Temporal type constructors for computer music programming

Eli Brandt

July 2002

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Roger Dannenberg, Chair
Guy Blelloch
Robert Harper
Perry Cook, Princeton University

Abstract

Existing high-level ways of doing computer music programming have sacrificed generality, making some algorithms impossible to express. My idea is that they fail due to type systems that are inadequate. I propose a way that is high-level without being restrictive: applicative programming with temporal type constructors.

Temporal type constructors are type constructors that introduce temporal structure. I propose a set of three—event, vector, and infinite vector—and give examples of the variety of musically useful types that they form. Temporal type constructors let the programmer describe exactly the data type that is key to a given algorithm, and manipulate it from a viewpoint outside of time.

I describe a programming system called Chronic that provides these temporal type constructors. Code examples written in Chronic show how some computer music algorithms can be expressed unprecedentedly well: several times shorter than their equivalent in a low-level language, and with a smaller semantic gap between the program and the idea—and these are algorithms that cannot be expressed at all in the high-level languages.

I then go into some detail about how Chronic is implemented, and why an infinite vector is not just a lazy list. I arrived at an infinite-vector implementation that does dynamic dataflow with a self-modifying graph of O'Caml objects, but the decisions along the way may be of interest to implementors facing different trade-offs. Interesting points include how to do block computation, efficient fan-out to multiple consumers, and in-place update. Vector comprehensions are necessary future work; they would give an order-of-magnitude speedup in some common cases. I explain why lazy vectors are a necessary step towards real-time operation, and why real-time remains a hard problem in Chronic's programming model, particularly due to its standpoint outside of time.

The novelty here is using type constructors to introduce temporal structure into the data, so that pure applicative programming becomes possible over the entire domain of computer music. This kind of programming is especially well suited to problems involving data with significant structure or multiple levels of abstraction. It has expressiveness from the construction of complex time-structured types suited to the problem at hand, and generality because every type can be deconstructed down to the individual audio sample. Many interesting computer music programs become easier to write and to read.

Thanks to Roger Dannenberg, my advisor these N years, for believing that music languages had another thesis in them, for help in shaping and thinking about it, and for indispensable comments on innumerable drafts of this document.

Thanks to those who did the work I'm following on to, and to all of the people who talked with me about this work, who gave me ideas and who asked questions.

Thanks to my family for being patient, to my friends for making it all fun, to `zephyr.el` for a window out of my interior office, to Pittsburgh for Tram's and Tessaro's and Frick Park on a fine day, and to Deanna for coming along for the ride.

He had been eight years upon a project for extracting sun-beams out of cucumbers, which were to be put into vials hermetically sealed, and let out to warm the air in raw inclement summers.
—Jonathan Swift, *Voyage to Laputa*

It is impossible to enjoy idling thoroughly unless one has plenty of work to do.
—Jerome K. Jerome, *Idle Thoughts of an Idle Fellow*

I am screaming out loud all the time I write and so is my brother which takes off my attention rather and I hope will excuse mistakes.
—Dickens, Fanny Squeers in *Nicholas Nickleby*

All this buttoning and unbuttoning.
—Anonymous (in 18th-century suicide note)

This is a rotten argument, but it should be good enough for their lordships on a hot summer afternoon.
—(Annotation to ministerial brief, said to have been read inadvertently in the House of Lords.)

Take care of the sense, and the sounds will take care of themselves.
—Lewis Carroll, *Alice's Adventures in Wonderland*

Contents

Chapter 1

Introduction

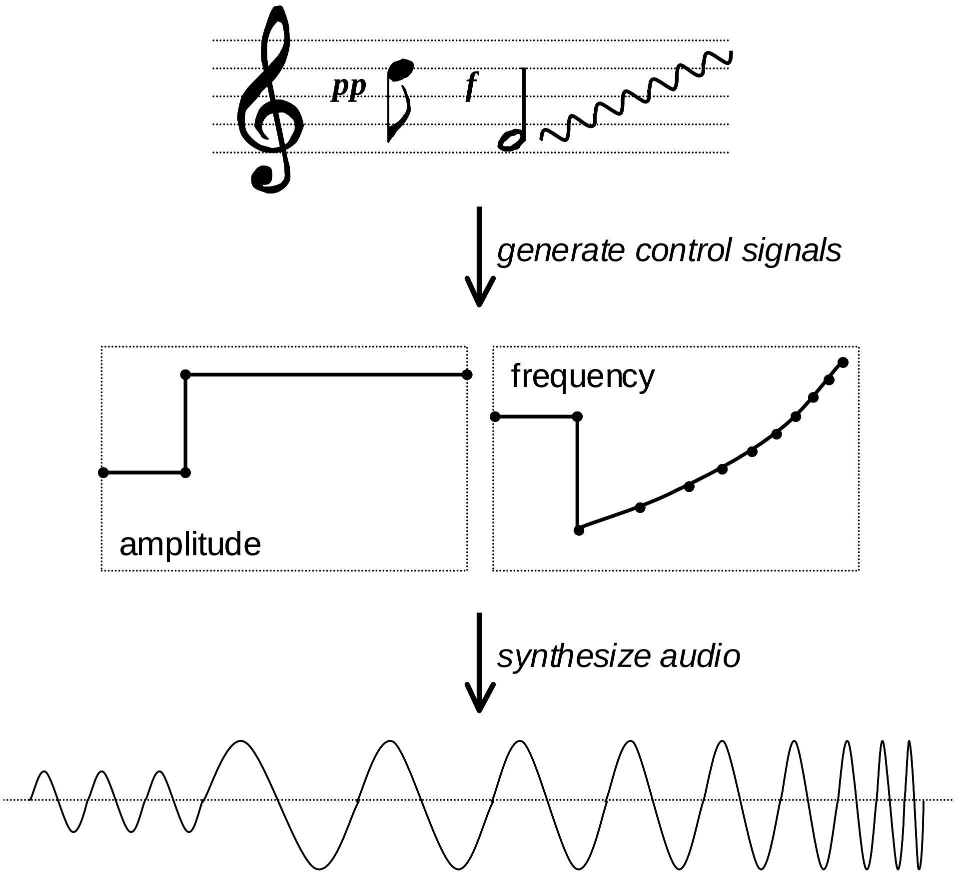
This dissertation is about how to do computer music programming in a manner that is both expressive and fully general. It explains this programming domain, and the tradeoff between expressiveness and generality that programmers presently face. An analysis of the tradeoff leads me to propose *temporal type constructors*.

1.1 Computer music

Computer music covers a variety of applications. It includes digital signal processing (see Appendix A for a minimal introduction to DSP), asynchronous event handling, and manipulation of data representing musical or sonic structure. This work addresses all of these subdomains, and is especially at home with applications where two or more come together and interact. Computer music software may be real-time or distributed, which this work does not address except in passing.

Perhaps the prototypical application is shown in Figure 1.1: start with a score of discrete note events, generate control signals from it, and use these to synthesize a digitized audio waveform. The synthesizer here is a simple sine oscillator. At each stage we are working with a different kind of data, only one of which is waveform audio.

Figure 1.1: A three-stage synthesis of audio from a score



On the other hand, we might go in the opposite direction: start with the audio, analyze it for frequency and amplitude envelopes, and abstract these into discrete note events. A wide variety of data types and transformations are used by common computer music applications:

- Building computer-music scores graphically and algorithmically from composer-defined score events, control gestures and diagrams, and nested assemblages of these; synthesizing audio from these scores.
- Techniques where audio is analyzed and then resynthesized from the analysis data, which may undergo transformations before resynthesis.
- Feature abstraction from audio signals, such as score transcription from audio.
- Signal processing that is adaptive to abstracted characteristics of the signal: speech effects sensitive to word or phoneme boundaries, drum-loop manipulation based on beat detection.
- Other signal processing that involves significant data structure: psychoacoustic compression (such as MPEG audio), granular synthesis, filterbanks, wavelet transforms, operations based on linear algebra.

- Speech processing with time warping and multiple levels of abstraction.

Our goal is to minimize the semantic gap between the algorithm and the implementation.

1.2 Goals in programming

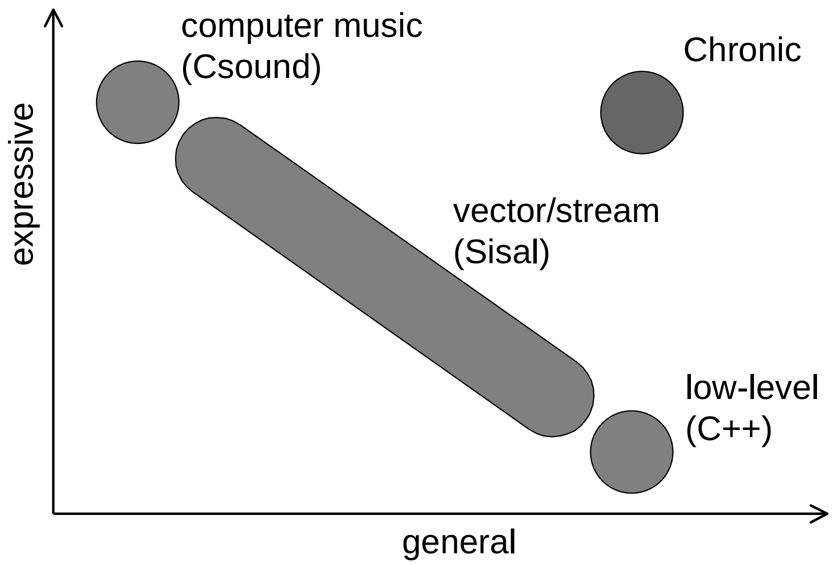
We would like a way of doing computer music programming that is both expressive and general. *Expressive* means that programs are easy to write and easy to read. *General* means that expressiveness is maintained over all or almost all of the programming domain. This is a more useful definition than requiring only that all programs be in some way representable: we exclude such loopholes as theoretical appeals to Turing-equivalence in a practically restrictive high-level language.

One can think of expressiveness as the maximum height achievable, and generality as the breadth of the plateau. Both are domain-relative. These metrics are rough, but they are sufficient to point out the problem with the state of the art.

1.3 The current tradeoff

The state of the art is that we can have expressiveness or generality, but not both. Existing computer music programming methods include low-level languages such as C++, specialized computer-music languages such as Csound, and a broad class of languages with vector or stream datatypes, such as Sisal or Matlab. Chapter 8 describes these classes of languages; here we just summarize their expressiveness and generality. Figure 1.2 plots them, sketching the tradeoff. *Chronic* is a prototype implementation of how I propose to move beyond this tradeoff.

Figure 1.2: The expressiveness vs. generality tradeoff



The computer-music languages vary widely, but share a model where certain signal-processing operations are provided as black-box primitives, "unit generators". The programmer wires them together so that audio signals flow between them. If a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language. In general, new DSP requires adding a new unit generator. New data types, such as new kinds of analysis data or control data, require adding an entire suite, because none of the existing ones recognize the new type.

Low-level languages have obvious difficulties with expressiveness. One might suppose a domain-specific library is the answer, but this tends merely to throw us back to the previous situation: library functions corresponding to unit generators, operating on vectors or synchronous-dataflow streams, leaving us with the same restriction on new DSP and new representations. One way to see this research is as an approach towards designing a different kind of library.

Languages placed in the vector and stream group, such as Sisal and Matlab, are diverse. Broadly, they have the generality of low-level languages, but they handle vectors (or streams, or occasionally both) better. Vector operations, including some DSP, can be expressed well. For other operations, particularly those involving events or more complex temporal structures, vector languages act low-level. Their extended bar on the chart reflects how they tend to permit expressiveness or generality, both in the same language, but not both in the same code.

1.4 Analysis of the problem

One way of looking at the problem here is that we want to merge the sublanguages present in existing computer music languages. Older languages have three: "score", "orchestra", and the underlying implementation language (C or Fortran). Respectively, these handle event scheduling, signal routing between unit generators, and the DSP inside unit generators. This division into sublanguages is widely seen as cumbersome, but was done for lack of any known alternative.

Newer languages have merged the score and orchestra into a single high-level language, but kept the low-level DSP language separate: an expressive language layered on top of a general one. This division of labor is the point of the unit generator model. Can we break it down and maintain both virtues?

What, in the specific domain of computer music programming, has been keeping us from the dual goal of expressiveness with generality? The problems seem to be in supporting new DSP and new musical representations.

The first problem is to be able to write arbitrary DSP in a high-level language, to avoid dropping down to C to write another unit generator. A key point here is that we need access to individual audio samples. The audio stream is a useful high-level concept, but it must not be made atomic.

The second problem is to be able to use new data representations. A new kind of data may be needed only for one specific algorithm, so introducing it must be easy, and so must be doable within the high-level language.

1.5 A solution

We need access to data types both below and above the level of the audio stream. Consequently, I propose to transcend the generality/expressiveness tradeoff by giving access to these types through *temporal type constructors*. Out of the most basic types, these build the time-structured types needed for the problem at hand, from "audio stream" to complex score structures. A prototype system, Chronic, enables programming by this method.

What exactly is it we're talking about here, "a method of programming"? Section 1.3 associated it with languages, but it is not a particular language (though Chronic is a domain-specific language, embedded in O'Caml). It is higher-level than a design pattern (perhaps an "architectural pattern", in Buschmann *et al.*'s usage (Buschmann et al. 1996)). It is a domain-specific style of programming, applicative programming with temporal type constructors, which can be embodied as a new language, or inside certain existing ones.

This dissertation will show that this style can make programming in our domain both comprehensive and unprecedentedly brief and clear. It demonstrates a prototype programming system embodying this style, and some programs written using this prototype. To give a taste of the results, Section 4.2 describes how Chronic works to implement a synthesis algorithm known as FOF. This algorithm cannot practically be realized in a unit-generator language, and a C version is long and has to make rather a mess of the algorithm's structure. The Chronic version is several times shorter, but more important is that it expresses the algorithm directly.

