would be meaningless as a discrete-time data signal, since the filter cannot account for the non-uniform timing of the stream elements. The actual time at which each element is received by the filter cannot be used—as we have seen, this time is only loosely coupled to the "correct" real time.

[2] The terminology from signal processing and functional programming clashes here. In this case, I mean a filter that changes the frequency response of a signal, but maintains its sample rate.

4 Asynchronous Streams

To be able to usefully manipulate asynchronous streams, we must encode timing information into the stream itself. I use two methods for doing so: hiatonic streams, and time-stamped streams.

4.1 Hiatonic Streams

A hiatonic stream represents a uniformly-sampled signal with "non-values"—that is, time instants that do not carry a meaningful value. These instants are called *hiatons*, a term suggested in [11]. The domain of any type to be carried by a hiatonic stream must be extended to include the special hiaton value. In Haskell, we cannot just extend, say, the domain of integers; however, we can generate a new algebraic type that includes both integers (or whatever) and hiatons:

```
> data Slot a = Hiaton | Val a
```

That is, a value of type *Slot a* can be either a hiaton or a value of type a. A hiatonic stream is a list of these "slots," type [*Slot a*]. An example of a hiatonic stream is:

```
> hiatonic = [ Val 1, Hiaton, Val 3, Val 4,
>              Hiaton, Hiaton, Val 7, Hiaton ] :: [Slot Int]
```

for which I use the short-hand representation

$$[1, *, 3, 4, *, *, 7, *]_*.$$

For implementation in a real-time context, it will be necessary to "hide" the hiatons from the programmer, and provide the programmer access to hiatonic streams only through the functions of an abstract data type (ADT). This will ensure that a compiler is able to use the most efficient mechanism available for implementing them.

Firstly, we will consider equivalents of the *map* and *prefl* functions. To apply a *map* operation on a hiatonic stream, the argument function must be applied only to the values, and the hiatons left alone. The *mapH* function is defined as follows:

```
> mapH :: (a -> b) -> [Slot a] -> [Slot b]
> mapH f []           = []
> mapH f (Hiaton:xs)  = Hiaton : mapH f xs
> mapH f ((Val x):xs) = Val (f x) : mapH f xs
```

For example,

$$mapH\ (+1)\ [1, *, 3, 4, *, *, 7, *]_* \ \Rightarrow\ [2, *, 4, 5, *, *, 8, *]_*$$

The asynchronous equivalent of *prefl* also produces an output value only for each input value:

```
> preflH :: (a -> b -> a) -> a -> [Slot b] -> [Slot a]
> preflH f a []           = []
> preflH f a (Hiaton : xs) = Hiaton : preflH f a xs
> preflH f a (Val x : xs)  = Val y  : preflH f y xs where y = f a x
```

For example,

$$preflH\ (+)\ 0\ [1, *, 3, 4, *, *, 7, *]_* \Rightarrow [1, *, 4, 8, *, *, 15, *]_*$$

4.2 Time-stamped Streams

In a hiatonic stream, non-value time slots are marked with a special value. In effect, hiatonic streams are still synchronous. An alternative representation pairs each value with a "time-stamp," which denotes the time at which the value occurs.

A time-stamped stream has an associated *clock*—the clock is the sample rate the signal would have if it were represented by a hiatonic stream. The time stamp associated with each value, $t >= 1$, is the number of clock periods, or "ticks," since the previous value occured. Note that only one value can occur in each tick.

The datatype used to represent a value and its time-stamp is:

```
> data Stamp a = Stamp Int a
```

That is, the type *Stamp a* contains a value of type a together with an integer time-stamp. A stamped stream is a list of stamped values, type [*Stamp a*]. Here is the time-stamped version of the hiatonic stream given earlier:

```
> stamped = [ Stamp 1 1, Stamp 2 3,
>             Stamp 1 4, Stamp 3 7] :: [Stamp Int]
```

for which I use the short-hand representation

$$[1_1, 3_2, 4_1, 7_3].$$

As for hiatonic streams, versions of the *map* and *prefl* functions for time-stamped streams operate on the value parts of the stream, and leave the time parts unchanged:

```
> mapT :: (a -> b) -> [Stamp a] -> [Stamp b]
> mapT f []                   = []
> mapT f (Stamp t x : xs)     = Stamp t (f x) : mapT f xs

> preflT :: (a -> b -> a) -> a -> [Stamp b] -> [Stamp a]
> preflT f a []               = []
> preflT f a (Stamp t x : xs) = Stamp t y : preflT f y xs
>     where
>     y = f a x
```

For example,

$$mapT\ (+1)\ [1_1, 3_2, 4_1, 7_3] \Rightarrow [2_1, 4_2, 5_1, 8_3]$$

$$preflT\ (+)\ 0\ [1_1, 3_2, 4_1, 7_3] \Rightarrow [1_1, 4_2, 8_1, 15_3]$$

We can also define a point-wise operator on time-stamped streams, $zipWithT$. We will assume that this function produces a defined result only if values occur on both input streams at the same time instants:

$$zipWithT\ (*)\ [1_1, 3_2, 4_1, 7_3]\ [1_1, 3_2, 4_1, 7_3] \Rightarrow\ [1_1, 9_2, 16_1, 49_3]$$

5 Utility Functions

This section contains a number of functions that operate on asynchronous streams. Although there are enough here to present some interesting examples (In section 6), the set is not meant to be exhaustive. For the sake of brevity, most function definitions are left to Appendix B.

