A Signal Processing Toolkit.

6.898 Final Project
Jay B. Hancock

Abstract - *I implement a signal processing toolkit in Ocaml to take advantage of type checking. Processors implement a mapping between input and output. Signals store values between processors, and at inputs and outputs to the system. A system graph is built via the interconnection of processor nodes and signal edges. Causal assumptions, plus an ordered production of output points from processors allow us to insure that the system will operate properly. The system is tested in an IIR recursive filter definition.*

Background and Motivation

Digital Signal Processing uses a mathematical entity called a signal, which is a function that maps the integers to an abstract data type, and operations on signals, such as convolve, add, delay, and fast-fourier transform. A system is a directed graph of operation nodes and signal edges, with one node as the input and another node as the output. The main book of influence for such a system is [1].

In a real-time application, we want to stream data through a sequence of processors. Some processors have different signal input vs. output rates, which indicates that some signals will require greater storage than others.

There are two possibile design choices for building systems. The first is to take advantage of currying the processor functions, which would allow signal edges between processor nodes to disappear. The second is to define an output signal for each processor node, so that all processors are connected by signals. The latter may be more inefficient, but it is closer to the model of a signal flow graph used in DSP courses, thus I chose it.

To be efficient, we would rather have a push model instead of a pull model. If we use a pull model, then the processor would waste cycles polling the signal to see if its value has been defined. For the push model, we shall consider the observer design pattern. Then, the signal is a subject to be observed by processors. When the signal's state changes, it notifies its processor observers so that they may attempt to generate their next output values.

Since real-time data is causal, we impose three conditions. First, that any signal point for $n<0$ equals 0. We note one exception to this rule, and that will be when we are generating a mathematical signal, such as $\exp(jx)$. Secondly, that all processors will produce outputs successively, i.e. the first output point we produce will be at $n=0$, and given that we just produced an output at point $n=N$, then the next point we must produce is $n=N+1$. Third, that all processors must operate causally, i.e. when a processor produces an output point at time $n=N$, then it must only rely on input values for times $n \leq N$. Using this policy, signals generated as outputs of processors will not accidentally skip generating a value and leave the system stuck. Also, the total delay to the system will be the minimum sum delay through each non-feedback paths from input to output. As a practical matter, processors input points must have a finite delay from the output point, so infinite values cannot be used, which may otherwise cause the processor to wait forever.

We want to assure correct operation of the system. Given the above properties, if all inputs are defined starting at n=0 and successively, then all output points will be eventually defined. We want to check that this will ensure proper operation in all cases, including those with feedback loops. I will simply assert that this is the case for our purposes in this paper.

As a practical matter, signals have finite memory capacity, even if they have infinite domain. It is necessary to purge the signal's memory of values that are no longer needed. We can determine these values *a priori* because processors have finite delays. The memory requirement of a signal is proportional to the memory requirement the processor. For instance, $y_1[n] = x[n]$, where y is the output and x is the input, requires one point of memory, while $y_2[n] = x[n-1] + x[n-2]$ requires two memory points. Because we are generating point successively, $y_3[n] = x[n] + x[n-M]$ requires M memory points.

The locking function is therefore a function that maps n to an interval of the domain. For the three cases above, the locking function is $l_1(m) = \{m \mid m \geq n\}$, $l_2(m) = \{m \mid m \geq n-2\}$, and $l_3(m) = \{m \mid m \geq n-M\}$. Since signals are causal, all normal locking functions will take the form of $l_3(m)$, where M is dependent on the processor. There may be unforeseen cases where the locking function will not take this form. One such example is, if we want to debug a system, we can set the locking function to All so that no points will be thrown away.

A signal can have several observers. The locking mechanism must account for several locking functions on a signal. Since the observers will change their locking function independently of each other, we must store each observer's function separately and take the union of them to determine the total locked region.

Purging the signal is necessary for memory management. A purge operation simply compares defined points in the signal with the locking function and deletes the points that are not locked.

There are two possible techniques for purging the signal. One is to purge a particular signal each time the locking function is updated. The purge operation could be expensive, so a second improved technique is to purge all signals when memory usage reaches a certain size. Despite expense, I chose the first technique.

A simple mechanism will simply recheck all points against the lock, while a more complicated mechanism could identify the points that were just unlocked, and only compare those. If points in the signal data structure are stored successively, and all locking functions are GreaterThanEqual, then the points can be checked and purged in order from the beginning of the storage until the first locked point. In a real system, if the purge command checks all points in the data structure, it could have an overhead as high as the number of memory points, which is M from y_3 above. This may not be deterministic if the processor relies on a stream with random delays.

Care must be taken during initialization because multiple observers can be added to the signal. Initialization must set the lock, but not cause a purge command. If it were to purge the signal, then one processor may start observing a signal and erase it before a second processor is attached to observe the same data that was just erased.

After initialization of the system, it may be run by defining data on the input stream(s) of the system. Once the system is run, there are no guarantees about unlocked data. Thus, if input data is stopped and another processor is attached as an observer to

any stream in the system, this processor can only rely on locked values. In the system where updating the lock causes a purge operation, the output with no observers will never be purged. However, if the system has a global purge based on memory filling up, then the output would be purged unless it has an output observer that simply sets the lock to All and does nothing when notified. I chose to purge all input signals whenever a processor generates an output and updates the lock.