All too often a conceptually simple algorithm, rendered in an inappropriate language, turns into a morass of explicit bookkeeping, implicit assumptions, and unnecessary control flow. Algorithms often have a complex temporal structure—concisely expressible as operations on corresponding temporal types, if only we can make them available.

1.6 Other domains

While this work focuses on computer music and audio, it is not necessarily limited to that domain. Many programming domains need to combine discrete events and continuous signals. Examples include multimedia, computer games, real-time systems, robot control, telecommunications (especially rich-media devices such as PDAs), simulation, and scientific computing.

1.7 This document's structure

Do read Chapter 2. It presents the temporal type constructors, and shows how they build a remarkable gallery of musically relevant types, which is really the heart of this dissertation.

Chapter 9 summarizes and draws conclusions. To see whether you believe these conclusions, read the applications in Chapter 4, which are written in Chronic, using the proposed programming style. Understanding the code will depend on Chapter 3, explaining Chronic's interface, and perhaps Appendix B, briefly introducing O'Caml.

The chapters above talk primarily about the proposed style; very little has been specific to Chronic—mostly the concrete syntax of the code. Chapter 5, on Chronic's implementation, and Chapter 6, on the reasoning behind it, will be of interest to implementors of languages for signal processing. Chapter 7 explores possible extensions to Chronic—the ideas are general, but the discussion is specific to Chronic. Chapter 8 classifies and describes related work.

Chapter 2

Temporal types

The previous chapter suggested that music programming languages have problems with expressiveness and generality because they lack a system of time-structured types. This chapter describes a system which I propose. It uses type constructors to build up increasing complexity of time structure from scalar atomic types.

2.0.1 What is a temporal type?

A function of time, a signal sampled at a regular time interval, an event located in time—all of these values have some relation to a time axis. Their types are what we call "temporal types". Temporal types are used extensively in computer music and signal processing, and occur in other areas such as simulation and scientific computing.

2.0.2 What is a type constructor?

A type constructor builds complex types from simpler ones. For example, C has the "pointer to..." type constructor. We can write this as "α pointer", where α is a free type variable which might be, for example, int. Or α can be a non-atomic type, like "int pointer", leading to "int pointer pointer". In this way a single type constructor can build a series of types—and three, as we'll see, can cross-fertilize to build a great variety.

2.0.3 Motivation

The benefits of temporal type constructors are twofold. First, complex temporal types can be created to fit the problem at hand. Second, the existence of first-class temporal types improves code structure and allows the programmer a more powerful model of time. Programs become shorter, clearer, and probably more maintainable.

2.1 The three temporal type constructors

I propose using three temporal type constructors: event, vec, and ivec. An "α event" is an α at a single specified time. An "α vec" is a finite sequence of αs, and an "α ivec" is an infinite sequence of αs. In this work Time is a non-negative integer, interpreted as sample count, though other possibilities exist. Sampling rate is not specified (see Section 7.1.1).

α event	an event whose timestamp is of type Time.
α vec	a finite vector, indexed by Time.
α ivec	an infinite vector, indexed by Time.

These type constructors introduce a one-dimensional axis. Often this is a time axis, measured in sample count. Sections 7.6 and 7.1.1 describe other possibilities.

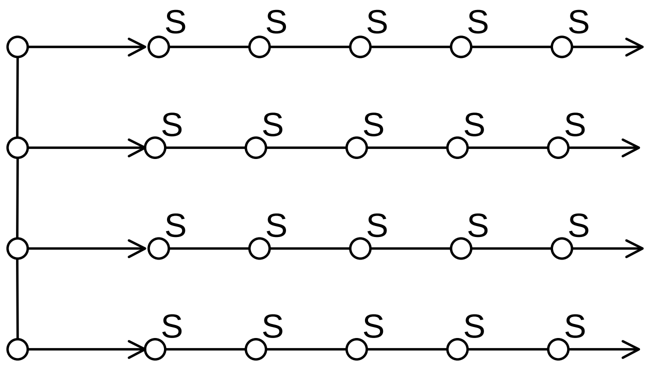
2.2 A cornucopia of types

Many useful temporal types arising from these three type constructors. A variety of examples follow; each example has some concept to be represented, a type that realizes it, and a diagram of the type's structure. These diagrams are just visual hints, not according to any formal scheme.

2.2.1 Multi-channel audio.

Multi-channel audio can be Sample ivec vec.

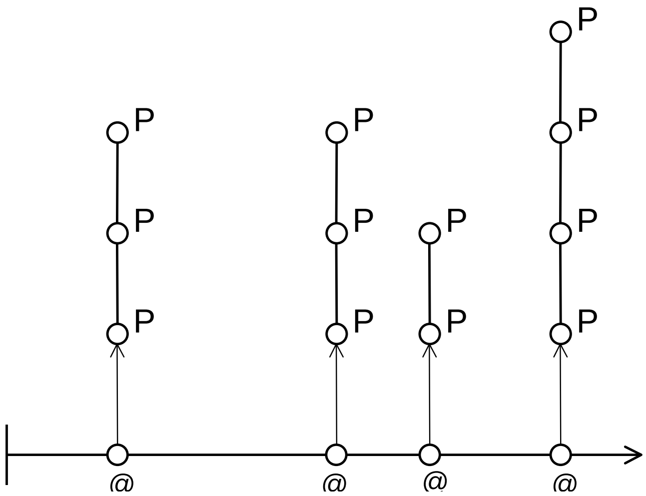
The precedence here is (Sample ivec) vec, so the type is read right-to-left: a vector of infinite vectors of samples. Sample is an audio sample; it could be 16-bit integer or floating point. (Multi-channel audio also has a transposed representation as Sample vec ivec which is convenient for certain manipulation.) Four-channel audio is pictured here. The number of channels is not encoded into this type.



Sample ivec vec

2.2.2 A chord progression.

A chord progression can be Pitch vec event vec.



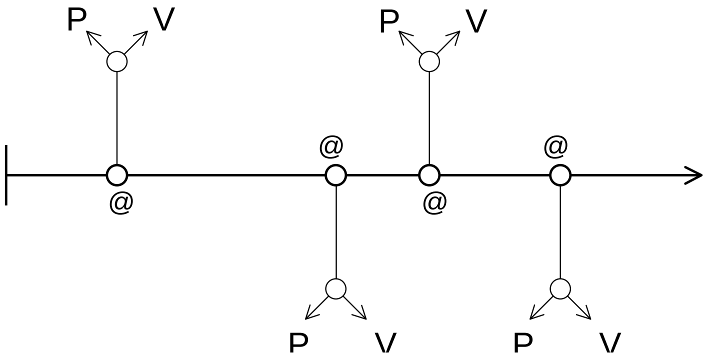
Pitch vec event vec

2.2.3 A MIDI-like score

A score with instantaneous note events, like MIDI's, is (Pitch×Velocity) event ivec.

Pitch indicates which note, and Velocity how hard it was struck. Each note is delimited by a note-on and a note-off; say an attack is represented by a positive Velocity, and a release by a negative one.

Each note event has a timestamp. These times might, depending on the way the score is generated and used, be non-decreasing. This would be enforced by run-time checks, not by this type system.

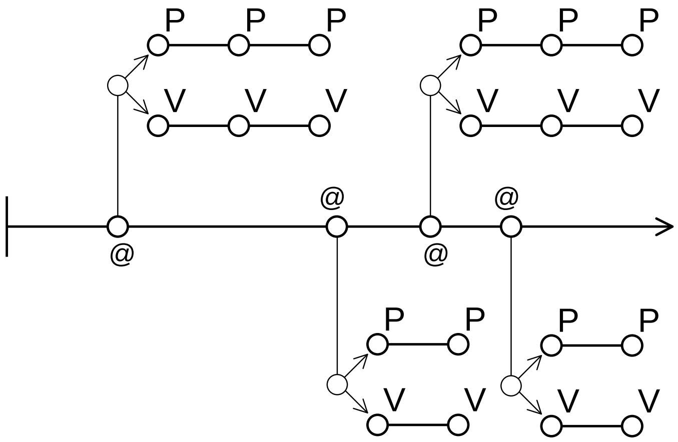


(Pitch×Velocity) event ivec

2.2.4 A non-MIDI score

A score suitable for instruments that are not like keyboards would use time-extended values: (Pitch vec × Pressure vec) event ivec.

These note events can represent continuous control gestures. Each one now encodes its own duration (as the length of the vecs), whereas the MIDI-like score used note-on/note-off. (We might use a vector of pairs rather than a pair of vectors, depending on the data's eventual use.)

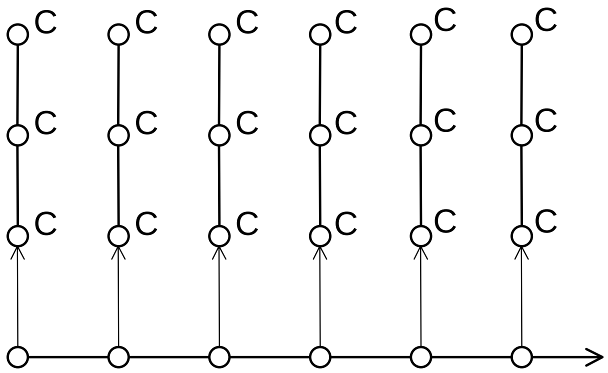


(Pitch vec × Pressure vec) event ivec

2.2.5 Short-time spectrum data

Data from a short-time Fourier transform could be Complex vec ivec.

Note that the vec has a frequency axis, but the ivec has a time axis.

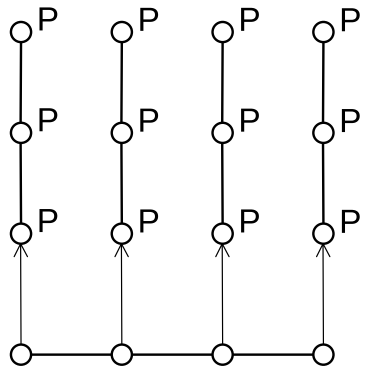


Complex vec ivec

2.2.6 A two-dimensional image

A two-dimensional image can be Pixel vec vec.

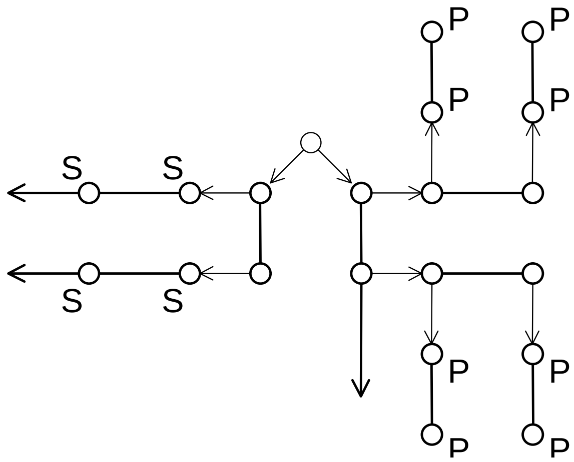
For a color image, Pixel has several components, and there is a another representation, transposed so each component gets an image plane.



Pixel vec vec

2.2.7 Video with audio

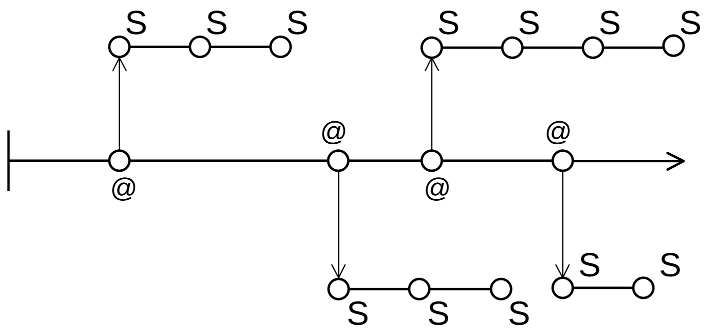
Time-extending the image with `ivec` gives a video type, and combining it with multi-channel audio results in
Pixel vec vec ivec × Sample ivec vec.



Pixel vec vec ivec × Sample ivec vec

2.2.8 Grains of sound

In granular synthesis, one uses a stream of timestamped segments of audio:
Sample vec event ivec.



Sample vec event ivec

This last type is a good example of synergy between vectors and events. It allows irregular segmentation of audio, with overlap, underlap, both, or neither. It can be used to cue and mix audio samples, or to represent sparse signals. It allows processing of selected pieces of a signal in different ways. A whole family of techniques, such as FOF vocal synthesis and formant-corrected pitch shifting, involve arranging small grains of sound.

2.3 Models of time

There are two general approaches to temporal data. One, made possible by Chronic's explicit temporal types, is strictly more powerful than the other. Informally, one allows code to refer to present, past, and future; the other allows only present and past. For example, both allow `delay`:

`IV.delay 2 [1 4 9 16 ...]` = `[0 0 1 4 9 ...]`

But one lacks what is roughly the inverse operation, `drop`:

`IV.drop 2 [0 0 1 4 9 ...]` = `[1 4 9 16 ...]`

In Csound, which uses the weaker model, the code to negate an audio stream is:

```
anegated = -ainput
```

The truth is that every value named here is a scalar; this scalar code is wrapped in a loop to form the actual program. Chronic uses actual time-extended values, and is explicit about the looping:

```
let (negated: int ivec) = IV.map (fun x -> -x) input
```

Csound, and the many languages like it, can perform *only* those stream operations that can be expressed by *mapping a scalar kernel across time.* This restriction is widespread, occurring wherever a language builds temporal programs from scalar programs rather than temporal types from scalar types.