5.1 Selecting

The standard Haskell functions *take* and *drop* return and remove the first k elements of a list, where k is the first argument. These functions can be applied as-is to hiatonic streams. For stamped streams, we need functions that operate on the time component of the stream, rather than on the number of values.

dropT k removes elements of a stream that occur before k ticks, and adjusts the first time-stamp of the remaining stream:

$$dropT\ 5\ [1_1, 3_2, 4_1, 7_3]\ \Rightarrow\ [7_2]$$

takeT k returns all elements of the stream that occur within k ticks. It can be used to convert a potentially-infinite stream into a finite one:

$$takeT\ 5\ [1_1, 3_2, 4_1, 7_3]\ \Rightarrow\ [1_1, 3_2, 4_1]$$

The standard Haskell function *takeUntil* takes all elements from the front of a list up to and including the first element that satisfies a predicate. *takeUntilT* is the time-stamped stream version:

$$takeUntilT\ even\ [1_1, 3_2, 4_1, 7_3]\ \Rightarrow\ [1_1, 3_2, 4_1]$$

delayT delays a signal by a specified number of ticks:

$$delayT\ 3\ [1_1, 3_2, 4_1, 7_3]\ \Rightarrow\ [1_4, 3_2, 4_1, 7_3]$$

5.2 Filtering

filter, described earlier, removes unwanted elements from a stream but does not preserve time information. Versions of *filter* that do preserve time information could either a) take a synchronous stream and produce an asynchronous stream, or b) take an asynchronous stream and produce another asynchronous stream. *selectH* and *selectT* produce an asynchronous stream from a synchronous stream:

$$selectH\ odd\ [1, 2, 3, 4, 5]\ \Rightarrow\ [1, *, 3, *, 5]_*$$

$$selectT\ odd\ [1, 2, 3, 4, 5]\ \Rightarrow\ [1_1, 3_2, 5_2]$$

We can also define functions that remove unwanted elements from asynchronous streams (see *filterH* and *filterT* in Appendix B).

5.3 Sampling

A synchronous stream can be "sampled" at time instants specified by a stamped stream. The $whenT$ function does exactly this:[3]

$$[5,4,3,2,1]\,`whenT`\,[1_1,3_2,4_1,7_3] \;\Rightarrow\; [5_1,3_2,2_1]$$

5.4 Resynchronising

To convert a time-stamped stream back into a synchronous stream, we must "fill in" the time slots between values. The function $resyncT$ takes two function arguments. At a tick in which a value occurs, the first function is applied to that value and the previous output value, to generate the new output value. At a tick in which no value occurs, the second function is applied to the previous output value, to produce the new output value:

```
> resyncT :: (b -> a -> a) -> (a -> a) -> a -> [Stamp b] -> [a]
> resyncT f g a [] = []
> resyncT f g a (Stamp 1 x : xs )
>                    = y : resyncT f g y xs where y = f x a
> resyncT f g a (Stamp t x : xs )
>                    = y : resyncT f g y (Stamp (t-1) x : xs) where y = g a
```

For example,

$$resyncT\ (+)\ (+1)\ 0\ [1_1,3_2,4_1,7_3] \;\Rightarrow\; [1,2,5,9,10,11,18]$$

When we convert an asynchronous stream back into a synchronous stream, we must fill in the empty time slots. There are two choices for the missing values: i) a specified value, or ii) the most recent value. The time-stamped versions of the functions below use the $resyncT$ function.

$fillH$ and $fillT$ insert a specified value:

$$fillH\ 0\ [1,*,3,4,*,*,7,*]_* \;\Rightarrow\; [1,0,3,4,0,0,7,0]$$

$$fillT\ 0\ [1_1,3_2,4_1,7_3] \;\Rightarrow\; [1,0,3,4,0,0,7]$$

$holdH$ and $holdT$ perform a zero-order hold. The first argument specifies the value inserted into initial empty time slots:

$$holdH\ 0\ [1,*,3,4,*,*,7,*]_* \;\Rightarrow\; [1,1,3,4,4,4,7,7]$$

$$holdT\ 0\ [1_1,3_2,4_1,7_3] \;\Rightarrow\; [1,1,3,4,4,4,7]$$

5.5 Desynchronising

If we longer need the timing information in a stream, we can discard it. $desyncT$ and $desyncH$ discard timing information, generating a normal list:[4]

$$desyncH\ [1,*,3,4,*,*,7,*]_* \;\Rightarrow\; [1,3,4,7]$$

$$desyncT\ [1_1,3_2,4_1,7_3] \;\Rightarrow\; [1,3,4,7]$$

[3] The backquotes surrounding `$whenT$` are Haskell's notation for infix operators: $f\ x\ y\ =\ x\,`f`\,y$.

[4] We could perhaps view this as producing a *true* asynchronous stream.