The definition of a locking mechanism can make use of Ocaml's pattern matching. The type <interval> has constructors called GreaterThan of 'a, Between of 'a*'a, All, None, Or of 'a*'a, etc. Intervals can be converted into functions that test whether an element 'a lies within the interval, and whether or not two intervals overlap.

Processors kernels that are useful include:

- Scale ($y[n] = a * x[n]$)
- Delay ($y[n] = x[n-M]$)
- Add ($y[n] = \Sigma x_i[n]$)
- Impulse ($y[n] = 1$ for $n=0$, 0 otherwise)
- Convolve ($y[n] = \Sigma^N_{m=0} x[n-m]h[m]$)

Summary and Evaluation

The code utilizes the objective part of Ocaml's language, including multiple-inheritance and parameterized classes. The Observer pattern is very easy to implement and use in Ocaml. I implemented the observer pattern as a module by modifying code from the Ocaml O'Reilly book. Of coarse, there are many possible variants of the observer pattern, such as those whose subjects pass events to their observers. One possible improvement to the Observer module is to make it into a functor that produces these variants.

The intervals module describes the interval type along with a function that checks if an element is a member of an interval. I attempted to define a more complicated function to determine if two intervals shared any members, but decided the effort required was far greater than time available.

The lock module made use of intervals. The multi-lock class, parameterized over the type of lock ID and the type of domain points (used in interval), contains a map from ID to Array index, and an Array data structure that stores the intervals. A method *create_locker*, which requires an ID, creates a locker if no locker already exists for the ID. The method *set_lock* replaces the interval in Array for the ID provided with a new interval. The method *get_lock* tests whether a value of the domain is locked.

The signal module includes two classes. The parent class, *signal_basic*, inherits the subject class, and provides methods for defining values of the signal and getting those values. The child class, *signal*, inherits from both *lock* and *signal_basic*, and overrides the *subject* class's *add_observer* method by adding a lock in addition to registering the observer with the subject. The lock ID is the observer object ID. The purge method is included in *signal*, and removes values from *signal*'s data that are not locked.

The processor module includes two virtual classes, *generator* and *processor*. The *generator* class defines output points based on files or mathematical functions, and in the

processor derived class, based on input signals. Classes derived from *generator* can describe how to produce outputs, but for the additional functionality of observing input signals and for locking them, classes must be derived from the *processor* class.

The *generator* contains an output signal and requires derived classes to implement the virtual function *kernel*, which describes how the *generator* will produce values. The generator keeps a single state value, the index of the next output value it must produce, initialized to zero. The *output* method advances the state by one, and tries to generate the previous state's output value using the kernel function. The kernel function will either return the value of the next output point, or raise the *Undefined_value* exception. If a value is returned, then the output is defined and the output signal's observers are called. If the *Undefined_value* exception is called, then a handler will reset the state to point to the index of the output value that just failed. When an output point is defined and the output signal's observers are called, a call tree is generated and may be circular if a processor's output is fed back to one of its subjects. Thus the state must be advanced before calling so that if the same processor is called via recursion its state will reflect the appropriate output index to be generated.

The *processor* is derived from *generator* to provide output and from *observer* to be notifies of input. The processor has the responsibility for updating the lock of its inputs. The derived classes must implement three virtual functions: *kernel* from the generator class, *register inputs* to add the processor as observers for either signals or lists of signals, and *update_lock*, which defines the lock interval based on the value of the state (output point to be generated).

The architecture is easy to extend. The following generators and processors were written within a couple hours and used to help debug the architecture. *Adder*s, *Delay*s and *Scaler*s can be used to implement IIR Filters, but the *causal_IIR_filter* implements the above kernels efficiently.

The file <test3.ml> shows examples of the processors in use. I would like to add a higher-level module called System.ml that uses a graph structure describe the interconnections between processors. Then a nice graphical interface will allow visual programming of systems. Maybe I can interface this with graphical models, HMMs, and FSTs to build an actual speech recognizer.

Generators I implemented include:

- *Impulse* y[n] = if n=0 then a else 0.0
- *Step* y[n] = if n<0 then 0.0 else a
- *Ramp* y[n] = a * n

Processors I implemented include:

- *Identity* y[n] = x[n]
- *Scale* y[n] = a * x[n]
- *Delay* y[n] = x[n-M]
- *Adder* y[n] = $\Sigma x_i[n]$
- *Causal_FIR_filter* y[n] = sum m=0..(len(x_coeffs)-1) {x_coeff.(m) *. x(n-m)}. A special case of *Convolve.*
- *Causal_IIR_filter* y[n] = x[n] + sum m=1..len(y_coeffs) a_m y[n-m]

Lessons Learnt

I learned about the object system in Ocaml. I find it more delightful to use than C++ objects, and very easy to understand exactly what is going on. The relationships of derived classes and subclasses are distinct. The object system can be combined with the module system to provide a very powerful organization of the software.