We call Csound's model *SST*, and Chronic's *SSE*. (These abbreviate Spinoza's terms *sub specie temporis*, "under the aspect of time", and *sub specie æternitatis*, "under the aspect of infinity". Alternatively, Xenakis (1971) used the terms "in-time" and "out-of-time" in this sense.) Types that contain explicit temporal information make the *SSE* model possible. That is, rather than operating at a particular instant in time, one can step outside the timeline and operate on it as an eternal whole. From here one can readily step back to the *SST* programming model, by mapping a function across time.

The FOF code example in Section 4.2 shows the value of the *SSE* model, but let us look at a simpler example: processing a signal X to induce a 90-degree phase shift, and mixing this X_{90} back in with X to create X_θ with any desired phase shift.

What this "90-degree phase shift" actually *is* doesn't really matter here, but for the curious: a 90-degree phase shift delays each sinusoidal component independently by one quarter of its wavelength (note that no delay of the whole signal achieves this). We do it by convolution with a Hilbert-transformer filter, a vector resembling the plot shown in Figure 1. Each output sample of the convolution is the dot product of this vector, time-reversed, with a sliding window of the input.

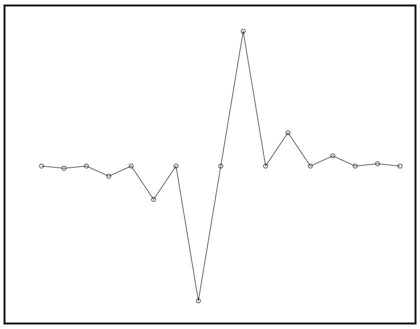


Figure 1: Hilbert filter

Implemented in the *SST* model, the sliding window is necessarily a record of the past, a delay line. Notice that the weight of the filter tends towards its middle, corresponding to data half the window into the past. This gives some intuition as to why the filtering delays the whole output by about half of the filter length, in addition to the desired quarter-wavelength phase shifts.

The 90-degree phase shift is implemented through convolution, which inevitably causes both the desired phase shift and some extra delay. The amount of extra delay, $T_\triangle$, is implementation-dependent, varying with the quality of the phase shift. An *SSE* phase shifter can cancel out this delay internally. An *SST* phase shifter cannot; code outside the phase shifter must expect the delay, know its length, and account for it: X and X_{90} must be re-aligned before they can be mixed to generate X_θ.

The *SST* approach is to delay X by $T_\triangle$, aligning it with X_{90}, before mixing them. In the following code, X is named `input`, X_{90} is `shift90_delayed`, and $T_\triangle$ is `tdelta`. `IV.delayz` performs the delay, and `IV.map2 (+.)` the mixing. In Chronic code:

```
let shift90_delayed input = LIV.convolve filter input
let shift_delayed theta input =
  let tdelta = ((V.length filter) - 1) / 2
  in IV.map2 (+.) (IV.scale (cos theta) (IV.delayz tdelta input))
                  (IV.scale (sin theta) (shift90_delayed input))
```

The resulting X_θ suffers from the same delay, so the same approach must be taken for any signal that *it* comes in contact with. If `shift90_delayed` changes the length of `filter`, this change ripples through the program.

The *SSE* solution is to 'anti-delay' the output, i.e. to drop T_Δ worth of data from the beginning. This can happen immediately after the convolution, inside `shift90`, maintaining proper abstraction:

```
let shift90 input = IV.drop ((length HFIR)-1)/2 (LIV.convolve filter input)
let shift theta input =
  IV.map2 (+.) (IV.scale (cos theta) input)
               (IV.scale (sin theta) (shift90 input))
```

Chapter 3

A system with temporal types

Chronic is a system for music programming, based on the temporal type constructors developed in the previous chapter. It is embedded in O'Caml, a strongly typed functional programming language with imperative and object-oriented features. Code written with Chronic is O'Caml code and follows O'Caml conventions; these are explained briefly in Appendix B and thoroughly in the O'Caml manual. (Leroy 2001)

This chapter gives an overview of the interfaces that Chronic presents to the programmer; Appendix C documents them fully. Chapter 5 describes the implementation behind the interfaces, while Chapter 6 explains the decisions that were made to arrive at Chronic as described.

Chronic has a core and a standard library. Each core module deals with one kind of temporal type. The standard library is a library of computer-music primitives. Each module is named according to the class of types that its functions deal with.

module	type
`E`	`event`
`V`	`vec`
`M`	`vec vec` (matrices)
`IV`	`ivec`
`EV`	`event vec`
`EIV`	`event ivec`
`L`	mostly scalars
`LV`	`vec`
`LIV`	`ivec`

3.1 An introduction to Chronic code

The following example creates an `ivec` of the perfect squares, selects the first five, and prints them. It shows how functions are accessed from Chronic modules and used within O'Caml.

```
(* the explicit type declarations are for readability -- not required *)
let (squares: int ivec) = IV.tabulate (fun x -> x*x)
let (five_squares: int vec) = IV.take 5 squares
```

```
let _ = V.print_int five_squares           (* prints 0, 1, 4, 9, 16. *)
```

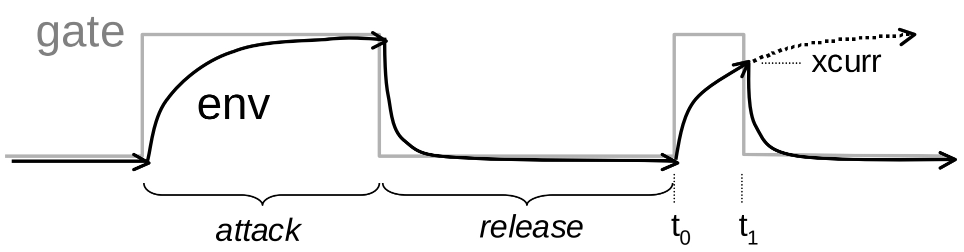
The `tabulate` and `take` functions are found not at the global level, but inside the `IV` module (as well as in the `V` module; this dual occurrence is why the modules exist). The syntax `IV.tabulate` refers to the `tabulate` function within the `IV` module.

`LIV`'s attack/release envelope generator is a more musically relevant example. It demonstrates how to define a function, and how a function can be partially applied.

```
(* attack and release are measured in number of samples to 90% *)
let env_ar (attack: float) (release: float) (gate: bool event ivec)
           (x0: float) =                    (* initial value, usually 0. *)
  let kfrac = 0.90  in
  let k_atk = kfrac ** (1. /. attack)  in    (* k_atk ** attack = kfrac *)
  let k_rel = kfrac ** (1. /. release)  in
  let relax k target x = (target -. k *. (target -. x))  in
  let fchain xcurr gate_event =
    let f = if (E.v gate_event)             (* examine value of event *)
      then (relax k_atk 1.)                 (* (partially apply the fn) *)
      else (relax k_rel 0.)  in
    IV.iterate f xcurr                      (* apply f repeatedly *)
  in EIV.chain fchain x0                    (* chain stages together *)
```

O'Caml appends dots to the floating-point operators, except for exponentiation.

The following diagram shows the input `gate` and output `env`, and marks the first attack/release cycle. At each `bool event` in `gate`, `fchain` generates an `ivec`. For example, at the `true` at time t_0, it generates the `ivec` starting at zero and extending along the dotted line. `EIV.chain` cuts this short at time t_1 when the `false` appears, notes the level at which it was cut short, and feeds this value as `xcurr` to its next `ivec`. In this way it pieces together a single continuous `ivec` output.



Although `relax` is a three-argument function, it is applied here to two arguments. This partial application evaluates to an anonymous one-argument function. In different terms, `relax` is used as a factory that generates functions of type `float -> float`, each of which exponentially decays its argument one step towards a specified target value — either zero or one, here, depending on whether the gate is high or low.

The `fchain` function iterates this decay-step function, generating a `float ivec` that asymptotically approaches the gate value. `EIV.chain`, documented in Section C.5, does the job of switching from one to the next of these `ivecs`, chaining the last value taken from the old `ivec` as the first of the new one.

The next example defines a simple synthesis patch using `LIV.env_ar`, and uses it to play a three-note score. The score format is entirely under the control of the programmer—it is whatever the synth patch accepts. This pervasive use of the sampling rate `fs` is inelegant; it could be avoided if the signal carried its own rate, as in Section 7.1.1.

```
let fs = 44100.                                      (* audio sampling rate *)
let synth (hz: float, duration: time) =
  let attack = 0.001 *. fs  in
  let release =  1.0 *. fs  in

  let sine = LIV.osci L.sinetab (hz /. fs) 0.  in
  let env = LIV.env_ar attack release [| true @@ 0; false @@ duration |] in
  let fulldur = duration + 2*(int release)  in    (* give it time to fall *)
  LV.declick (IV.take fulldur (IV.map ( *.) sine env))

let t = int (fs *. 0.5)
let score = [| (440., 3*t) @@ 0; (550., 2*t) @@ t; (660., t) @@ (2*t) |]
let audio = EV.mix (EV.map synth score)
```

For further and larger examples of Chronic code, see Chapter 4.

3.2 The event module

An α event, which in O'Caml is written "`'a event`", is a timestamped `'a`. The timestamp is of type `time`, which in the end is a synonym for `int`.

The most basic `event` functions are those to construct an event, and to take it apart. The following code example shows some of these. Notice that the infix operators `@@`, to construct an event, and `@+`, to increment its timestamp, are accessible without an `E.` qualifier. (Various operator and non-operator versions of these functions and others are listed in Section C.1.)

```
let fs = 44100.
let note_data = (440. /. fs, 1. *. fs)            (* say (freq, dur) *)
let note_time = truncate (2. *. fs)               (* two seconds *)
let note_event = note_data @@ note_time
   (* construct event from value and timestamp *)
let nudged_note_event = note_event @+ 1           (* one sample later *)
let same_note_data = E.v nudged_note_event
   (* extract value from event *)
let new_note_time = E.t nudged_note_event
   (* extract timestamp from event *)
```

Another example of a function in `E` is `E.lift`, which transforms a function operating on some data to one operating on events containing that data:

```
let modify (freq, dur) = (freq *. 2., dur /. 2.)
   (* modify: float * float -> float * float *)
```

```
let modify_event = E.lift modify
   (* modify_event: (float * float) event -> (float * float) event *)
let modifyd_event = (E.lift modify) note_event
   (* apply modify to the value part, preserving the timestamp *)
```

For a complete listing of core event functions, see Section C.1.

3.3 The **vec** module

An `'a vec` is a finite vector whose elements are of type `'a`. Indexing is by zero-based time. Elements can be addressed through O'Caml array syntax:

```
let x = V.tabulate (fun x -> x*x) 4    (* x is [| 0; 1; 4; 9 |] *)
let _ = (x.(2) <- 17)                  (* becomes [| 0; 1; 17; 9 |] *)
let x2 = x.(2)                         (* i.e. x.(2) is now 17 *)
```

The `V` module has the usual functions used for applicative programming on lists or vectors, a few of which are used here:

```
let squares = V.tabulate (fun i -> i*i) 5
   (* [| 0; 1; 4; 9; 16 |] *)
let pows_of_two = V.iterate (fun x -> 2*x) 1 5
   (* [| 1; 2; 4; 8; 16|] *)
let squares_mod_four = V.map (fun x -> x mod 4) squares
   (* [| 0; 1; 0; 1; 0 |] *)
let _ = V.print_ints squares_mod_four
   (* no output; prints "0 1 0 1 0" on stdout as side effect. *)
let odd_squares = V.filter (fun x -> x mod 2 = 1) squares
   (* [| 1; 9 |] *)
let three_squares = V.take 3 squares
   (* [| 0; 1; 4 |] *)
let three = V.length three_squares
   (* 3 *)
```

The `scan` function comes in several varieties, behaving differently at the endpoints:

```
let cumulative = V.scan (+) 100 pows_of_two
   (* [| 100; 101; 103; 107; 115 |], ignores final 16 *)
let cumulative' = V.scan' (+) 100 pows_of_two
   (* [| 101; 103; 107; 115; 131 |], does get to use the 16 *)
let cumulativex = V.scanx (+) 100 pows_of_two
   (* [| 100; 101; 103; 107; 115; 131 |], "eXtended" *)
```

It also comes extended to `sidescan`, a scan with side state, which proves useful for avoiding `zip`/`unzip` juggling:

```
let phasor (f: float vec) ph0 =
  let step ph f = (mod_float (ph +. f) 1.0, ph)
  in V.sidescan step ph0 f
let phasor_test = phasor [| 0.2; 0.2; 0.5; 0.2; 0.2 |] 0.
   (* [| 0.0; 0.2; 0.4; 0.9; 0.1 |] *)
```

For a complete listing of core vector functions, see Section C.2.

3.4 The `matrix` module

Chronic handles finite matrices as vectors of vectors. Matrix indexing is zero-based, corresponding to vector indexing, rather than one-based; the `matrix.(i).(j)` syntax works.

The `M.V` submodule has routines for finite matrices, `'a vec vec`. Other submodules could be added to handle the two kinds of half-infinite matrices, `'a ivec vec` and `'a vec ivec`, and perhaps fully infinite, `'a ivec ivec`.

The type system does not express whether a matrix is rectangular or ragged. Passing a ragged matrix to a function expecting a rectangular one will result in a run-time error.

Chronic provides a small selection of functions for I/O and for extracting vectors from matrices, but does not delve into linear algebra:

```
let mat = M.V.tabulate (fun i j -> 10*i + j) (2, 3)
   (* [| [|  0;  1;  2 |];
         [| 10; 11; 12 |] |] *)
let (rows, cols) = M.V.dims mat
   (* (2, 3) *)
let main_diag = M.V.diag 0 mat
   (* [| 0; 11 |] *)
```

For a complete listing of core matrix functions, see Section C.3.

3.5 The `ivec` module

An `'a ivec` is a infinite vector whose elements are of type `'a`. Indexing is by zero-based time, but the `x.(i)` syntax is not supported. There is no notation for immediate `ivecs` (like `[| 1; 2; 3 |]` for `vecs`).