5.6 Converting

stamp converts a hiatonic stream to a stamped stream, while *unstamp* does the reverse:

$$stamp\ [1, *, 3, 4, *, *, 7, *]_* \;\Rightarrow\; [1_1, 3_2, 4_1, 7_3]$$

$$unstamp\ [1_1, 3_2, 4_1, 7_3] \;\Rightarrow\; [1, *, 3, 4, *, *, 7]_*$$

Notice that these functions are not quite inverses of each other, since

$$stamp \cdot unstamp \;=\; id$$

but

$$unstamp \cdot stamp \;\neq\; id.$$

This raises an interesting question. The lists $[1, *, 3, 4, *, *, 7, *]_*$ and $[1, *, 3, 4, *, *, 7]_*$ are clearly not the same (the former has an extra empty time slot at the end). But do they represent the same stream? If it is not meaningful to have non-value time slots at the end of a stream, then these two lists should be taken to mean the same stream. If a *Stream* datatype were provided, it would be possible to ensure that a user program could not distinguish between these two streams.

If a stream is taken to represent an *infinite* discrete-time signal, then there is no need to distinguish between the two. The restriction to infinite streams only is not an unreasonable one in many application domains. For example, the Gabriel programming system [2] supports only infinite streams. Even if streams are finite (as in the nested streams in section 5.8), I can think of no reason why empty time slots at the end of a stream should be significant.

5.7 Working on Time

We sometimes need to know the *time* at which values in a time-stamped stream occur. *deltaT* extracts the time stamps and places them into the stream values:

$$deltaT\ 0\ [1_1, 3_2, 4_1, 7_3] \;\Rightarrow\; [1_1, 2_2, 1_1, 3_3]$$

It may be necessary to change the base data rate of an asynchronous stream—in other words, to up-sample or down-sample the stream. Up-sampling is easy, since it is just a matter of inserting extra hiatons or scaling the time-stamps:

$$upsampleH\ 2\,[1, *, 3, 4, *, *, 7, *]_* \;\Rightarrow\; [1, *, *, *, 3, *, 4, *, *, *, *, *, 7, *, *, *]$$

$$upsampleT\ [1_1, 3_2, 4_1, 7_3] \;\Rightarrow\; [1_1, 3_4, 4_2, 7_6]$$

Down-sampling is slightly more involved, because more than one input item can map into the same output time slot. It is treated in section 6.

5.8 Nested Streams

Nested streams are useful for expressing dynamically changing process networks. Suppose, for example, that each value of a time-stamped stream is to call a new process into being. For simplicity, let us suppose that the value is a pair (v, k), indicating that the new process is to produce a synchronous stream containing k data values with value v. We can express this as:

```
> nested :: [Stamp (a,Int)] -> [Stamp [a]]
> nested = mapT f where f (v,k) = copy k v
```

where the standard Haskell function *copy* generates k copies of v.

This process effectively spawns a new process each time it receives a value. For example,

$$\textit{nested}\ [(3,4)_1, (7,4)_2] \;\Rightarrow\; [[3,3,3,3]_1, [7,7,7,7]_2]$$

Since each stream within the (time-stamped) output stream is synchronous, their values overlap in time as follows:

```
Time slot   0   1   2   3   4   5
            ---------------------
Values      3   3   3   3
                    7   7   7   7
```

One way of making use of these dynamically-created streams is to combine them again in some way. The function *combineT* accepts a function argument; at each tick, this function is applied to a list containing the front elements of all synchronous streams active at that time. For example,[5]

$$\textit{combineT}\ \textit{sum}'\ [[3,3,3,3]_1, [7,7,7,7]_2] \;\Rightarrow\; [3,3,10,10,7,7]$$

nested used *mapT* to generate a nested stream. Sometimes, however, it is necessary to know both the value at that time, and the rest of the input stream. The *mapContextT* function provides this:

```
> mapContextT :: (a -> [Stamp a] -> b) -> [Stamp a] -> [Stamp b]
> mapContextT f []               = []
> mapContextT f (Stamp t x : xs) = Stamp t (f x (Stamp 1 x : xs))
>                                   : mapContextT f xs
```

mapContext is used in section 6.7.

[5] *sum'* (Appendix C) is an overloaded list sum function—the standard Gofer version of *sum* is not overloaded.

6 Examples

In this section, I develop some simple examples that illustrate how the functions defined in the preceding sections can be used. Note that none of the functions defined here makes direct access to hiatons or time-stamps.

6.1 Detecting change

One use for asynchronous streams may be to reduce the data rate of a signal. The *edges* function can be used for this—it outputs the input value whenever it changes. Its first argument is the starting value with which it compares the first value in the stream.

```
> edges :: (Eq a) => a -> [a] -> [Slot a]
> edges  a = mapH snd . selectH neq . pairs a where
>            neq (x,y) = x /= y
```

For example,

$$edges\ 0\ [1,1,3,4,4,4,7,7]\ \Rightarrow\ [1,*,3,4,*,*,7,*]_*$$

The *pairs* function used above combines adjacent elements of a stream into pairs (see Appendix C):

$$pairs\ 0\ [1,1,3,4,4,4,7,7]\ \Rightarrow\ [(0,1),(1,1),(1,3),(3,4),(4,4),(4,4),(4,7),(7,7)]$$