The debugger and interpreter provide enough complimentary tools that make Ocaml a favorite language of mine. One fellow on the web said, and I agree with, that you feel as if you actually understand data structures clearly when using Ocaml, as opposed, say C++.

The observer pattern is powerful, and special considerations must be considered if the observer relationships become recursive, like state update and recursion escape.

Bibliography.

[1] Oppenheim, Schafer, and Buck. Discrete Time Signal Processing. Prentice Hall; ISBN: 0137549202; 2nd edition (February 15, 1999).

Test3.ml

```
open Signal
open Processor
open Intervals

let main () =
(*
  let a = new ramp_generator 1.0 in
  let e = new adder in
  let out = new identity e#y in (* e#y is output of adder, provided as
input*)
  let c = new delay e#y 1 0.0 in
  let d = new scale_float c#y 2.0 in
*)

  let a2 = new ramp_generator 1.0 in
  let f = Array.of_list [2.0] in
  let i = new causal_iir_filter a2#y f in

(*
  e#add_inputs [a#y; d#y];
  a#output;
*)
  a2#output;;


(* problem: d doesn't know about the delay in c, therefore doesn't give
zero
value to e, instead always raising undef.  maybe chain delays together?
system should be initialized to zero values, so that zeros propagate.
*)

main () ;;
```

```
(*
(* IIR system implemented with basic parts *)
  let a = new ramp_generator 1.0 ;;
  let e = new adder ;;
  let out = new identity e#y;;
  let c = new delay e#y 1 0.0 ;;
  let d = new scale_float c#y 2.0 ;;
  e#add_inputs [a#y;d#y];;

  a#output;;

(* same system with an IIR part *)
  let a2 = new ramp_generator 1.0 ;;
  let f = Array.of_list [2.0] ;;
  let i = new causal_iir_filter a2#y f ;;
  let out2 = new identity i#y;;

  a2#output;;


(* IIR and FIR filters in series: *)
  let a3 = new step_generator 1.0 ;;
  let f = Array.of_list [(-0.5); 0.9];;
  let i2 = new causal_iir_filter a3#y f ;;
  let out3 = new identity i2#y;;
  let pipe = new identity i2#y;;

  let f2 = Array.of_list [0.5; 1.0; 0.5];;
  let i3 = new causal_fir_filter pipe#y f ;;
  let out4 = new identity i3#y;;

  (* CANT DO THIS?  need to decouple typing for multiple observers of
various
     types.  splitting and adding values *)
  let out5 = new adder  ;;
  let out5b = (out5 : ('a,'b) adder :> ('b,float) processor);;
  out5b#add_inputs [out3#y; out4#y];;

  a3#output;;

  (* processor init with a stream will use all values currently
defined:*)
  let out4b = new identity out4#y;;
  (* but, it will not purge the previous stream in case we haven't
finished
     connecting processors *)

  (* once we run the system again, it will push more data through and
purge
     all streams *)
  a3#output;;


(* IIR and FIR filters in series: *)

(* a step generator is zero for values less than zero *)
```

```
  let a3 = new step_generator 1.0 ;;
(* we produce an IIR filter, which implements:
   y[n] = x[n] + (-0.5) *. y[n-1] + (0.9) *. y[n-2]
*)
  let f = Array.of_list [(-0.5); 0.9];;
  let i2 = new causal_iir_filter a3#y f ;;
(* Here we attach an output, which will save data, and a pipe, which
will
   attach to a later stage.  When a stream has observers, the data will
be
   purged when all the observers say it's ok.  I could attach a "no
purge"
   processor, which locks the entire stream. *)
  let out3 = new identity i2#y;;
  let pipe = new identity i2#y;;
(* and an FIR filter... y[n] = 0.5 x[n] + 1.0 x[n-1] + 0.5 x[n-2] *)
  let f2 = Array.of_list [0.5; 1.0; 0.5];;
  let i3 = new causal_fir_filter pipe#y f2 ;;
(* and the final output *)
  let out4 = new identity i3#y;;


# val a3 : < notify_at : unit; _.. > Processor.step_generator = <obj>
# val f : float array = [|-0.5; 0.9|]
# val i2 : ('a, 'a) Processor.causal_iir_filter as 'a = <obj>
# val out3 : ('a, float, 'a) Processor.identity as 'a = <obj>
# val pipe : ('a, float, 'a) Processor.identity as 'a = <obj>
# val f2 : float array = [|0.5; 1; 0.5|]
# val i3 : ('a, 'a) Processor.causal_fir_filter as 'a = <obj>
# val out4 : ('a, float, 'a) Processor.identity as 'a = <obj>

(* Here we tell the generator a3 to produce output *)
a3#output;;
- : unit = ()

(* We check the output streams with nothing attached and find our
filtered
values *)
out3#y#get_defined;;
- : (Signal.domain * float) list =
[(9, -0.642134375); (8, 3.41038125); (7, 0.0700625); (6, 2.717125);
 (5, 0.47625); (4, 2.1725); (3, 0.625); (2, 1.65); (1, 0.5); (0, 1)]

out4#y#get_defined;;
- : (Signal.domain * float) list =
[(9, 3.3904103125); (8, -1.642134375); (7, 2.41038125); (6, -
0.9299375);
 (5, 1.717125); (4, -0.52375); (3, 1.1725); (2, -0.375); (1, 0.65);
 (0, -0.5)]

(* after the system is in operation, output waiting for use, we can
attach
further processors: *)
let out4b = new identity out4#y;;
# val out4b : ('a, float, 'a) Processor.identity as 'a = <obj>
```

```
(* note, that initialization of a processor causes it to immediately
process
defined data from input streams, but it doesn't purge those streams in
case
we want to connect more processors. *)

(* here is the new output: *)
out4b#y#get_defined;;
- : (Signal.domain * float) list =
[(9, 3.3904103125); (8, -1.642134375); (7, 2.41038125); (6, -
0.9299375);
 (5, 1.717125); (4, -0.52375); (3, 1.1725); (2, -0.375); (1, 0.65);
 (0, -0.5)]

(* here is the previous output, which remains untouched: *)
out4#y#get_defined;;
- : (Signal.domain * float) list =
[(9, 3.3904103125); (8, -1.642134375); (7, 2.41038125); (6, -
0.9299375);
 (5, 1.717125); (4, -0.52375); (3, 1.1725); (2, -0.375); (1, 0.65);
 (0, -0.5)]

(* when we tell the generator to push more input into the head of the
system,
then the locking mechanism will purge data not needed by observers. *)
a3#output;;
- : unit = ()

(* we see here that the identity processor only requires the present
value, so
all values in the previous stream are purged *)
#   out4#y#get_defined;;
- : (Signal.domain * float) list = []

(* and there are more output points resulting from the second set of
input
points from the generator *)
# out4b#y#get_defined;;
- : (Signal.domain * float) list =
[(19, 22.7948353484); (18, -17.3122652851); (17, 15.2652252287);
 (16, -11.1996140787); (15, 10.2949090993); (14, -7.16906614336);
 (13, 7.01152891953); (12, -4.51477964844); (11, 4.83793232812);
 (10, -2.77312609375); (9, 3.3904103125); (8, -1.642134375); (7,
2.41038125);
 (6, -0.9299375); (5, 1.717125); (4, -0.52375); (3, 1.1725); (2, -
0.375);
 (1, 0.65); (0, -0.5)]


*)
```