Some `ivec` functions require an *exemplar* argument, an example of one output element. For example, `map trunc` converts a `float ivec` to an `int ivec`, so it needs an `int` exemplar—any `int`; 42 is as good as any other. Section 5.5.3 gives the technical reason for this awkwardness.

3.5.1 Applicative-programming functions

The `IV` module has many of the same functions as `V`, including the usual applicative-programming functions. In the following examples, notation like `[| 1; 2; 3; ... |]` is used informally to show results. The `map`, `scan`, `scan'`, and `sidescan` functions, among others, take exemplars.

```
let squares = IV.tabulate (fun i -> i*i)
   (* [| 0; 1; 4; 9; 16; ... |] *)
let pows_of_two = IV.iterate (fun x -> 2*x) 1
   (* [| 1; 2; 4; 8; 16; ... |] *)
let squares_mod_four = IV.map 0 (fun x -> x mod 4) squares
   (* [| 0; 1; 0; 1; 0; ... |] *)
let _ = IV.print_ints 5 5 squares_mod_four
   (* no output; prints "0 1 0 1 0" on stdout as side effect. *)
let (three_squares, other_squares) = IV.split 3 squares
   (* ([| 0; 1; 4 |], [| 9; 16; 25; ... |]) *)

let cumulative = IV.scan 0 (+) 100 pows_of_two
   (* [| 100; 101; 103; 107; 115; ... |] *)
let cumulative' = IV.scan' 0 (+) 100 pows_of_two
   (* [| 101; 103; 107; 115; 131; ... |] *)
let phasor (f: float ivec) ph0 =
  let step ph f = (mod_float (ph +. f) 1.0, ph)
  in IV.sidescan 0. step ph0 f
let phasor_test = phasor (IV.const 0.2) 0.5
   (* [| 0.5; 0.7; 0.9; 0.1; ... |] *)
```

Although `ivec`s do not support single-element indexing like `x.(i)`, one `ivec` can be indexed by another:

```
(* index squares.(pows_of_two) *)
let pows_of_four = IV.peekiv 0 pows_of_two squares
   (* [| 1; 4; 16; 64; ... |] *)
```

3.5.2 Delay and feedback delay

The `delay` family of functions handles not only fixed- and variable-length delays, but also delays with feedback, and with arbitrary operations in the feedback path. Delay means referencing earlier samples of the source `ivec`, possibly samples before time zero: in the following no-feedback examples, the `[| -3.; -2.; -1. |]` vector is the "basis" which gives this information, or `delayz` assumes an all-zero basis.

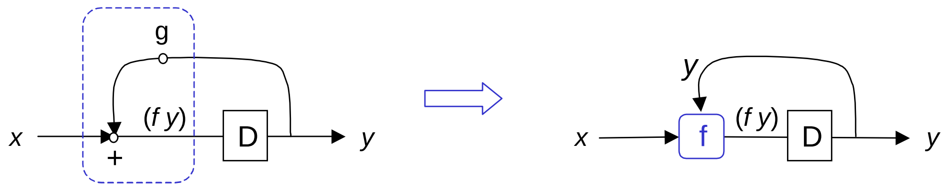
```
let count = IV.tabulate (fun x -> float_of_int x)
   (* [|  0.;  1.;  2.;  3.;  4.;  5.; ... |] *)
let basis = [| -3.; -2.; -1. |]

let delay_count = IV.delay 3 basis count
   (* [| -3.; -2.; -1.;  0.;  1.;  2.; ... |] *)
let delayz_count = IV.delayz 3 count
   (* [|  0.;  0.;  0.;  0.;  1.;  2.; ... |] *)

let delay_by = IV.join (V.const 3 5) (IV.const 0)
```

```
   (* [|   3;   3;   3;   3;   3;   0;   0;   0; ... |] *)
let variable_delay = IV.vdelay delay_by basis count
   (* [| -3.; -2.; -1.;  0.;  1.;  5.;  6.;  7.; ... |] *)
```

Feedback delay is used, for example, in making a repeating echo. We would like to realize the data flow shown on the left below, so that the delayed `y` feeds back to mix with the input `x`. We could describe `y` recursively:

```
(* given echo_len, echo_gain *)
let f xi yi = xi +. echo_gain *. yi
y = IV.delayz echo_len (IV.map2 f x y)
```

But this recursive definition is not legal in O'Caml.



Instead, we think in terms of the diagram on the right, encapsulating `f`. `IV.delayz_rec2` constructs that diagram and plugs in the `f` we give it:

```
(* given echo_len, echo_gain *)
let f xi yi = xi +. echo_gain *. yi
y = IV.delayz_rec2 echo_len f x y
```

The "`rec`" in the name indicates that it builds recursive flow graphs, and the "`2`" that its `f` takes two inputs. Section C.4 lists all of the various flavors of delay functions that Chronic provides.

Recursive (IIR) filters are an important special case of feedback delay. The `IV` module provides first- and second-order filter primitives that are more efficient than using `delay_rec2`.

3.5.3 Other functions

Like `vecs`, `ivecs` can be read from and written to text files. `LIV.write_wav`, which writes to a `.wav` audio file, may also be useful.

See Section C.4 for a complete listing of `ivec` functions.

3.6 The **event ivec** module

The `IV` module is for operations on `ivecs` of arbitrary element type. Functions in the `EIV` module require that the `ivec` elements be events, and, furthermore, that they be time-ordered. That is, their timestamps must not decrease along the `ivec`. This monotonicity constraint is not expressed in the data type, so it is not checked at compile time; violating it produces undefined results.

A time-ordered `'a event ivec` can be interpreted as a sparse representation of an `'a ivec`. The question is how to interpolate between the data points. `EIV` supports zero- and first-order interpolation, so the output is piecewise constant, from `EIV.pwc`, or piecewise linear, from `EIV.pwl`. These functions also take an initial value—`3.` in the following examples—to provide data at time zero in case the first event doesn't come until later.

```
let events = IV.tabulate (fun i -> (float_of_int i) @@ (1 lsl (i+1)) )
   (* [| 0. @@ 2;  1. @@ 4;  2. @@ 8;  3. @@ 16; ... |] *)

let pwc = EIV.pwc 3. events
   (* [| 3.; 3.; 0.; 0.; 1.; 1.; 1.; 1.; 2.; 2.; ...|] *)
   (*            2      4               8            *)

let pwl = EIV.pwl 3. events
   (* [| 3.; 1.5; 0.; 0.5; 1.; 1.25; 1.5; 1.75; 2.; 2.125; ... |] *)
   (*             2       4                   8                *)
```

The `EIV.chain` function was used in the envelope generator in Section 3.1. It provides something like `IV.iterate`, updated with information from each event. For example, we might perform iterated multiplication by a gain factor, and update the gain:

```
let events = IV.join [| 0.1 @@ 0; 2. @@ 2 |] (IV.const (1. @@ time_max))
let f x_curr gain = IV.iterate (fun x -> gain*.x) x_curr
let chain = EIV.chain f 1. events
   (* [| 1.; 0.1; 0.01; 0.02; 0.04; ...|] *)
   (*     0         2                    *)
```

Finally, the `EIV.vfold` function deals specifically with types of the form `'a vec event ivec`. It treats the vectors as fragments which may leave gaps or overlap, determines the list of vector elements overlapping for each output sample, and folds the list by a user-provided function. The fold is a `foldl` on the list in old-to-new order. An example may make this clearer:

```
let mixme = IV.tabulate (fun n -> Event (n*n, V.const n 4))
   (* [| [| 0; 0; 0; 0 |] @@ 0;
            [| 1; 1; 1; 1 |] @@ 1;
                     [| 2; 2; 2; 2 |] @@ 4;
                                 [| 3; 3; 3; 3 |] @@ 9;
                                                     ... |] *)
   (* t =   0  1        4              9 ...   *)

let mixed = EIV.vfold (+) 0 mixme
   (*    [| 0; 1; 1; 1; 3; 2; 2; 2; 0; 3 |] *)
   (* t =   0  1        4              9 ...   *)
```

`EIV.vmix` is a specialized version of `vfold` for the common case where the folding function is floating-point addition.

See Section C.5 for a complete listing of `event ivec` functions.

3.7 The **event vec** module

The `EV` module is the `vec` counterpart to `EIV`, and has the same requirement of time-ordering. It has similar `pwc` and `pwl` functions, with one additional complication: what happens when a finite vector ends. If the time of the right endpoint is t_{end}, do we treat the endpoint as open, and generate output for $t < t_{\text{end}}$, or as closed, and generate for $t \leq t_{\text{end}}$? `EV` chooses the former, and provides `EV.close` to get the latter:

```
let events = [| 1 @@ 2; 2 @@ 4 |]
let open_pwc = EV.pwc 3 events
   (* [| 3; 3; 1; 1 |] *)
let closed_pwc = EV.pwc 3 (EV.close events)
   (* [| 3; 3; 1; 1; 2|] *)
```

The `chain` and `vfold` functions are not implemented for `EV`. See Section C.6 for a complete listing of `event vec` functions.

3.8 The standard library

The `L` module is a library of DSP and computer music primitives. Some of its functions occur both in vector and in `ivec` versions; for technical reasons (`L.V` would want to refer to both `IV` and `L.IV`, but one shadows the other), these are organized into sibling modules `LV` and `LIV`.

The library includes musical unit conversions, table generation and lookup, window functions, oscillators, filters, and envelope generation. It is, however, nowhere near as fully populated as it would need to be for actual use. See Section C.7 for a complete listing of library functions.

Chapter 4

Evaluation on several applications

How do we judge whether Chronic's style of programming is as general and expressive as I am claiming it is? In the absence of a formal methodology recognized as valid for making this judgement, I will present several computer-music applications implemented in Chronic, and evaluate expressiveness according to the functional factoring of the code, its line count in some cases, and most importantly the size of the semantic gap between the code and our idea of the algorithm.

Generality is more cut-and-dried. Of all the applications, only the reverberator of Section 4.3 can be implemented in Csound, a MUSIC-N computer music language. A modern computer music language such as Nyquist or SuperCollider can also handle scores roughly as in Section 4.1 (but neither of these languages are statically typed). The remaining applications involve writing DSP, which is what traditional computer music languages are unable to do.

4.1 Musical scores

Western music has a long history of formal notation, paralleling and predating algebraic notation. The notes-on-staves system we call common music notation dates in outline from the eleventh century, and reached roughly its present form in the sixteenth and seventeenth. In the twentieth century, though, the abstraction that had made this notation so useful began to pinch. Composers wanted new pitches and rhythms, and they put far more emphasis on timbre; they even broadened the scope of composition to include chance, interactivity, and branching paths.

As scores moved beyond what could be written on paper and beyond what human instrumentalists could perform, many composers turned to electronics and then to computers to render their ideas. The score became either a list of software-interpretable note data, or an entire piece of software generating sound.

A score as a program is more powerful, because it can use the full expressive power of a programming language. A score as data is more manipulable, because operations can be performed on it once its interpretation as sound is understood. No specific notion of a score is hardwired into Chronic, but you can build, manipulate, and synthesize from any time-structured data format you choose, from flat data to full-blown program.

4.1.1 Expressiveness without the performer

A human performer playing a score lends expressiveness to the performance: through interpreting notations such as slurs and crescendos in the score, through performance practice left tacit but expected by the score, and through personal choices of how to portray the music's structure. The expression is through fine variations in note timing, loudness, shape, and other characteristics.

When the score is not to be interpreted by a human performer, it has to contain all of this expression itself, explicitly notated. It has to apply a repeating pattern of emphasis, through loudness and brightness and timing, to give the metric structure. It has to be explicit about the meaning of a crescendo, and the shaping of a phrase.

The score's control parameters may be scalars attached to notes, for timing deviation. They may be signals varying over the course of the note, to shape a swell or a trailing away. They may span multiple notes, indicating the tempo for an accelerando. Spans may overlap, may stack hierarchically and combine.

We will see how Chronic can support all of these kinds of expressive control. A musical phrase can be an event in a score, containing note sub-events and having a control signal for the shape of the phrase: it can be a `(`*`note`*` event vec * `*`emphasis`*` vec) event`.

4.1.2 Scores as data: note lists

In the simplest case of score as data, the score is a flat *note list*. "Note" here may be generalized beyond its usual sense, to mean anything from a millisecond-long grain of sound to a parameter update on a process that has been running for hours. Typically each note event has a starting time, a duration if that makes sense and is known in advance, and zero or more parameters defining the event.

The following example shows one way to build and synthesize from a note list in Chronic. The note list is a `vec` of note events, each containing a decibel level, a frequency, and a duration, and having a start time attached. Here the note list is entered by hand; the `makenote` function saves some boilerplate typing. The `synth_beep` function synthesizes a `float vec` from a note. `E.lift` makes it synthesize a `float vec event` from a note event, preserving the event timestamp; this lifted function can be mapped over the score. We then mix down the result.