6.2 Detecting zero crossings

Detecting the zero-crossings of a signal is a simple but useful operation. The *zero_crossings* function produces a stamped stream, each value of which is the time since the previous zero-crossing. It works by taking the sign of each input element and detecting changes in the sign.

```
> zero_crossings :: (Ord a, Num a) => [a] -> [Stamp Int]
> zero_crossings = deltaT . stamp . edges True . map (>= 0)
```

For example,

$$zero\_crossings\ [0,1,2,-1,-1,-2,1,-1]\ \Rightarrow\ [3_3,3_3,1_1]$$

6.3 Differential PCM

Suppose we wish to encode and transmit a signal as straight-line segments. The encoding produces as its first value the first value of the signal, as its second the slope of the first segment, and thereafter, a value whenever the slope of the signal changes:

```
> encode :: (Num a) => [a] -> [Stamp a]
> encode = stamp . edges huge . map diff . pairs 0
>     where
>     diff (x,y) = y - x
>     huge = -100000000
```

where *huge* can be any value that cannot appear in the input stream.

For example,

$$encode\ [0,1,2,2,2,0,-2,-3,-3,-3,-3] \ \Rightarrow\ [0_1,1_1,0_2,-2_2,-1_2,0_1]$$

The signal can be re-assembled by doing a zero-order interpolation and then integrating:

```
> decode :: (Num a) => [Stamp a] -> [a]
> decode = prefl (+) 0 . holdT 0
```

For example,

$$decode\ [0_1,1_1,0_2,-2_2,-1_2,0_1] \ \Rightarrow\ [0,1,2,2,2,0,-2,-3,-3]$$

Notice that the final segment has been thrown away, because there is no end marker produced when the differential stream is time-stamped.

6.4 Down-sampling

Earlier, I postponed the definitions of functions to down-sample asynchronous streams. More than one input value can map to a single output value, so to deal with this, the down-sampling functions require an additional function argument. This function must accept a (non-empty) list of values and return a single value.

```
> downsampleH :: Int -> ([a] -> a) -> [Slot a] -> [Slot a]
> downsampleH k choose =
>     mapH choose . selectH (not . null) . map desyncH . group k
```

(The *group* function is defined in Appendix C.)

The stamped stream version is defined in terms of the hiatonic stream function; this implies a large ineffiency if the items occur infrequently:

```
> downsampleT :: Int -> ([a] -> a) -> [Stamp a] -> [Stamp a]
> downsampleT k choose = stamp . downsampleH k choose . unstamp
```

For example,

$$downsampleH\ 2\ head\ [1,*,3,4,*,*,7,*]_* \ \Rightarrow\ [1,3,*,7]_*$$

$$downsampleT\ 2\ head\ [1_1,3_2,4_1,7_3] \ \Rightarrow\ [1_1,3_1,7_2]$$

6.5 Filtering asynchronous signals

One of the reason cited earlier for asynchronous streams was to allow meaningful filtering of non-uniformly sampled signals. To reconstruct a uniformly-sampled signal from a time-stamped signal, we can simply insert zero's and filter. Having done so, we can then down-sample to obtain a signal at the desired rate.

```
> resampleT1 :: (Num a) => ([a] -> [a]) -> Int -> [Stamp a] -> [a]
> resampleT1 filt k = map (!!0) . group k . filt . fillT 0
```

where the first argument $filt$ is some filtering function, and the second k is the down-sampling factor. $group\ k$ splits the input stream into a stream of k-length sequences, and map (!!0) selects the first element of each sequence.

If $filt$ is an FIR (finite-impulse-response) filter, then the zeroes inserted by $fillT$ result in redundant computation. In this case, reconstructing a time-stamped stream at some period k is much the same as a standard FIR interpolating-decimating filter, but with a time-varying input data rate. It can be expressed as follows:

```
> resampleT :: (Num a) => [a] -> Int -> [Stamp a] -> [a]
> resampleT coeffs k input = resampT (delayT (n-1) input) where
>     resampT []    = []
>     resampT input = ipT coeffs input : resampT (dropT k input)
>     n             = length coeffs
```

This works as follows: n is the length of the coefficient vector, and k is the output sampling period (in ticks). Firstly the input signal is delayed to simulate the initial state of the filter ($delayT\ (n - k)\ input$). The auxiliary function $resampT$ uses ipT to calculate the current output value; it then calls itself with the input stream moved along by k ticks ($resampT\ (dropT\ k\ input)$).

Each output value is calculated by taking the inner product of the stream of input data items in the first n ticks and the coefficients sampled by the input data clock. In effect, $resampleT$ creates a network of four processes—$whenT$, $zipWithT$, $desyncT$, and sum'—for each output value it produces. ipT is defined by:

```
> ipT :: (Num a) => [a] -> [Stamp a] -> a
> ipT coeffs input = sum' (desyncT
>                           (zipWithT (*) input (coeffs `whenT` input)))
```

Figure 2 shows a) a signal (plain line); b) the signal obtained by i) sampling this signal at irregular intervals between two and five ticks, and ii) reconstructing the signal with $resampleT$ and $k = 1$ (open diamonds); and c) for comparison, the signal when filtered without the irregular sampling (black diamonds). (Appendix C gives additional code needed to generate this data.)