Processor.ml

```
open Signal
open Observer
```

```
open Intervals


(* the generator requires a <kernel> for determining an output from an
index
number.  each call to self#output will attempt to produce <buf_len>
number of
output values from the <kernel> function.  If the kernel throws an
<Undefined_value> exception, then the generator quits producing output
points.
This is used in the <processor> derived class.  *)

class virtual ['O, 'T] generator (buf : int) =
  object (self)
    val mutable output : ('O,'T) signal = new signal

    val mutable n : int = 0
    val mutable buf_len = buf
    val mutable count : int = 0

    method virtual kernel : int -> 'T
    method start_at (n' : int) = n <- n'

    method init () = ()

    method output =
      try
      for i = 1 to buf_len do
          (* THIS is messy.  n++ in case this def works! Must be easier
way *)
        n <- succ n;
        output#define (self#kernel (n-1)) (n-1)
        done
      with
        Undefined_value -> n <- pred n

    method y = output
    method get m = output#get m (*  called like a signal *)
end;;

(* the <processor> inherits generator to produce output points, and the
<kernel> function can rely on input <signal>s.  When the kernal trys to
<get> a
value from an input <signal>, if the value is undefined, then the
kernel
returns an <Undefined_value> exception, which causes the <generator>
super
class to quit producing points.

The <processor> also inherits <observer>, and derived classes must
implement
the <register_inputs> function that should call <add_observer> for each
input
<signal> in the derived class.  The <signal> as the <subject> calls the
<processor> as an <observer> using the <notify_at> function when there
are
```

```
newly defined values in an input <signal>.  The <processor> then
initiates
calls to <generator#output> to again attempt to produce more <output>
points.

The <processor> also must implement a <lock> on the input <signals>
(and in
some cases, like IIR filters, on the output signal too) by implementing
the
<update_lock> function.  It depends on the value <n> from the generate
class,
and the function is written to lock any present and future value needed
by a
signal.  The <lock> description uses the <interval> data type, which
allows
easy description of complicated intervals.

Upon initialization, the processor calls <register_inputs> and
<update_lock> so
that the streams know about the observers and so the processor
immediately
locks the data points it needs.  Each processor also calls
<generator#output>.
It is important that <kernel>s define negative values for IIR filters
and
delays.  For initialization purposes, a <processor> must not get stuck
because
of loops.  Consider how the delay kernel is written so that it returns
zero for
values less than the delay.  During initialization, the delay function
pushes
zeros onto its output stream until the delay so that feedback loops can
utilize those zeros in producing futures inputs for the delay.

(imagine an IIR filter:
            x[n] -> adder -> scaler -> split -> output
                       \---scaler<--delay/
 the adder requires a value from scaler at time zero before it will
continue,
 so the delay had better produce a zero that can be passes on by the
scaler.
)

*)

class virtual ['O, 'T] processor (buf : int) =
  object (self)
    inherit observer
    inherit ['O, 'T] generator buf as generator

    initializer self#init ()

    method init () =
      self#register_inputs ();
      self#update_lock ();
      generator#output
```

```
(*    method
      self#notify_at *)

    method virtual register_inputs : unit -> unit
    method virtual update_lock : unit -> unit

    method notify_at =
      generator#output;
      self#update_lock ()
end;;