This toy example could be extended by replacing `synth_beep` with a more interesting synthesis function, and extending the parameter list to give the composer more control over it. Section 4.1.3 will be a step in that direction.

```
(* score example with sine beeps *)

open Vec
open Event
open Evvec
open Chronlib
open Util

let fs = 44100.
```

```
let secs_to_samples s = int (fs *. s)

(*** handwritten score of note events *)
let makenote time_secs db hz dur_secs =     (* save a little typing *)
  (db, V.const hz (secs_to_samples dur_secs)) @@ (secs_to_samples time_secs)

(* score is (db: float, hz: float vec) event vec *)
let score = [|
       (*  time dB     Hz   dur  *)
  makenote 0.  (-.3.)  400. 0.5;
  makenote 1.  (-.10.) 400. 0.5;
  makenote 1.5 (-.10.) 500. 0.5;
  makenote 2.  (-.10.) 600. 0.5
|]

(*** define synthesizer of sine beeps *)
let sinetab = LV.table_of_sine 4096
(* synth_beep: (float * float vec) -> float vec *)
let synth_beep (db, hz) =    (* dur is implicit in length of hz vector *)
  let freq = V.scale (1./.fs) hz  in
  let beep = LV.osci_v sinetab freq 0.
  in V.scale (L.db_to_amp db) beep

(*** synthesize, mix and output *)
let sound = EV.vmix (V.map (E.lift synth_beep) score)
let _ = V.print_floats sound
```

4.1.2.1 MIDI scores

MIDI is a common format for scores as data. A Type-0 Standard MIDI File is a different kind of note list from the previous example, in that its note events do not contain durations; instead, each note is bracketed by note-on and note-off events. We could either determine durations and convert to the previous format, or synthesize directly from the note-on/note-off format.

To synthesize from a MIDI file would require a library routine to parse the MIDI file format and translate it to Chronic data, either with durations or without. The user's program would synthesize from that data.

4.1.3 Scores as programs: a generated score and a scoped LFO

Two advantages of using programs as scores are that scores can be generated by computation, and can use language structure such as local variables and scoping. In the next example, the notes are generated by upward arpeggiation of a chord. An LFO to control filter cutoff runs freely alongside this note event vector; `incorporate`'s `add_cutoff` function samples the LFO at the onset of each note and incorporates that value into the note. This forms the final score. (It would be easy for `add_cutoff` to extract a segment instead of a single value, so the cutoff would vary over the course of the note, but the time-varying filter primitive to render this is not implemented.)

Although simple, this example touches on a classic problem in computer music: how to scope and integrate multiple non-synchronous streams of score data. The MUSIC-N languages have exactly two levels of scoping: global to the entire score, and local to each note instance. In one of those languages, our filter-cutoff LFO would have to be made global. A complex score can easily devolve into global spaghetti. An LFO local to the arpeggio is cleaner, and permits multiple instantiation of the arpeggio/LFO combination.

Here `synth_beep`'s oscillator uses a bandlimited sawtooth table, to give some harmonics to filter. The filter is a two-pole lowpass with the note's specified cutoff frequency, and a Q of 5. Finally, `LV.declick` applies a fast linear attack/release envelope.

```
(* arpeggiated saw beeps filtered by an LFO scoped to the arpeggio *)

open Vec
open Event
open Evvec
open Chronlib
open Util

let fs = 44100.
let secs_to_samples s = int (fs *. s)

(*** create score with LFO running alongside arpeggio *)
let arpeggio =
  (** generate note events by arpeggiating a chord *)
  let makenote time_secs db hz dur_secs =
    (db, V.const hz (secs_to_samples dur_secs))
     @@ (secs_to_samples time_secs)  in

  let arpeggiate chord notedur reps =
    let hz = V.ravel2 (V.const chord reps)  in
    let n = V.length hz  in
    let time = V.count_by 0. notedur n
    in V.map2 (fun hz time -> makenote time 0. hz (0.9*.notedur)) hz time  in

  let chord = [| 200.; 250.; 300.; 400. |]  in
  let notedur = 0.125  in
  let reps = 8  in
  let dur = notedur *. (float reps) *. (float (V.length chord))  in
  (* notes: (db: float, hz: float vec) event vec *)
  let notes = arpeggiate chord notedur reps  in

  (** filter-cutoff LFO *)
  let lfo = LV.osc_sine_slow (secs_to_samples dur) (1. /. dur /. fs) 0.  in
  let fc_oct = V.map (fun x -> 2. *. x +. 7.) lfo
```

```
  (** arpeggio is notes packaged with filter cutoff *)
  in (notes, fc_oct)

(* incorporate filter cutoff into score events *)
let incorporate (notes, fc_oct) =
  let add_cutoff (Event (t, (db, hz))) =
    Event (t, (db, hz, fc_oct.(t)))
  in V.map add_cutoff notes

let arpeggio_score = incorporate arpeggio

(*** synthesize filtered saw beeps *)
let nharmonics = int (fs /. 500.)
let saw_harmonics = V.map (fun x -> 0.667 /. x) (V.count_by 1. 1. nharmonics)
let sawtab = LV.table_of_sine_harmonics saw_harmonics 4096

let synth_beep (db, hz, fc_oct) =
  let freq = V.scale (1./.fs) hz  in
  let beep = LV.osci_v sawtab freq 0.  in
  let filtered = LV.lowpass2 ((L.oct_to_hz fc_oct) /. fs) 5. beep
  in V.scale (L.db_to_amp db) (LV.declick filtered)

(*** mix and output *)
let sound = V.scale (L.db_to_amp (-.18.))
    (EV.vmix (V.map (E.lift synth_beep) arpeggio_score))

let _ = V.print_floats sound
```

4.1.4 Further possibilities

These score examples only scratch the surface of what a composer might like to do. I will briefly describe further possibilities, hoping that the complete examples above give insight into how the following incomplete ones could be fleshed out. User-defined values and Chronic library functions which are not implemented will be written in italics.

4.1.4.1 Beats and tempo maps

So far we have given note timestamps in seconds, which are related to sample counts simply by a scale factor. Musical time is often expressed in beats. The duration of a beat can vary; it is controlled by the tempo, usually given in beats per minute.

The first hurdle is that we want to talk about fractional beats, but the `time` type is an integer. So fix a convention by which time in a score is measured in *pulses*. A pulse is a certain fraction of a beat.

The tempo map is a `float vec` containing beats-per-minute values. Naturally, it should span the entire score. In this example, it is piecewise linear, generated from a list of break-points. In general, we could use any tempo specification that allows a realizable function to map beats into samples. Sampled tempo curves, for example, will work (Dannenberg 1997a).

`L.Score.pulses_to_samples` converts a score whose notes are timestamped in pulses to one whose notes are timestamped in samples. To do this, it needs the pulses-per-quarter-note value and the sampling frequency as well as the tempo map. Synthesis from the sample-based score then proceeds as before.

```
let fs = 44100.
let ppq = 960           (*  960 = 64 * 3 * 5 pulses per quarter note *)
let beats_to_pulses b = int ((float ppq) *. b)

let score_in_pulses = [|
  note1 @@ (beats_to_pulses 1.5);
  note2 @@ (beats_to_pulses 1.75);
  (* ... *)
|]

let tempo = EV.pwl_list 100. [ (* initial value 100 bpm *)
   (12, 100.);                 (* stay at 100 bpm for first 12 beats *)
   (16, 90.)                   (* ramp to 90 bpm over the next 4 beats *)
]

(* Convert score to samples.
 * It can then be synthesized from as shown earlier.
 *)
let score_in_samples =
  L.Score.pulses_to_samples tempo ppq fs score_in_pulses
```

4.1.4.2 Nested structure

The composer may prefer to program not with a flat note list, but with a hierarchically nested structure. This structure may contain motifs that are repeated with variation.

To allow arbitrary nesting, define an `'a element` type constructor. It allows two possibilities: either a single `'a`, or a vector of `'a` events. We include functions that correspond to `vmix` and `map` on these nested structures.

```
type 'a element = Note of 'a | Riff of 'a element event vec

(* float vec element -> float vec *)
let rec element_vmix elt =
  match elt with
    (Note x) -> x
  | (Riff x) -> EV.vmix (V.map (E.lift element_vmix) x)

(* ('a -> 'b) -> 'a element -> 'b element *)
```

```
let rec element_map f elt =
  match elt with
    (Note x) -> Note (f x)
  | (Riff x) -> Riff (V.map (E.lift (element_map f)) x)
```

Using this type, we can build a score that defines and reuses a motif, first unmodified and then transposed up two semitones.

```
let (motif: int element event vec)  = [|
  (Note note1) @@ (secs_to_samples 0.);
  (Note note2) @@ (secs_to_samples 0.5);
  (* ... *)
|]

let score = [|
  (Riff motif) @@ (secs_to_samples 0.);
  (Note note3) @@ (secs_to_samples 2.);
  (Riff (chromatic_transpose 2 motif)) @@ (secs_to_samples 3.);
  (* ... *)
|]
```

Then, if we define a synthesizer function, we can apply it to the score and mix down the result:

```
let synth note =
  (* ... *)

let sound = element_vmix (element_map synth score)
```

4.1.4.3 When the timestamp is not the beginning

Often, a note timestamp is referenced to the physical beginning of the sound. Then the timestamp on each note in a score is the same as the timestamp on the resulting fragment of audio. In this case, the synthesizer function does not touch the timestamp; `E.lift` means that it never even sees it.

Sometimes, however, the note timestamp is referenced to some other point in the sound. The following `synth_swell` function synthesizes a sound with a slow attack followed by a quick release, and makes the peak of this envelope be the reference point. It subtracts the attack length from the note's timestamp to get the sound's timestamp.

```
let score = [|
  (440., secs_to_samples 0.5) @@ (secs_to_samples 2.);
  (220., secs_to_samples 2.0) @@ (secs_to_samples 3.);
  (* ... *)
|]

let sinetab = LV.table_of_sine 4096
```

```
(* swell up to note time, then cut off.  *)
(* (float * time) event -> float vec event *)
let synth_swell (Event (t, (hz, attack_len))) =
  let release_len = int (0.001 *. fs)  in
  let dur = attack_len + release_len  in
  let env = LV.env_lin_ar dur (float attack_len) (float release_len)  in
  let beep = LV.osci sinetab dur (hz/.fs) 0.  in
  let physical_start = t - attack_len            (* place peak at t *)
  in (V.map2 ( *.) env beep) @@ physical_start

let sound =       (* EV.timesort corrects any out-of-order events *)
  EV.vmix (EV.timesort (V.map synth_swell score))
```

Since `synth_swell` adjusts timestamps, it may disrupt the time-ordering of the score. In this example, the first element of `V.map synth_swell score` has a timestamp corresponding to $2 - 0.5 = 1.5$ seconds, but the next timestamp is $3 - 2 = 1$ second. `EV.timesort` fixes this.

4.1.4.4 Musical notation

Types could represent more musical structure and abstraction: pitch classes, scales, key, time signature, note articulation, phrasing. Haskore (Hudak et al. 1996) shows how to do some of this in the Haskell type system, which is largely similar to O'Caml's. Chronic might also make use of control signals, in the original score or generated by a performer model, to give expression and shape gestures. Beyond the features of common music notation, however, two composers may not agree on what they want from a notation.

4.1.4.5 The standard library

These examples include a certain amount of boilerplate code that not every composer should have to think about—the use of `EV.vmix`, `V.map`, and `E.lift` to synthesize from a flat score, for instance, or the `element_vmix` and `element_map` functions for a nested score. No score format is hard-wired into Chronic, but the standard library could include functions for working with common kinds like flat and nested note-event vectors.

4.1.4.6 Parsing a score format

Writing the score as Chronic code is all very well for composition that is largely algorithmic, but composition that requires entering and editing a large number of notes is better done in a graphical editor. If the editor saves the score in a known format, a Chronic program can read it, and then go on to manipulate it algorithmically if desired.

4.1.5 How well does Chronic work here?

Leaving the score to be defined by the user works well if the language is capable of providing whatever score feature the user needs. The user can take advantage of a full programming language within the score language, giving it great power. We do not have an example of a real composition scored in Chronic, but the examples give some confidence that it could be.

4.2 FOF synthesis

Chronic's style of programming applies beautifully to granular synthesis. Take the FOF synthesis algorithm (Rodet 1984) as an example.

FOF is a technique for synthesizing a pitched signal with a spectral peak, a formant, as found in voice. It involves a stream of grains whose spacing controls fundamental frequency, whose pitch controls formant frequency, and whose envelope controls formant shape. The stream is irregularly timed, and each grain is a sequence of samples: this stream of grains is a `float vec event ivec`.

In Chronic we can work directly with this type, so implementing FOF is pleasant and easy:

```
(* FOF synthesis.
 * ffund: float ivec       fundamental frequency (normalized)
 * phase0: float           initial phase of fundamental
 * octavi: float ivec      number of octaves to crossfade downwards
 * fform: float ivec       formant frequency (normalized)
 * bw: float                       -3dB bandwidth
 * db: float ivec          attenuation
 * rise: time              grain's rise time (in samples)
 * fall: time                      fall time
 * dur: time                       duration
 * risefalltab: float vec          rise and fall shape
 *
 * The grain envelope is the product of a decaying exponential and
 * a smoothing envelope.  The rise and fall halves of risefalltab
 * shape the ends of the smoothing envelope; for example, if they
 * are linear (risefalltab is a triangle), then it is trapezoidal.
 *
 *)
let fof
    ffund phase0 octavi
    fform bw db
    rise fall dur
    risefalltab
    =
  let octmul oct i =
    let (octf, octi) = modfi oct
    in let pow2 = 1 lsl octi            (* 2^octi *)
    in if 0 < (i land (pow2 - 1)) then 0.0
       else if 0 = (i land pow2) then 1.0 else 1.0-.octf
```

```
  in let times = LIV.phasor_wrap ffund phase0  in
  let fform_at_times = (IV.peekivr 0. times fform)  in
  let octavi_at_times = (IV.peekivr 0. times octavi)  in
  let db_at_times = (IV.peekivr 0. times db)  in

  let fgrain i t fform octavi db =
    let octm = octmul octavi i  in
    let grain =
      if (octm=0.0) then V.const 0.0 0
      else let ampl = octm *. (L.db_to_amp db)  in
           let sine = LV.osc_sine dur fform (~-.(frac t) *. fform)  in
           let kenv = exp(~-.pi*.bw)  in
           let env = V.iterate (fun x -> kenv *. x) 1.0 dur  in
           let smooth_ph = EV.pwl_list 0.0
               [(rise, 0.5); (dur-1-fall, 0.5); (dur-1, 1.0)]  in
           let smooth = LV.tablei risefalltab smooth_ph  in
           V.map3 (fun x y z -> ampl *. x*.y*.z) sine env smooth
    in grain @@ (int t)

  in let unflattened = IV.map4i ([||] @@ 0) fgrain
      times fform_at_times octavi_at_times db_at_times
  in EIV.vfold (+.) 0. unflattened
(*
 * BUGS: should power be constant w/ ffund, bw?
 * should max formant level be?
 * right now peak amplitude is.
 *
 *)
```

Grain start times are generated by `LIV.phasor_wrap`, which is like an oscillator but returns not its waveform but the times at which it cycles around. For each start time we form a grain as the product of a sinusoid, an exponential envelope, and a smoothing envelope, and timestamp the grain with its start time. This stream of grains is named `unflattened`, of type `float vec event ivec`.