6.6 Asynchronous user control

The Introduction to this paper gave a simple example of an asynchronous user control over a real-time signal processing function. The "digital gain control" is easily defined by:

```
> digitalGain :: (Num a) => [a] -> [Stamp a] -> [a]
> digitalGain signal control = zipWith (*) signal (holdT 0 control)
```

This simple idea can be used to convert any kind of real-time process into one with dynamic user control.

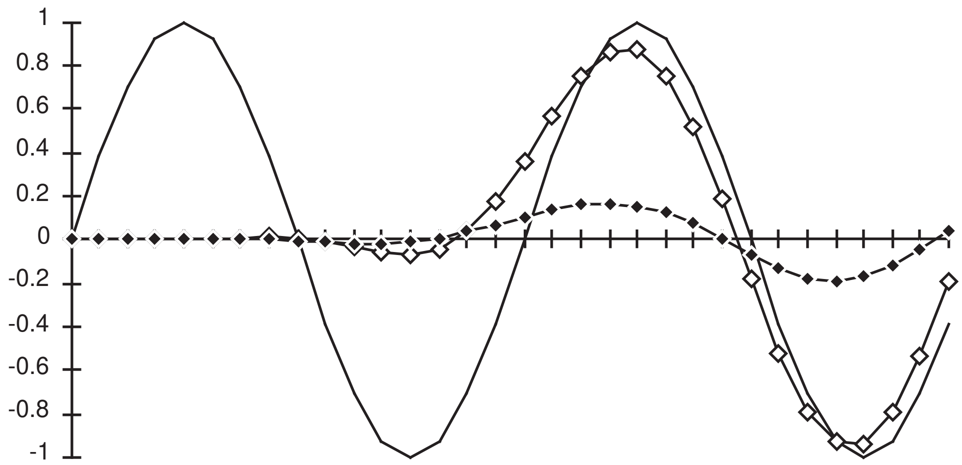

Figure 2: Sample filter input and output

6.7 Music Synthesiser

This example illustrates the use of dynamic process creation to generate sinusoidal tones. It is a simplified version of a more complete musical synthesiser program currently under development.

In this example, the asynchronicity occurs in the timing of the occurrence of notes. The input to the synthesiser is a time-stamped stream of note events:

```
> type Frequency = Float
> type Amplitude = Float
> type NoteID = Int

> data NoteEvent = NoteOn NoteID Frequency Amplitude
>                | NoteOff NoteID

> isNoteOn (NoteOn _ _ _) = True
> isNoteOn (NoteOff _)  = False
```

A note is generated from the time that its note-on event occurs until a corresponding note-off event occurs. An arbitrary number of notes can be active simultaneously. Our sample event stream is

```
> events :: [Stamp NoteEvent]
> events = [ Stamp 5 (NoteOn 7 3000 0.25),
>            Stamp 9 (NoteOn 3 1500 0.4),
>            Stamp 15 (NoteOff 3), Stamp 4 (NoteOff 7),
>            Stamp 6 (NoteOn 11 6000 0.1), Stamp 8 (NoteOff 11) ]
```

When a note-on event occurs, we wish to generate a synchronous stream containing a sinusoid of the appropriate frequency, that lasts until the corresponding note-off event. To do so, we create, on each note-on event, a new process that starts with that note-on event, and terminates when it sees the note-off event.

```
> noteWatcher :: NoteEvent -> [Stamp NoteEvent] -> [Stamp NoteEvent]

> noteWatcher (NoteOff noteid) events = []
> noteWatcher (NoteOn noteid _ _) events
>             = takeUntilT (noteTurnsOff noteid) events
>     where
>     noteTurnsOff noteid (NoteOff nid)
>         | noteid == nid   = True
>         | otherwise       = False
>     noteTurnsOff noteid _ = False
```

Generating the note is simply a matter of converting the time-stamped stream output by *noteWatcher* into a synchronous stream. This is done by generating a stream of phase values with *resyncT*, applying the *sin* function to each, and multiplying by the amplitude of the note:

```
> noteGenerator :: [Stamp NoteEvent] -> [Float]
> noteGenerator noteonoff
>     = (map (* ampl) . map sin . resyncT f g 0.0) noteonoff
>     where
>     f x phase = fmod (phase + phinc) (2 * pi)
>     g phase   = fmod (phase + phinc) (2 * pi)

>     fmod x m = if x > m then x - m else x
>     phinc    = (2 * pi * freq) / sampleRate
>     pi       = 3.1415926535

>     NoteOn _ freq ampl = head (desyncT noteonoff)
```

The sample rate of the synthesiser is a global variable:

```
> sampleRate = 32000 :: Float
```

The complete synthesiser is a pipeline of four stages: i) a *noteWatcher* process for each active note; ii) a filtering process that removes empty streams (caused by note-off events); iii) a *noteGenerator* process for each active note; and iv) a *combineT* sum' process to sum all notes together into the output signal. Thus:

```
> synthesiser :: [Stamp NoteEvent] -> [Float]
> synthesiser = combineT sum'
>             . mapT noteGenerator
>             . filterT (not.null)
>             . mapContextT noteWatcher
```

Figure 3 shows the output of the synthesiser, together with the contribution from individual notes, obtained by executing `synthesiser events`. Note the sharp discontinuities cause by the abrupt ending of each note, which will result in audible clicks. A more realistic synthesiser program would have an amplitude slope applied to the beginning and end of notes to prevent this.