(* y[n] = a' * n *)
class ['Oout] ramp_generator (a' : float) =
  object (self)
    inherit ['Oout, float] generator 10
    method kernel (n:int) = a' *. (float_of_int n)
end;;

(* y[n] = 1 for n=0, 0 otherwise *)
class ['Oout] impulse_generator (a' : float) =
  object (self)
    inherit ['Oout, float] generator 10
    method kernel (n:int) = if (n=0) then a' else 0.0
end;;

(* y[n] = 1 for n>=0, 0 otherwise *)
class ['Oout] step_generator (a' : float) =
  object (self)
    inherit ['Oout, float] generator 10
    method kernel (n:int) = if (n>=0) then a' else 0.0
end;;

(* y[n] = x[n] *)
class ['Oin, 'T, 'Oout] identity (x' : ('Oin,'T) signal) =
  object (self)
    inherit ['Oout, 'T] processor 10 as super
    val mutable x = x'

    method register_inputs () = x#add_observer(self)
    method kernel n = x#get n
    method update_lock () = x#lock (self) (GreaterThanEq n)
end;;

(* y[n] = a' * x[n] *)
class ['Oin, 'Oout] scale_float (x' : ('Oin, float) signal) (a' :
float) =
  object (self)
    inherit ['Oin, float, 'Oout] identity x'
    val a = a'
    method kernel n = (x#get n) *. a
end;;

(* y[n] = x[n-m] u[n-m] *)
class ['Oin, 'Oout, 'T] delay (x' : ('Oin, 'T) signal) (m :
Signal.domain)
 (zero_val': 'T) =
```

```
  object (self)
    inherit ['Oin, 'T, 'Oout] identity x'
    val n' = m
    val zero_val = zero_val'

    method kernel n = if (n<n') then zero_val else (x#get (n-n'))
    method update_lock () = x#lock self (GreaterThanEq (n-n'))
end;;

(* y[n] = x1[n] + x2[n] + ... + xN[n] ; because of feedback, the inputs
are
provided after definition; initialization occurs when inputs are given
*)
class ['Oin, 'Oout] adder  =
  object (self)
    inherit ['Oout, float] processor 10 as super
    val mutable x = []

    method add_inputs (x' : ('Oin,float) signal list) =
      x <- x';
      self#init ()

    method register_inputs () =
      List.iter (fun s -> s#add_observer(self)) x
    method kernel n =
      if (List.length x > 0) then
        List.fold_left (+.) 0.0 (List.map (fun s->s#get n) x)
      else
        raise Signal.Undefined_value
    method update_lock () =
      List.iter (fun s->s#lock (self) (GreaterThanEq n)) x
end;;

(* implements y[n] = sum m=0..len(x_coeffs)-1 {x_coeff.(m) *. x(n-m)}
*)
class  ['Oin, 'Oout] causal_fir_filter (x' : ('Oin, float) signal)
 ( x_coeffs' : float array )=
  object (self)
    inherit ['Oin, float, 'Oout] identity x'
    val x_coeffs = x_coeffs'
    val max_delay = (Array.length x_coeffs') - 1

    method kernel n = (* could be more efficient... *)
      (Array.fold_right (+.)
        (Array.mapi
          (fun n' e ->
            if (n-n') < 0 then 0.0 else (x#get (n-n')) *. e)
              x_coeffs)
          0.0
      )

    method update_lock () = x#lock (self) (GreaterThanEq (n-max_delay))
end;;

(* implements an iir filter: y[n] = x[n] + sum m=1..len(y_coeffs) a_m
y(n-m).  values of y for negative n are zero. the array y_coeffs
describe the a_i's above.  the coeff for x[n] is fixed at 1 *)
```

```
class ['Oin, 'Oout] causal_iir_filter (x' : ('Oin, float) signal)
  ( y_coeffs' : float array ) =
  object (self)
    inherit ['Oin, float, 'Oout] identity x' as super
    val y_coeffs = y_coeffs'
    val max_output_delay = (Array.length y_coeffs')

    method kernel n = (* could be more efficient... *)
      (Array.fold_right (+.)
        (Array.mapi
          (fun n' e ->
             (if (n-(n'+1)) < 0 then 0.0 else output#get (n-(n'+1))) *.
e)
               y_coeffs)
        (x#get n)
      )

    method update_lock () =
      x#lock (self) (GreaterThanEq (n));
      output#lock (self) (GreaterThanEq (-max_output_delay))
    method register_inputs () =
      output#add_locker (self); (* need to rely on output values *)
      super#register_inputs ()
end;;
```