Finally, `fof` needs a flattened version of `unflattened`, with the overlapping grains mixed to a single audio stream. All of the *sub specie temporis* bookkeeping—the worrying about overlapping grains—that the code has evaded is reusably encapsulated in `EIV.vfold`.

Octaviation shifts pitch downward from the nominal fundamental by fading out every other grain. A value of 1.7 octaves, for example, results in grain amplitudes with a repeating pattern of $\{1, 0, 0.3, 0\}$.

4.2.1 How well does Chronic work here?

What's pleasing about this code is that it expresses our idea of the FOF algorithm. It builds grains one by one and places them in time. And notice what it does not do: it does not keep track of when *now* is and what grains are sounding and which should advance to the next stage of their envelopes. We think about the FOF algorithm *sub specie æternitatis*; finally we can write code for it that way too.

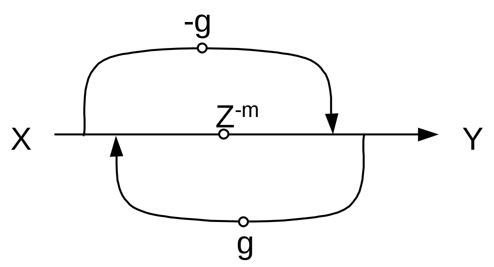
One might compare this code with the implementation of FOF written in C for Csound. In terms of lines of code, this version is one fifth the size, or one third if we include the utility functions. More importantly, its expression of the algorithm is more direct and more comprehensible.

4.3 A reverberator

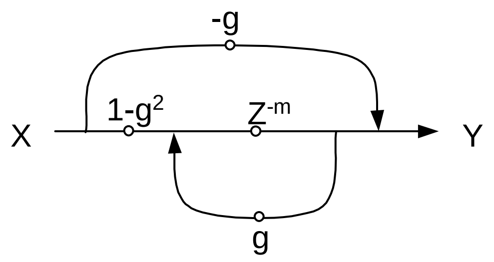
We implement a reverberator structure proposed by Moorer (Moorer 1979). It generates early echoes through a tapped delay line; we have implemented the IIR portion of the reverberator, which consists of six filtered combs in parallel, followed by an allpass filter. We follow Moorer's suggested values for these filter elements' parameters. This reverberator is not competitive in quality with current designs, but we chose to implement it because it is well-known and already specified, and illustrates typical structures found in a reverberator.

4.3.1 The allpass filter

The allpass filter is used to introduce time-domain diffusion—echoes—without steady-state spectral coloration. It has the following block diagram:



`IV.delay_rec2` places the feedback function upstream of the delay, but this diagram wants it downstream, to add in the feedforward signal after the delay. So we work with a diagram that is equivalent, but has had a slight algebraic rearrangement to disentangle the feedforward and the feedback paths. The delay-free path around the delay (with gain $-g^2$) has been incorporated up front:



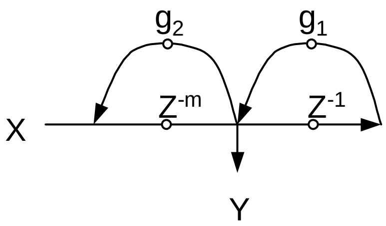
`L.IV.allpass1`, in the Chronic library, realizes this block diagram:

```
(* first-order allpass with gain _g_ and delay length _len_ *)
(* we rearrange the block diagram to fit it into delay_rec's model *)
let allpass1 len g x =
  let gg = 1. -. (g *. g)  in
  let f x y0 = IV.map2 0. (fun x y0 -> gg*.x +. g*.y0) x y0  in
  let y0 = IV.delayz_rec2 len f x  in
  IV.map2 0. (-.) y0 (IV.scale g x)
```

Here `f` is the feedback function, scaling the feedback path by g and the new input by $1 - g^2$. The feedforward path is introduced by the final `map2`.

4.3.2 The filtered comb

The filtered comb is a comb filter with a first-order IIR lowpass in its feedback path. The lowpass represents frequency-dependent attenuation of sound traveling through air; pragmatically, it serves the purpose of smearing brief clicks as they echo.



This is a natural for `delay_rec2`. The lowpass is written with `IV.filter1`, and the feedback function combines this with the comb's own feedback scaling.

```
let filtered_comb delay g1 g2 x =
  let filter = IV.filter1 (1., 0.) (-.g1)  in
  let feedback x y = IV.map2 0. (+.) x (IV.scale g2 (filter y))
  in IV.delayz_rec2 delay feedback x
```

4.3.3 The reverberator

The reverberator is six filtered combs in parallel, followed by one allpass filter. Implementation, given these elements, is straightforward:

```
let fs = 44100.

let moorer_reverb gain x =
  let delay = [| 50.;  56.;  61.;  68.;  72.;  78.  |]  in       (* msec *)
  let g1 =    [| 0.46; 0.48; 0.50; 0.52; 0.53; 0.53 |]  in
  let fcomb delay g1 = filtered_comb
      (truncate (delay *. 0.001 *. fs)) g1 (gain *. (1. -. g1)) x  in
  let (combs: float ivec vec) = V.map2 fcomb delay g1  in
  let sumcombs = V.foldl0 (IV.map2 0. (+.)) combs
  in L.IV.allpass1 (truncate (0.0061 *. fs)) 0.7 sumcombs
```

The combs' lengths (in milliseconds) and lowpass coefficients are read from tables; `fcomb` is mapped over these tables to generate a vector of filtered combs, all applied to the input `x`. This vector is mixed down with `V.foldl0` (which is like `foldl`, but uses the vector's first element in place of a base argument), and passed through an `L.IV.allpass1` whose parameters are hard-coded.

The reverberator can be tested by examining its impulse response:

```
let impulse = IV.iterate (fun _ -> 0.) 1.
let reverb = moorer_reverb 0.75 impulse
```

4.3.4 How well does Chronic work here?

The only point where we had to write Chronic code that did not represent our idea of the algorithm was in `allpass1`—a part of the Chronic standard library, not of the reverberator, but it seems worth discussing. The `delay_rec` functions can construct a particular schema of block diagrams; the standard allpass diagram does not fit into this schema. We had to rearrange it to make it fit. This is a little extra work, but more seriously, it foreshadows the problem that some diagrams are not so easily rearrangeable. This is what happens in Section 4.4.

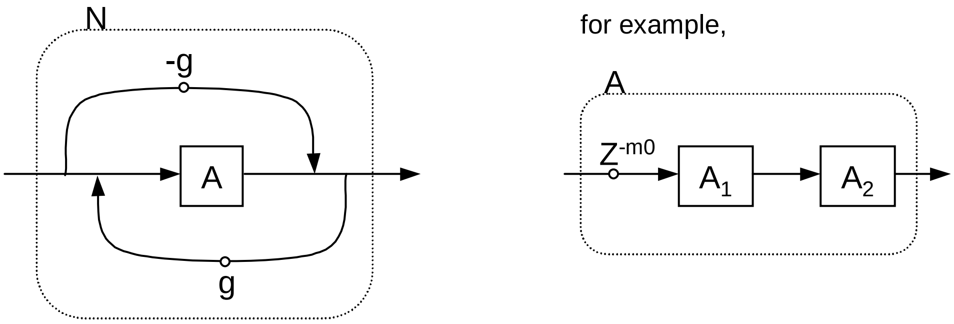
The filtered comb and the reverberator itself work well in Chronic. The reverberator's handling of the sampling rate is mildly inelegant. Its use of `V.map2` and `V.foldl0` shows how the dataflow DAG can be generated programmatically, where a language like Csound would have to enumerate the filtered combs, writing them all as in-line code.

4.4 A problematic reverberator

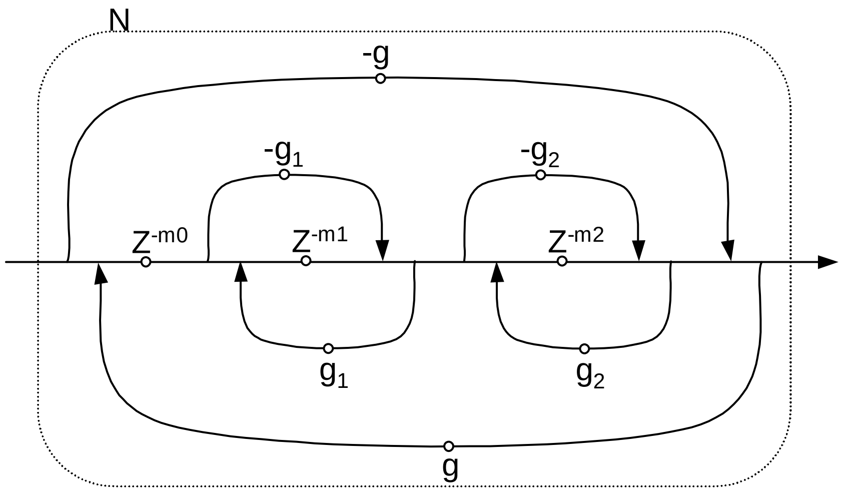
Gardner's nested-allpass reverberator structures (Gardner 1992), on the other hand, turn out to be poorly expressible in the `delay_rec` system. An implementation is possible, but only through an ungainly duplication of part of the structure.

4.4.1 Nested-allpass structures

A first-order allpass structure contains a delay element. A delay is itself an example of a system with allpass amplitude response (and linear phase); the following figure shows how another allpass system, A, can be swapped in for it while preserving the allpass nature of the entire nested-allpass structure, N. To the right is an example of what A might be: a delay, followed by first-order allpasses A_1 and A_2 in series.



Substituting these in, N in full detail is:



`L.IV.allpass1_nest` is a function to build nested allpass structures. It differs only in one line from `L.IV.allpass1`:

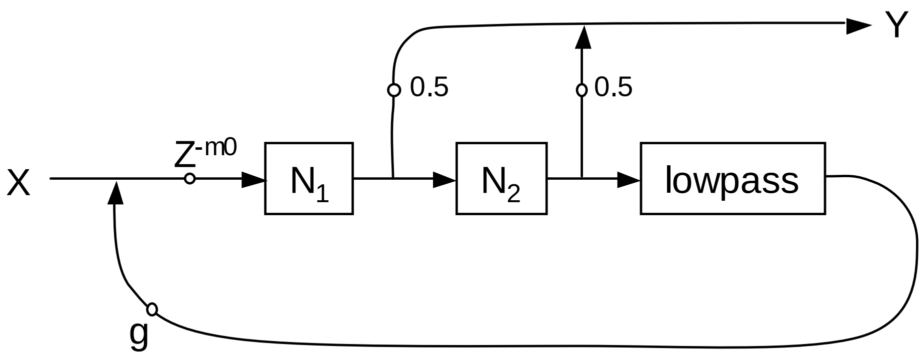
```
(* first-order allpass with gain _g_ and delay length _len_;
 * delay line is followed by (allpass) function _fnest_.
 *)
let allpass1_nest len (fnest: float ivec -> float ivec) g x =
  let gg = 1. -. (g *. g)  in
  let f x y0 = IV.map2 0. (fun x y0 -> gg*.x +. g*.y0) x y0  in
  let y0 = IV.delayz_rec2 len (fun x z -> fnest (f x z)) x  in
  IV.map2 0. (-.) y0 (IV.scale g x)
```

Now the structure N can be built by incorporating A_1 and A_2 into `fnest`:

```
let allpass_N = L.IV.allpass1_nest m0 (fun x -> allpass_A2 (allpass_A1 x))
```

4.4.2 Multiple taps

Gardner's small-room reverberator has the following structure, where N_1 and N_2 are nested-allpass elements whose details do not matter here:

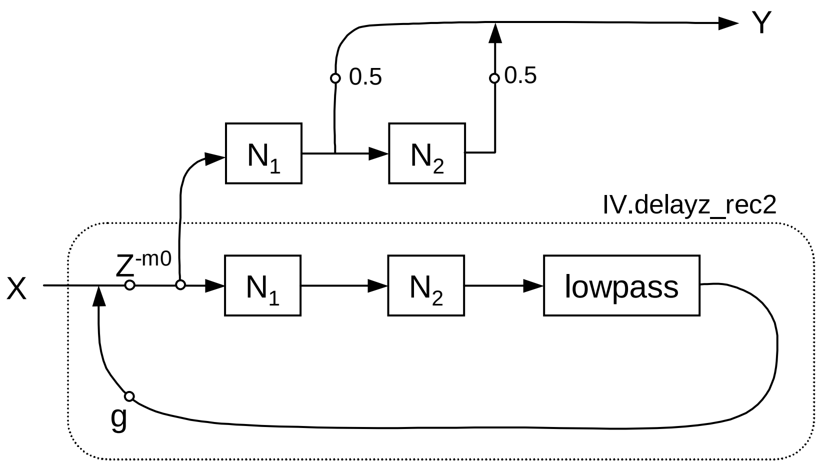


The techniques of the previous section can express N_1 and N_2, and `IV.delay_rec2` can join them into a feedback loop along with the delay, the lowpass, and the gain g. This elaborate feedback loop has the proper data flowing around it, yet there is one small but fatal problem: we need to tap that data off at two points; `delay_rec2` exposes only one (and it is neither of the two—it's right after Z^{-m_0}).