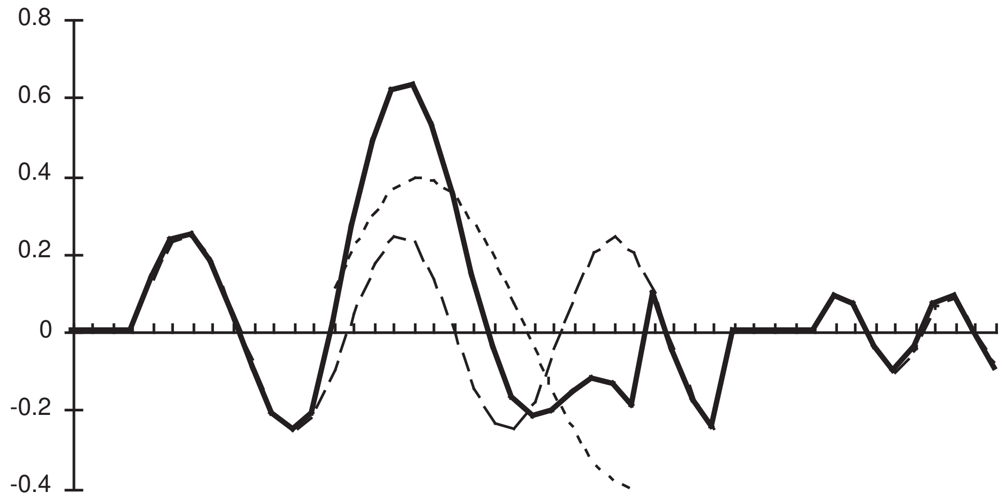

Figure 3: Synthesiser output

7 Conclusions

Although expressing programs that use asynchronous streams seems easy enough, implementing them in real time is a different matter. For a start, the unpredictable time and space behaviour of graph reduction [6]—the most common implementation technique for lazy functional languages—makes real-time execution impossible. A more immediately useful approach would be to view the Haskell functions presented here as a prototyping and simulation tool. [12], for example, have used functional prototypes (written in the strict language SML) to model and to predict performance of parallel programs.

By appropriate analysis or through programmer-supplied annotations that indicate strictness conditions, however, real-time implementation may be possible. For example, [1] show how appropriate programming constructs can be used to "escape" from lazy evaluation in special circumstances. While the program as a whole can take advantage of the programming style of lazy languages, the programmer can still specify regions in which analysis and optimisation techniques developed for strict languages can be used. In the context of stream-based programs, it may be possible to use this idea to enable efficient analysis and code generation techniques developed for dataflow programs [8, 2] to be applied to functional process network programs. Work on this idea is currently in progress.

References

[1] Steven Anderson and Paul Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proc. Programming Language Design and Implementation (PLDI)*, White Plains, New York, June 1990.

[2] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, pages 28–45, October 1990.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[4] Paul Hudak et al. Report on the functional programming language Haskell, a non-strict purely-functional language, version 1.2. *SIGPLAN Notices*, May 1992.

[5] Mark Jones. *An Introduction to Gofer*, version 2.21 edition, 1991. The source code and documentation for Gofer is available by anonymous ftp from dcs.glasgow.ac.uk.

[6] Simon L. Peyton Jones. *The Implementaion of Functional Programming Languages.* Prentice-Hall, 1987.

[7] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Reasearch Monographs in Parallel and Distributed Computing. Pitman Press, 1989.

[8] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[9] H. John Reekie. Towards effective programming for parallel digital signal processing. Technical Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[10] D.A. Turner. An approach to functional operating systems. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 199–218. Addison-Welsey, 1990.

[11] W. W. Wadge and A. Ashcroft. *Lucid–the Dataflow Programming Language.* Academic Press, 1985.

[12] A. Wallace, G. Michaelson, P. McAndrews, K Waugh, and W. Austin. Dynamic control and prototyping of parallel algorithms for intermediate and high-level vision. *IEEE Computer*, 25(2), February 1992.

A Merge Revisited

For completeness, I give here a definition of a function that performs the fair merge operation discussed in the Introduction. The fair merge is obliged to produce on its output, at some time, every value that occurs on either of its two inputs. The definition given here resolves a collision between two input values in the same time slot by outputting both of them. A version of fair merge that delays one until the next time slot is a little more complex, but not difficult to define[6].

```
> data OneOrMany a = One a
>                  | Many [a]

> mergeT :: [Stamp a] -> [Stamp a] -> [Stamp (OneOrMany a)]
> mergeT (Stamp t1 x : xs) (Stamp t2 y : ys)
>     | t1 < t2  = Stamp t1 (One x) : mergeT xs (Stamp (t2-t1) y : ys)
>     | t1 > t2  = Stamp t2 (One y) : mergeT (Stamp (t1-t2) x : xs) ys
>     | t1 == t1 = Stamp t1 (Many [x,y]) : mergeT xs ys
> mergeT _ _     = []
```

[6]See the source of this paper if required.