Signal.ml

```
open Observer;;
open Intervals;;
open Lock;;

signal_basic simply allows definition of arbitrary length signals,
limited by
  the *int* domain data type.
signal uses a lock to allow observers to lock parts of the signal.
purge will
  delete any parts of the signal not locked.

A signal is an infinite stream towards positive and negative infinity.
Data can be defined at any point of the signal.
Data is stored by an associative array with an int index
The signal class assumes that signals will be defined from left to
right and
  that once an observer to the signal releases a lock on the signal it
will not
  access the signal again.  It would be better to include a mechanism
to track
  erased sections of the signal.
If the value is defined, it is returned, else the Undefined_value
exception is
  thrown.
Undefined_value is to be the mechanism which allows calling routines to
  operate: an attempted calculation will fail if any value is Undefined
and
```

```
  then the calculation will wait and try again when new values are
defined.

  (More efficiency can later be added, such as
  an efficient abstract interval representation of defined values so a
check
  can be performed independent of the lookup.  Then a signal observer
can do
  a definition check before initiating a series of complicated
lookups.)

The data is remembered until it is explicitly erased.  An abstract
interval
  describes the erasure.  This interval is or'ed with the previous
erasure to
  determine the new erasure.  Once a value is erased, it is considered
defined
  but lost and an Erased exception will be thrown.  In all cases, this
  exception should be considered an error.
*)

exception Undefined_value;;
exception Redefinition of string;;
exception Error;;

type domain = int;;

(* <signal_basic> allows definition of a <signal> point-by-point using
<define>
or as a list using <define_list>.  The storage is an associative array
with
domain element (integers) and abstract output type.  Assoc. arrays may
not be the best implementation, but it is easy to change later if
needed.

<signal_basic> is derived from <subject> so that <observers> can
register for
notification when data is defined.  This is effectively a push
mechanism.

Data is defined and cannot be overwritten, hence the exception
<Redefinition>.
When each data point is defined, the observers are notified.
Inefficient, and
a buffering scheme may be better.  Also, the indices will run out
eventually,
so an index shifting may be necessary.

<get> will return a value if it is defined or will raise
<Undefined_value>.
*)

class ['O, 'T] signal_basic =
  object (self)
    inherit ['O] subject
    val mutable data : (domain * 'T) list = []
```

```
    method private avail n = List.mem_assoc n data

    method define (d : 'T) (n : domain) =
      if (self#avail n) then raise (Redefinition "Attempt to redefine
value: prohibited")
      else data <- (n,d)::data;  self#notify

    method define_list (l : (domain * 'T) list) =
      List.iter (fun li -> match li with (n,d) -> self#define d n) l

    method get (n : domain) =
      if (self#avail n) then List.assoc n data
      else raise Undefined_value

    method get_defined = data;

    method reset =
      data <- [];
end;;

(* causal signal will return zero for any index less than zero *)
class ['O, 'T] causal_signal_basic (zero' : 'T)=
  object (self)
    inherit ['O, 'T] signal_basic as super
    val zero = zero'
    method get (n : domain) =
      if (n < 0) then zero
      else super#get n
end;;

(* locked signal: when registering as an observer, set the lock region
to the
values that your observer will need in the future.  Using the interval
type,
infinitely long intervals of arbitrary complexity can be described *)

class ['O,'T] signal =
  object (self)
    inherit ['O,'T] signal_basic as super
    inherit [domain,'O] Lock.domain_lock as domain_lock

    method add_observer (obs : 'O) =
      super#add_observer obs;
      self#add_locker obs

    method lock (i : 'O) (l : domain Intervals.t) =
      domain_lock#lock i l;
      self#purge

    method purge = data <- List.filter ( fun x -> self#locked (fst x) )
data

    method reset = super#reset;  domain_lock#clear
end;;
```