4.4.3 How well does Chronic work here?

The nested-allpass structures themselves are represented cleanly. The reverb feedback assemblage is represented too, except for the seemingly trivial task of tapping output from points inside it.

Chronic could include a feedback-multitap-delay operator that would cover this case, but even a painfully general operator would not cover every possible case. The programmer could express the entire small-room reverberator in code that manipulates delay lines just as one would in C, but this is what Chronic is supposed to avoid. A workaround is to use `delay_rec2` and accept the tap point it presents, and then replicate N_1 and N_2 to get the effect of tapping at the two desired points. The flow diagram is then like this:



But let's call a kludge a kludge.

4.5 A pitch shifter

To pitch-shift a signal is to transpose its musical pitch, as resampling it would do, but without changing its length. A harmonizer effect module might take a voice input and shift its pitch up a major third, which means scaling the frequency of each partial by 5/4 (or $2^{4/12} \approx 1.256$ for an equal-tempered third).

The simplest kind of pitch shifter that doesn't fail utterly on polyphonic signals is based on the short-time Fourier transform. It models each short-time frame as a sum of sinusoids, using phase vocoder techniques (Dolson 1987) to estimate their true frequencies, which do not necessarily fall on FFT bin centers. It then scales the frequency of each sinusoid, and from that resynthesizes the pitch-shifted output. Stephen Sprenger gives an introduction to this kind of pitch shifter (Sprenger 1999), with example source code in C, which I used as a reference.

4.5.1 The short-time Fourier transform

The short-time Fourier transform involves taking overlapping frames from the input data, multiplying each frame by a window function, and applying the FFT to each. The inverse STFT involves an IFFT on each frame, windowing again, and mixing down the overlapping frames.

4.5.1.1 Overlapping frames

Our STFT helper routines talk about overlap in terms of the size of each frame, and the distance between the start of one frame and the next—`size` and `stride`. Since `stride` is not encoded into the `float vec vec` that `takeframes` returns, it is made an argument to `mixframes`.

```
(* break a vector into overlapping frames.
 * zero-pads beginning of x so as to return (V.length x)/stride frames.
 * takeframes: int -> int -> float vec -> float vec vec *)
let takeframes size stride x =
  let x' = V.join (V.const 0. (size - stride)) x  in
  let x'len = V.length x'  in
  let frame_starts = V.int_count_by_upto 0 stride (x'len - size + 1)  in
  let headless = V.map (fun start -> V.drop start x') frame_starts
  in V.map (V.take size) headless

(* mix down a vector of frames, by breaking each frame into stride-sized
 * fragments, and picking time-aligned fragments with M.V.crossdiag.
 * mixframes: int -> float vec vec -> float vec *)
let mixframes stride x =
  let nframes = V.length x  in
  let framelen = V.length x.(0)  in
  let osamp = framelen / stride  in
```

```
  let frags = V.map (takeframes stride stride) x  in
  let ofs = V.int_count 0 (nframes + osamp - 1)  in
  let fragpiles = V.map (fun ofs -> M.V.crossdiag ofs frags) ofs   in
  let sumpiles = V.map (V.foldl0 (V.map2 (+.))) fragpiles  in
  V.ravel2 sumpiles
```

4.5.1.2 The FFT

We also need an implementation of the FFT. The following is a direct O'Caml translation, using a slightly awkward homebrew `Complex` module, of the radix-2 decimation-in-time FFT from Oppenheim and Schafer (Oppenheim and Schafer 1975), with no attempt made to re-express it in a vector-functional idiom.

```
(* radix-2 decimation in time, Oppenheim and Schafer p. 297
 *
 * complex-to-complex FFT from x into y.
 * n must be a power of 2.
 *
 * example of a stage: n = 16, log2n = 4, stage = 2
 * |\ |\/| /|  |\ |\/| /|  |\ |\/| /|  |\ |\/| /|
 * | \|/\|/ |  | \|/\|/ |  | \|/\|/ |  | \|/\|/ |
 * | /|\/|\ |  | /|\/|\ |  | /|\/|\ |  | /|\/|\ |
 * |/ |/\| \|  |/ |/\| \|  |/ |/\| \|  |/ |/\| \|
 * ngroups = 4, groupsize = 2 (butterflies/group) = wingspan (of butterfly)
 *
 *)

(* Complex operator naming conventions:
 *    +.    R + R
 *    +:    C + C     (mnemonic for "." and ":" is "one dot per component")
 *    +.:   R + C
 *    +:.   C + R
 * and "=" suffix when appropriate means "perform in place", as in C++.
 *
 *)

let cfft_core x y inverseP =
  let n = V.length x in
  let log2n = log2 n in
  let w = C.empty_vec n in              (* complex exp table *)

  (* forward or inverse transform? *)
  let sign_inverse =  if inverseP then -1. else 1. in
  let scale_inverse = if inverseP then 1./.(float n) else 1. in

  (* copy x into y in bit-reversed order *)
```

```
  for i = 0 to n-1 do
    let j = bit_reverse i (log2n - 1) in
    y.(j) <-: x.(i);
    y.(j) *:.= scale_inverse
  done;

  (* fill w *)
  let wstep = C.expi (-2. *. pi /. (float n) *. sign_inverse) in
  let wi = C.unity in
  for i = 0 to n-1 do
    w.(i) <-: wi;
    wi *:= wstep
  done;

  (* do FFT *)
  for stage = 1 to log2n do
    let wingspan = 1 lsl (stage-1) in
    let wshift = log2n - stage in

    let eta = C.z () in
    let ngroups = n lsr stage in
    for g = 0 to ngroups-1 do
      let groupi = g lsl stage in       (* beginning of group *)
      for b = 0 to wingspan-1 do        (* do group of butterflies *)
        let i1 = groupi + b in
        let i2 = i1 + wingspan in
        eta <-: y.(i2);
        eta *:= w.(b lsl wshift);
        y.(i2) <-: y.(i1);
        y.(i2) -:= eta;
        y.(i1) +:= eta
      done
    done
  done

(* complex to complex *)
let cfft_into  x y = cfft_core x y false
let cifft_into x y = cfft_core x y true

let cfft x =
  let y = C.empty_vec (V.length x)
  in (cfft_into x y;  y)

let cifft x =
  let y = C.empty_vec (V.length x)
```

```
  in (cfft_into x y;  y)

(* real to complex *)
let rcfft x =
  cfft (V.map C.from_float x)
```

It uses these simple-minded renderings of `log2` and `bit_reverse`:

```
(* base-2 log, rounded down *)
let log2 i =
  let i = ref i in
  let e = ref (-1) in
  while (!i > 0) do
    i := !i lsr 1;
    e := !e + 1
  done;
  !e

(* 0..hi bits of x, reversed *)
let bit_reverse x hi =
  let xrev = ref 0 in
  let mask = ref 1 in
  for i = 0 to hi do
    let bit = x land !mask in
    let left = hi - 2*i in
    let moved_bit = if (left >= 0)  then bit lsl left  else bit lsr (-left)
    in (xrev := !xrev lor moved_bit;
        mask := !mask lsl 1)
  done;
  !xrev
```

4.5.2 Manipulating the STFT data

The following is a direct translation of the C reference code:

```
(* based on Stephen Sprenger's smsPitchScale.cp
 * (scale: float) is the scaling factor, e.g. 0.5 for a one-octave shift down.
 * (framesize: int) is the size of each FFT frame.
 * (oversample_by: int) is how many FFT frames overlap at each sample.
 * (fs: float) is the sampling rate, used to get frequency values internally.
 * (x: float vec) is the input.
 * the output is another float vec.
 * an example call might be (pitchshift 0.5 2048 4 44100. x).
 * pitchshift : float -> int -> int -> float -> float vec -> float vec
 *)
let pitchshift scale framesize oversample_by fs x =
```

```
let xlen = V.length x  in
let stride = framesize / oversample_by  in
let framesize2 = framesize/2 + 1  in
let frames = takeframes framesize stride x  in
let window = L.V.cyclic_raised_cosine framesize in  (* SMS uses cyclic *)
(* calculate bin width, expected phase difference per stride *)
let freqperbin = fs /. (float framesize)  in
let expectdiff = twopi *. (float stride) /. (float framesize)  in

(* FFT input frames for magnitude and phase *)
let to_polar frame =
  let fft_in = (V.map2 ( *.) window) frame  in
  let fft_out = V.take framesize2 (Fft.rcfft fft_in)  in
  let (mag, ph) = V.unzip (V.map C.to_polar fft_out)
  in (V.scale 2. mag, ph)  in                (* SMS does this mul by 2 *)
let (anamag, anaphase) = V.unzip (V.map to_polar frames)  in

(* analyze phase difference to get true frequency within bin *)
let phdiff = V.map_slide2
    (V.map2 (fun x y -> y -. x)) (V.const 0. framesize2) anaphase  in
let analyze phdiff =
  let f i phd =
    let unexpected = phd -. (float i)*.expectdiff  in
    let wrapped =               (* mod 2pi into (-pi, pi] interval *)
      let raw = mod_float_nonneg unexpected twopi
      in if (raw > pi) then (raw -. twopi) else raw   in
    let freqdev = (float oversample_by) *. wrapped /. twopi
    in ((float i) +. freqdev) *. freqperbin
  in V.mapi f phdiff  in
let anafreq = V.map analyze phdiff  in

(* do the pitch shift: generate synthesis data from analysis data. *)
let scale_frame anam anaf =
  let len = V.length anam  in
  let synm = V.const 0. len  in      (* initialize scaled mag and freq *)
  let synf = V.const 0. len  in
  for i = 0 to framesize2-1 do    (* loop rendition is in fact cleaner *)
    let i' = int ((float i)/.scale)  in     (* idx to scaled frequency *)
    if (i' < framesize2) then (
      if (anam.(i') -. synm.(i) > 1.0e-5) then ( (* choose larger magn *)
        synm.(i) <- anam.(i');       (* (w/ fuzz to match SMS results) *)
        synf.(i) <- anaf.(i') *. scale
       );
      if (synf.(i) = 0. && i > 0) then (     (* try to fill empty bin *)
        synf.(i) <- synf.(i-1);
```

```
        synm.(i) <- synm.(i-1)
       )
     )
   done;
   (synm, synf)  in
 let (synmag, synfreq) = V.unzip (V.map2 scale_frame anamag anafreq)  in

 (* synthesize phase from rescaled frequencies *)
 let synthesize_phdiff sfreq =
   let f i sfreq =
     let freq_deviation = sfreq -. (float i)*.freqperbin  in
     let bin_deviation = freq_deviation /. freqperbin  in
     let with_osamp = twopi *. bin_deviation /. (float oversample_by)  in
     let with_expected = with_osamp +. (float i)*.expectdiff  in
     with_expected
   in V.mapi f sfreq  in
 let synphdiff = V.map synthesize_phdiff synfreq  in
 let accum oldphase phdiff =
   let newphase = V.map2 (+.) oldphase phdiff
   in (newphase, newphase)  in
 let synphase = V.sidescan accum (V.const 0. framesize2) synphdiff  in

 (* IFFT to synthesize output frames *)
 let ifft magn phase =
   let ab = V.map C.from_polar (V.zip magn phase) in
   let pad = C.empty_vec (framesize - framesize2) in  (*SMS zero-fills??*)
   let ifft_in = V.join ab pad  in
   let ifft_out = Fft.cifft ifft_in  in
   let re = V.map C.re ifft_out  in
   V.reverse (V.rotate_left 1 re)  in       (* diddle to match SMS ifft *)
 let iffted = V.map2 ifft synmag synphase  in

 (* window and mix output frames *)
 let windowed = V.map (V.map2 ( *.) window) iffted  in
 let scale = 2. /. ((float framesize) *. (float oversample_by))  in
 let ready_to_mix = V.map (V.scale scale) windowed  in
 let out = mixframes stride ready_to_mix  in
 out
```

The C example operates on a finite array of samples, so the Chronic version does the same—its short-time analysis data would be `complex vec vec` for the STFT frames themselves, but we immediately convert through the polar form to two `float vec vecs`, magnitude and phase. The first `vec` here is indexed by frequency, with spacing `freqperbin` Hz; the second `vec` is indexed by time, with spacing `stride` samples.

This Chronic code was intended to be an exact translation of the C original. In fact, they match up to the `mixframes` step, and then deviate. On the other hand, the C version as wrapped in a simple input/output driver routine of mine sometimes gives unreasonable output: amplitude 92179 on unity input, for example (with scale 1.0, frame size 512, and oversampling by 4, on 1024 samples of a sine at $f_s/8$). Consequently I have not tried to remove this discrepancy. It is not certain that the Chronic code is correct (even at the inherently incorrect task of pitch-shifting), but its output sounds reasonable.

4.5.3 How well does Chronic work here?

The pitch-shifting code is not dramatically clarified by being in Chronic instead of C. It is very roughly half as many lines, but then it draws on a richer library and has fewer comments. In part, it resembles the C code because it follows its data flow exactly, to make checking intermediate data easier; a freer translation might prove simpler.

Chronic's advantage over C is not necessarily how elegantly it can represent pitch shifting. Its advantage is that the pitch shifting occurs in the context of a high-level language. It can be factored into general-purpose spectral-manipulation functions, communicating in appropriate constructed types. Everything is accessible from the high level—from the compositional level. For example, the analysis functions could readily be broken out and used for a new purpose, such as generating a score where orchestral instruments track the partials of a voice sample, or the analysis/synthesis framework could be used with other spectral manipulation such as cross-synthesis, replacing the `scale_frame` function. Inadequate type systems discourage this kind of factoring into reusable subcomponents, because they makes it awkward to construct the types appearing at subcomponent interfaces, such as "stream of short-time spectra". The pitch shifter has no spectral I/O, merely audio, so it is representable, but not divisible any further.