The fact that I had to make explicit access to time-stamps to construct the fair merge function indicates that the set of higher-order functions I have defined for time-stamped streams is not yet powerful enough to serve as the basis of a general-purpose stream ADT.

B Function Definitions

Following are function definitions elided from the main text.

combineT merges synchronous streams contained within a time-stamped stream. It accepts a function argument; at each tick, this function is applied to a list containing the front element of the synchronous streams active at that time. The *curr* variable represents the set of currently active streams:

```
> combineT :: ([a] -> b) -> [Stamp [a]] -> [b]
> combineT f ss = combineT' f ss []
>     where
>     combineT' f [] [] = []
>     combineT' f [] curr
>         = f (map head curr) : combineT' f [] (stails curr)
>     combineT' f (Stamp 1 s : ss) curr
>         = f (map head (s : curr)) : combineT' f ss (stails (s : curr))
>     combineT' f (Stamp t s : ss) curr
>         = f (map head curr)
>                        : combineT' f (Stamp (t-1) s : ss) (stails curr)
>
>     stails :: [[a]] -> [[a]]
>     stails = filter (not . null) . map tail . filter (not . null)
```

delayT delays a signal by a specified number of ticks:

```
> delayT :: Int -> [Stamp a] -> [Stamp a]
> delayT k (Stamp t x : xs) = Stamp (t+k) x : xs
```

deltaT extracts the time from time stamps:

```
> deltaT :: [Stamp a] -> [Stamp Int]
> deltaT []               = []
> deltaT (Stamp t x : xs) = Stamp t t : deltaT xs
```

desyncH and *desyncT* remove timing information from a stream:

```
> desyncH :: [Slot a] -> [a]
> desyncH []            = []
> desyncH (Hiaton : xs ) = desyncH xs
> desyncH (Val x  : xs ) = x : desyncH xs

> desyncT :: [Stamp a] -> [a]
> desyncT []                = []
> desyncT (Stamp t x : xs ) = x : desyncT xs
```

dropT advances a stream by a specified number of ticks:

```
> dropT :: Int -> [Stamp a] -> [Stamp a]
> dropT _ []               = []
> dropT 0 xs               = xs
> dropT k (Stamp t x : xs)
>          | k < t         = Stamp (t-k) x : xs
>          | otherwise     = dropT (k-t) xs
```

fillH and *fillT* produce synchronous streams by inserting a specified value:

```
> fillH :: a -> [Slot a] -> [a]
> fillH val = map strip where
>     strip Hiaton  = val
>     strip (Val x) = x

> fillT :: a -> [Stamp a] -> [a]
> fillT val = resyncT f g val where
>     f x a = x
>     g a   = val
```

filterH and *filterT* remove elements of asynchronous streams that do not satisfy a predicate:

```
> filterH :: (a -> Bool) -> [Slot a] -> [Slot a]
> filterH p = map f where
>     f Hiaton              = Hiaton
>     f (Val x) | p x       = Val x
>               | otherwise = Hiaton

> filterT :: (a -> Bool) -> [Stamp a] -> [Stamp a]
> filterT p = f 0 where
>     f s [] = []
>     f s (Stamp t x : xs)
>         | p x       = Stamp (s+t) x : f 0 xs
>         | otherwise = f (s+t) xs
```

holdH and *holdH* produce synchronous streams by performing a zero-order hold:

```
> holdH :: a -> [Slot a] -> [a]
> holdH val = prefl hold val where
>     hold v Hiaton  = v
>     hold v (Val x) = x

> holdT :: a -> [Stamp a] -> [a]
> holdT val = resyncT f g val where
>     f x a = x
>     g a   = a
```

selectH and *selectT* remove elements from a synchronous stream to produce an asynchronous stream:

```
> selectH :: (a -> Bool) -> [a] -> [Slot a]
> selectH p = map f where
>     f x | p x       = Val x
>         | otherwise = Hiaton

> selectT :: (a -> Bool) -> [a] -> [Stamp a]
> selectT p = stamp . selectH p
```

stamp and *unstamp* convert between hiatonic and time-stamped streams:

```
> stamp :: [Slot a] -> [Stamp a]
> stamp = stamp' 1 where
>     stamp' count []            = []
>     stamp' count (Hiaton : xs) = stamp' (count+1) xs
>     stamp' count (Val x  : xs) = Stamp count x : stamp' 1 xs

> unstamp :: [Stamp a] -> [Slot a]
> unstamp = resyncT f g Hiaton where
>     f x a = Val x
>     g a   = Hiaton
```

takeT and *takeUntilT* terminate a stream after a specified number of ticks, or when a value satisfies a predicate:

```
> takeT :: Int -> [Stamp a] -> [(a, Int)]
> takeT _ []                = []
> takeT 0 _                 = []
> takeT k (Stamp t x : xs)
>          | k < t          = []
>          | otherwise      = (x,t) : takeT (k-t) xs

> takeUntilT :: (a -> Bool) -> [Stamp a] -> [Stamp a]
> takeUntilT p []           = []
> takeUntilT p (Stamp t x : xs)
>          | p x            = [Stamp t x]
>          | otherwise      =  Stamp t x : takeUntilT p xs
```