Lock.ml

```
open Intervals

exception DuplicateLocker
class ['D,'I] domain_lock =
  object (self)
    val mutable lock : 'D Intervals.t array = Array.make 1 None
    val mutable id : ('I*int) list = []

    method add_locker (i : 'I) =
      if (List.mem_assoc i id) then
        raise DuplicateLocker
      else
      lock <- Array.append lock (Array.make 1 Intervals.None);
      id <- (i, (Array.length lock)-1)::id

    method lock (i : 'I) (d : 'D Intervals.t) = lock.(List.assoc i id)
<- d
    method locked (n : 'D) =
      (  Array.fold_left (fun f1 f2 -> (fun n -> f1 n || f2 n))
      (fun n -> false) (Array.map Intervals.inside lock)  )  n
    method clear = lock <- Array.make 1 None; id <- []
end;;
```

Intervals.ml

```
(* Intervals *)
type 'a t =
  None
  | All

  | Element of 'a
  | LessThan of 'a
  | GreaterThan of 'a
  | LessThanEq of 'a
  | GreaterThanEq of 'a

  | Outside of 'a*'a
  | OutsideEq of 'a * 'a
  | Between of 'a*'a
  | BetweenEq of 'a*'a

  | And of ('a t) * ('a t)
  | Or of ('a t) * ('a t)
  | Inverse of ('a t)

(* Delta Intervals
  | JustLessThan of 'a * 'a dt
  | JustGreaterThan of 'a * 'a dt
  type 'a dt = 'a
*)
```

```
let rec inside (i : 'a t) =
  match i with
    None -> fun x -> false
    | All -> fun x -> true
    | Element(a) -> fun x -> x = a

    | LessThan(m) -> fun x -> x < m
    | GreaterThan(m) -> fun x -> x > m

    | LessThanEq(m) -> fun x -> x <= m
    | GreaterThanEq(m) -> fun x -> x >= m

    | Outside(m1,m2) -> fun x -> x < m1 || x > m2
    | OutsideEq(m1,m2) -> fun x -> x <= m1 || x >= m2
    | Between(m1,m2) -> fun x -> x > m1 && x < m2
    | BetweenEq(m1,m2) -> fun x -> x >= m1 && x <= m2

    | And(a,b) -> fun x -> ((inside a) x) && ((inside b) x)
    | Or(a,b)  -> fun x -> ((inside a) x) || ((inside b) x)
    | Inverse(a) -> fun x -> not ((inside a) x)
```

Observer.ml

```
(* The observer design pattern, modified from
 Développement d'applications avec Objective Caml by Emmanuel
Chailloux, Pascal Manoury and Bruno Pagano, published by O'Reilly
France
*)

class ['O] subject =
  object (self)
    val mutable observers : 'O list = []

    method add_observer (obs : 'O) =
      observers <- obs::observers;

    method notify =
      List.iter (fun obs -> obs#notify_at) observers
end;;

class observer =
  object
    method notify_at = ()
end;;
```

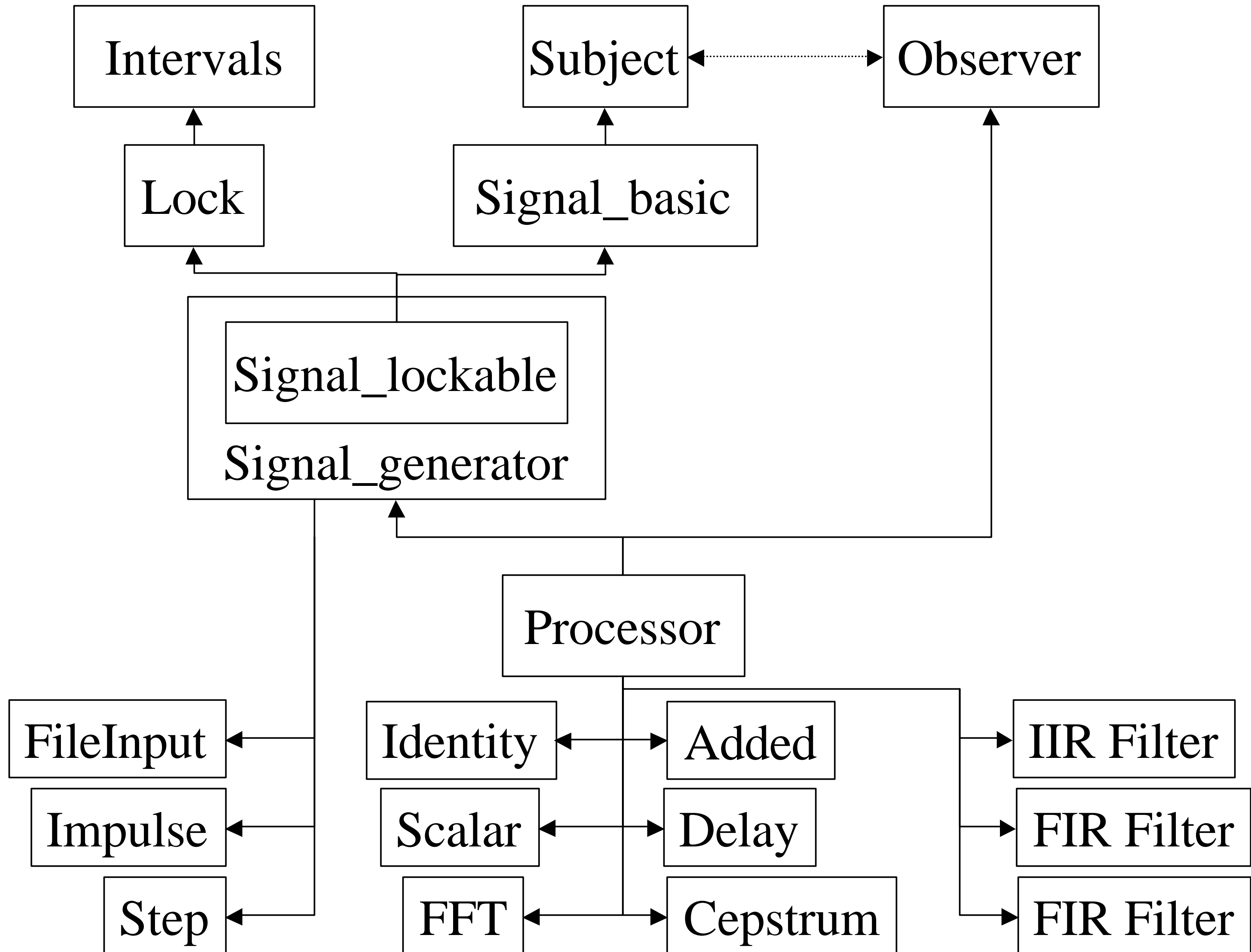
Ocaml Signal Processing

Jay B. Hancock

Main products.