upsampleH and *upsampleT* increase the clock rate by some factor k:

```
> upsampleH :: Int -> [Slot a] -> [Slot a]
> upsampleH k = concat . map extend where
>     extend Hiaton  = copy k Hiaton
>     extend (Val x) = Val x : copy (k-1) Hiaton

> upsampleT :: Int -> [Stamp a] -> [Stamp a]
> upsampleT k = dropT (k-1) . map ups where
>       ups (Stamp t x) = Stamp (k * t) x
```

whenT samples a synchronous stream at times given by a time-stamped stream:

```
> whenT :: [a] -> [Stamp b] -> [Stamp a]
> whenT xs ys = whenT' 1 xs ys
>     where
>     whenT' :: Int -> [a] -> [Stamp b] -> [Stamp a]
>     whenT' i (x:xs) (Stamp 1 y : ys) = Stamp i x : whenT' 1 xs ys
>     whenT' i (x:xs) (Stamp t y : ys) = whenT' (i+1) xs (Stamp (t-1) y : ys)
>     whenT' _ _ _                     = []
```

zipWithT performs a point-wise operation on two streams with the same time-stamps:

```
> zipWithT :: (a -> b -> c) -> [Stamp a] -> [Stamp b] -> [Stamp c]
> zipWithT f  (Stamp t1 x : xs) (Stamp t2 y : ys)
>     | t1 == t2  = Stamp t1 (f x y) : zipWithT f xs ys
>     | otherwise = error "Unmatched time stamp in zipWithT"
> zipWithT f _ _  = []
```

C Utility Functions

In order to print asynchronous streams more neatly, here are versions of *show* for the *Slot* and *Stamp* datatypes:

```
> instance (Text a) => Text (Slot a) where
>     showsPrec p Hiaton s  = "*" ++ s
>     showsPrec p (Val x) s = (show x) ++ s

> instance (Text a) => Text (Stamp a) where
>     showsPrec p (Stamp t x) s = show x ++ "_" ++ show t ++ s
```

For the synthesiser example, it is helpful to be able to display note-on and note-off events:

```
> instance Text NoteEvent where
>     showsPrec p (NoteOn n f a) s
>         = "(NoteOn: " ++ show n ++ " "
>            ++ " " ++ show f ++ " " ++ show a ++ ")" ++ s
>     showsPrec p (NoteOff n) s
>         = "(NoteOff: " ++ show n ++ ")" ++ s
```

group changes a list into a list of lists; each sub-list (except perhaps the last) contains k elements:

```
> group :: Int -> [a] -> [[a]]
> group k [] = []
> group k xs = take k xs : group k (drop k xs)
```

pairs produces all pairs of adjacent elements. The first pair contains a specified element and the first element of the list:

```
> pairs :: a -> [a] -> [(a,a)]
> pairs x0 (x1:xs) = (x0,x1) : pairs x1 xs
> pairs _ _        = []
```

An overloaded version of *sum* is required for Gofer (but is standard in Haskell):

```
> sum' :: (Num a) => [a] -> a
> sum' = foldl (+) 0
```

The version of Gofer I used did not have transcendental floating-point routines. I added a primitive to the Gofer run-time and the following Gofer code (the current version has these built in):

```
> primitive primSinFloat  "primSinFloat" :: Float -> Float
> sin :: Float -> Float
> sin = primSinFloat
```

The filtering examples need some filter coefficients. The following coefficients are a 31-tap low-pass filter with cut-off frequency of 0.1 of the sample rate (3.2 kHz) (the coefficients were generated by Matlab):

```
> lowpass = [ -0.0005, 0.0006, 0.0022, 0.0041, 0.0050, 0.0028, -0.0039,
>             -0.0142, -0.0239, -0.0259, -0.0133, 0.0179, 0.0653, 0.1198,
>              0.1679, 0.1960, 0.1960, 0.1679, 0.1198, 0.0653, 0.0179,
>             -0.0133, -0.0259, -0.0239, -0.0142, -0.0039, 0.0028, 0.0050,
>             0.0041, 0.0022, 0.0006, -0.0005 ]
```

To test *resampleT*1, we can use this FIR filter function:

```
> fir :: (Num a) => [a] ->[a] -> [a]
> fir coeffs signal = fir' (copy ((length coeffs)-1) 0 ++ signal)
>     where
>     fir' []     = []
>     fir' signal = ip coeffs signal : fir' (tail signal)

>     ip xs ys    = sum' (zipWith (*) xs ys)
```

To generate test sinusoids, we can use this function (use *take* to make the list finite before printing it!):

```
> wave freq = map sin (iterate (+ phinc) 0.0)
>     where
>     phinc = (2 * pi * freq) / sampleRate
>     pi    = 3.1415926535
```

Our test signal is a 2000 Hz sinewave sampled at intervals between two and five ticks:

```
> jittery = (wave 2000.0) `whenT`
>              (map (\t -> Stamp t (0::Int)) (cycle [3,5,4,5,2,4]))
```