- A type-checkable system of signals and processor units using O'Caml.
- A "push" model for streaming via interconnect using observer design pattern.
- An architecture for signal processing that allows easily defined components, derived from a processor base class.
- Use of O'Caml's pattern matching to define abstract intervals for use in the observer's locking mechamism of stream data
- Generic stream can handle floats for calculation, or symbols for speech recognition

Inheritance Diagram



Signal_basic Class

```
class ['O, 'T] signal_basic =
    object (self)
      inherit ['O] subject
      val mutable data : (domain * 'T) list = []

      method private avail n = List.mem_assoc n data
      method define (d : 'T) (n : domain) =
        if (self#avail n) then raise Redefinition
        else data <- (n,d)::data;  self#notify

      method get (n : domain) =
        if (self#avail n) then List.assoc n data
        else raise Undefined_value

      method reset =
        data <- [];
  end;;
```

Signal Class

- ```
  class ['O,'T] signal =
    object (self)
      inherit ['O,'T] signal_basic as super
      inherit [domain,'O] Lock.dlock as domain_lock

      method add_observer (obs : 'O) =
        super#add_observer obs;
        self#add_locker obs

      method lock (i : 'O) (l : domain Intervals.t) =
        domain_lock#lock i l;
        self#purge

      method purge = data <- List.filter ( fun x ->
  self#locked (fst x) ) data

      method reset = super#reset;  domain_lock#clear
  end;;
  ```
  ```

Generator Class

```
class virtual ['O, 'T] generator (buf : int) =
    object (self)
      val mutable output : ('O,'T) signal = new signal

      val mutable n : int = 0
      val mutable buf_len = buf
      val mutable count : int = 0

      method virtual kernel : int -> 'T

      method output =
        try for i = 1 to buf_len do
          n <- succ n;
          output#define (self#kernel (n-1)) (n-1)
         done
        with
          Undefined_value -> n <- pred n

      method y = output
      method get m = output#get m (*  called like a
  signal *)
  end;;
```

Processor Class

```
class virtual ['O, 'T] processor (buf : int) =
    object (self)
      inherit observer
      inherit ['O, 'T] generator buf as generator

      initializer self#init ()

      method init () =
        self#register_inputs ();
        self#update_lock ();
        generator#output

      method virtual register_inputs : unit -> unit
      method virtual update_lock : unit -> unit
        (* also virtual kernel : int -> ‘T *)
      method notify_at =
        generator#output;
        self#update_lock ()
  end;;
```

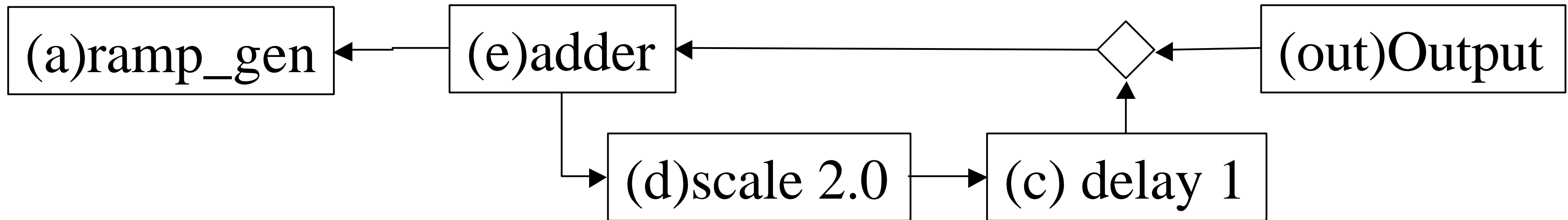
Example Processors

```
class ['Oout] step_generator (a' : float) =
    object (self)
      inherit ['Oout, float] generator 10
      method kernel (n:int) =
        if (n>=0) then a' else 0.0
  end;;

class ['Oin, 'T, 'Oout] identity (x' : ('Oin,'T)
  signal) =
    object (self)
      inherit ['Oout, 'T] processor 10 as super
      val mutable x = x'

      method register_inputs () = x#add_observer(self)
      method kernel n = x#get n
      method update_lock () =
        x#lock (self) (GreaterThanEq n)
  end;;
```

High-level code




```
let a = new ramp_generator 1.0 ;;
let e = new adder ;;
let out = new identity e#y;;
let c = new delay e#y 1 0.0 ;;
let d = new scale_float c#y 2.0 ;;
e#add_inputs [a#y;d#y];; (*dealing with loops*)

a#output;;  (*triggers 10 outputs from Step*)
```

- Upon init, each processor generates output points until it sees an undefined input, i.e. delay gives 0.
- Ideally, a GUI would allow the user to make a graph or the system and the code would then be generated.