In any case, programming an entire piece of music in C because the piece uses a pitch shifter is letting the tail wag the dog. On the other hand, MUSIC-N languages cannot themselves express pitch shifting at all. In Chronic's absence, the most plausible approach is to write the pitch shifter in C and place it as a unit generator in a MUSIC-N language. This makes it a black box, even less flexible than working directly with it in C.

Chapter 5

Implementing Chronic

Chapter 3 described the API that Chronic presents; this chapter is about the API's implementation. It presents the ideas that are key to the system, and are needed to begin reading the source code.

The first idea is that `ivecs` are handled in two phases, *building* and *computation*. The `IV` API in Chapter 3 is for the building phase; the computation phase is private. In this way, an API in a functional style is layered over an implementation in a different style, which might be painful to work with directly—for speed, our implementation makes heavy use of imperative update, but this is hidden from the API. It is important to note that computation may invoke further building, so Chronic programs are not restricted to being static dataflow graphs.

Defining `ivecs` by recursive delay—arbitrary operations in feedback paths—is an important feature of Chronic. The most direct way of handling it is disallowed by the host language. We work around this with the second idea: a delay that is higher-order, accepting a function which represents the feedback path.

5.1 Time

In Chronic, time is an integer. Time may be negative, to represent acausality. For example, a zero-phase filter could be represented as `([|-0.25; 0.5; -0.25|] @@ -1)`. Without this timestamped offset, the filter introduces a one-sample delay. The offset cancels this by making the filter acausal, meaning that its output depends on future input.

5.2 Events

An event is essentially a pair: `type 'a event = Event of time * 'a`. Operating on events themselves is trivial.

The `EV` and `EIV` modules are more complex, and use a highly imperative style. They are optimized for sparse events. For example, `EIV.pwc` (recall that it generates a piecewise-constant `ivec` from a sequence of events) is written to loop over the events; for each one, it calls `V.fill` on a segment of its output buffer. If events are dense, timestamped one sample apart, each `fill` only completes one sample of output. In that case, it would be faster to loop over the output buffer and avoid `fill`'s overhead. `EIV.vfold` (which folds overlapping timestamped vectors) makes the additional assumption that the degree of overlap is moderate—that is, not too many vectors are overlapping at once, so keeping track of them in an array is not a significant cost.

5.3 Vectors

Vectors are represented as arrays. The `V` module makes straightforward use of the O'Caml `Array` library for some of its functions; for others, it uses it with imperative update of closure data. For some, like `scanx`, it resorts to `for` loops, which unfortunately means creating an initialized array and then rewriting it. The functions whose output lengths are unpredictable—`filter` and the like—work on an overly-long buffer, and then `take` the part they need (as this proved faster than using lists). Section 7.7 explores how a different vector representation might address these and other problems.

The O'Caml compiler will not presently specialize a call such as `map (+)` by inlining the `(+)` into `map`'s inner loop, so many operations involve a function call per sample. This costs a great deal. Section 5.9 shows that a function call takes roughly ten times longer than a floating-point multiply.

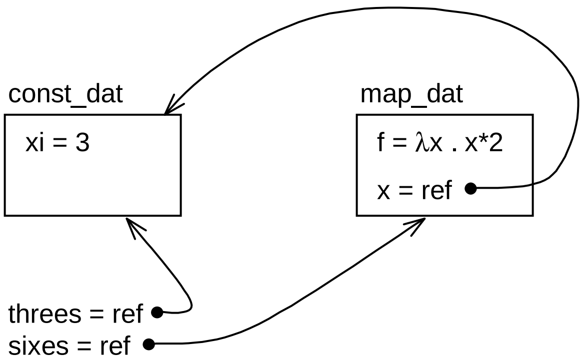
As `ivec` computation is built on vectors, it pays this price as well. Vector and `ivec` comprehensions, as proposed in Section 7.1.2, could solve this problem.

5.4 A little `ivec` example

```
  (* build a dataflow graph *)
let (threes: int ivec) = IV.const 3
let (sixes: int ivec) = IV.map (fun x -> x*2) threes
  (* ask it to compute *)
let _ = IV.print_ints 10 4 sixes
```

(Until "exemplars" are discussed in Section 5.5.3, we will leave them out of all code examples.)

`IV.const` returns an `int ivec`, which is synonymous with `int ivec_dat ref`. `IV.map` takes this `int ivec` and returns another. An `'a ivec_dat` is an O'Caml object. At this point what we have is a dataflow graph of two objects, with `!sixes` holding a reference to `!threes`.



The key methods which an `ivec_dat` presents are `get_buf: unit -> 'a array`, which exposes its output buffer, and `compute: time -> unit`, which writes output into the buffer. The `time` argument to compute is the time up to which output will be calculated.

`IV.print_ints` is called with an `upto` of 10 and a `blocklen` of 4, so it will invoke computation up to successive times of 4, 8, and 10. It calls the `compute` method on `sixes` three times, passing it these values, and prints the output on `stdout`. (This method is written `!sixes#compute`—the `!` dereferences `sixes` to get the underlying object, and the `#` accesses a method on this object `!sixes`.) Each time it is called, `!sixes#compute` calls upstream to `!threes#compute`, and maps its `fun` over the result.

This is demand-driven dataflow computation, performed in variable-length blocks, with the consumer specifying the output block length. The computation phase may involve further building: in more complex examples, pieces of the dataflow DAG will be created and destroyed as computation proceeds.

Working with individual samples would be simpler than working with blocks, but less efficient. The method call to `compute`, and the bookkeeping it does, take longer than the actual work on one sample is likely to. This overhead cost becomes tolerable when it is amortized over a block of samples. Block computation is a standard optimization dating back at least to Music V (Mathews 1969).

5.5 The computation phase

An `'a ivec` is implemented as a reference to an object. This object is an `['a] ivec_dat`, or some subclassed object that through subtype coercion can be viewed as an `['a] ivec_dat`.

5.5.1 The public interface of an `ivec_dat`

Clients acces an `ivec_dat` through an interface consisting of the following three methods:

```
get_buf: unit -> 'a array
compute: time -> unit                    (* upto: time *)
seek:    time -> unit
```

A global `maxblocklen` determines the size of the output buffer returned by `get_buf`.

The argument to `compute` is absolute time, not relative: it is the time up to which the caller wants data. It is non-strictly monotonic—that is, in successive calls, the time argument may stay the same, but may not decrease—in order to support fan-out to multiple readers. Multiple readers must all advance time in lockstep.

The `seek` method advances time without generating output, so it is permitted to step more than `maxblocklen` ahead of the current time. It may, for some `ivec_dat` subclasses, do less work than `compute`.

5.5.1.1 Legal and illegal fan-out

Fan-out is what we call the situation when one `ivec` is referenced in the definition of two or more further `ivecs`. Here, there is fan-out from `threes`:

```
let threes = IV.const 3
let sixes = IV.map (fun x -> x*2) threes
let nines = IV.map (fun x -> x*3) threes
```

Chronic makes an effort to handle fan-out without recomputing the fanned-out value for each of its readers (though in this toy case, the recomputation would be trivial). The cost of this effort is that the readers must coordinate their reading: they must all advance time by the same amount.

The familiar synchronous-dataflow regime translates to legal fan-out:

```
let threes = IV.const 3
let sixes = IV.map (fun x -> x*2) threes
let nines = IV.map (fun x -> x*3) threes
let the_answer = IV.map2 ( *) sixes nines
```

A trace of the `compute` calls might run as follows. Both `sixes` and `nines` advance `upto` by 10.

```
the_answer#compute 10
    sixes#compute 10
        threes#compute 10
    nines#compute 10
        threes#compute 10
the_answer#compute 20
    sixes#compute 20
        threes#compute 20
    nines#compute 20
        threes#compute 20
```

Fan-out is illegal when the readers advance time dissimilarly. The most blatant problem this can cause is time running backwards:

```
let threes = IV.const 3
let sixes = IV.map (fun x -> x*2) threes
let fivecount = IV.iterate (fun x -> x+5) 0
let some_threes = IV.peekiv fivecount threes
let oops = IV.map2 ( *) sixes some_threes
```

Since `fivecount` is $[0, 5, 10, \ldots]$, `some_threes` is $[\texttt{threes}_0, \texttt{threes}_5, \texttt{threes}_{10}, \ldots]$. Look at its tenth element: $\texttt{some\_threes}_9$ is equal to $\texttt{threes}_{45}$, so we compute `threes` up to (but not to) 46, to get that 45^{th} element. But this leads us to violate monotonicity. The second time `sixes` asks for more of `threes` (up to 20), `threes` has already been advanced far beyond the desired time, and it cannot go backward.

```
oops#compute 10
    sixes#compute 10
        threes#compute 10
    some_threes#compute 10
        fivecount#compute 10
        threes#seek 0                (* peekiv seeks to first offset, *)
        threes#compute 1             (* then computes one further. *)
        threes#seek 5                (* peekiv doing second offset. *)
        threes#compute 6
        ...
        threes#seek 45               (* peekiv doing tenth offset. *)
        threes#compute 46            (* fivecount.(9) = 45 *)
oops#compute 20
    sixes#compute 20
        threes#compute 20            (* fails: 20 < 46     *)
```

But divergent readers are never legal, even when they preserve monotonicity:

```
let threes = IV.const 3
let sixes = IV.map (fun x -> x*2) threes
let offset = IV.iterate succ 5           (* 5, 6, 7, ... *)
let some_threes = IV.peekiv offset threes
let oops = IV.map2 ( *) sixes some_threes
```

Now `some_threes` is equal to $[\texttt{threes}_5, \texttt{threes}_6, \texttt{threes}_7, \ldots]$. Its access pattern to `threes` is offset from that of `sixes`, which violates certain assumptions.

```
oops#compute 10
    sixes#compute 10
        threes#compute 10
    some_threes#compute 10
        offset#compute 10
        threes#seek 5                 (* offset.(0) = 5     *)
        threes#compute 6
        ...
        threes#seek 14                (* offset.(9) = 14    *)
        threes#compute 15
the_answer#compute 20
    sixes#compute 20
        threes#compute 20             (* 20 > 15, but...    *)
```

When `sixes#compute 20` calls `threes#compute 20`, it expects to get ten samples of output, since *its* previous call was to `threes#compute 10`. It gets only five, because `threes#compute 15` was called behind its back. (Worse, it mistakes these samples 15–19 for 10–14, and then continues along the array to find old 5–9 data which it misuses as 15–19.)

5.5.1.2 Techniques to support divergent readers

Occasionally one may require the effect of fan-out with divergent readers. The general workaround is to eliminate the fan-out by giving each reader a private replica of the fan-out source. One may simply duplicate the expression for the source:

```
let threes1 = IV.const 3
let threes2 = IV.const 3
let sixes = IV.map (fun x -> x*2) threes1
let fivecount = IV.iterate (fun x -> x+5) 0
let some_threes = IV.peekiv fivecount threes2
let ok_now = IV.map2 ( *) sixes some_threes
```

But it may be cleaner, especially if `threes` is defined by a complicated expression, to wrap it in a function and instantiate it twice:

```
let threes () = IV.map2 (+) (IV.const 1) (IV.const 2)
let sixes = IV.map (fun x -> x*2) (threes ())
let fivecount = IV.iterate (fun x -> x+5) 0
let some_threes = IV.peekiv fivecount (threes ())
let ok_now = IV.map2 ( *) sixes some_threes
```

It might sometimes be convenient to clone the source and all of its upstream dependencies after their instantiation. `IV.deepcopy` is intended to do this. Unfortunately, it is implemented in terms of the O'Caml standard library's `Marshal` module, which is missing support for objects. Since an `ivec` is an object reference, it can't be deep-copied until O'Caml plugs this gap. Using `IV.deepcopy` would look like this:

```
let threes1 = IV.map2 (+) (IV.const 1) (IV.const 2)
let threes2 = IV.deepcopy threes1
let sixes = IV.map (fun x -> x*2) threes1
let fivecount = IV.iterate (fun x -> x+5) 0
let some_threes = IV.peekiv fivecount threes2
let ok_now = IV.map2 ( *) sixes some_threes
```

Fortunately, the previous solution, of using a function wrapper to delay and duplicate instantiation, works rather well.

5.5.2 The internals of an `ivec_dat`

The definition of `ivec_dat`, leaving out debug hooks, is as follows:

```
class ['a] ivec_dat = object (self)
  val mutable buf = ([||] : 'a array)         (* derived class must init *)
  val mutable buf_start = 0
  val mutable buf_upto = 0
  method get_buf = buf

  method compute_hook new_upto len = ()       (* abstract *)

  method compute new_upto =
    assert (new_upto - buf_upto <= maxblocklen);
    if (new_upto = buf_upto) then ()
    else if (new_upto < buf_upto) then err_rewind "ivec_dat#compute"
    else (buf_start <- buf_upto;
          buf_upto <- new_upto;
          self#compute_hook new_upto (buf_upto - buf_start)
         )

  (* performs intermediate calculation -- override if you don't need to *)
  method seek new_upto =
    while (buf_upto < new_upto) do
        let upto = min (buf_upto + maxblocklen) new_upto
        in self#compute upto
    done
end

type 'a ivec = ('a ivec_dat) ref
```

The `ivec_dat#compute` method performs the monotonicity checking, factoring it out from the `compute_hook` that subclasses are expected to implement.

This default `seek` is implemented in terms of `compute`, in case `compute` has side effects. Subclasses where this can be ruled out will override `seek`.

5.5.3 Subclassing `ivec_dat`

Primitives in the `IV` library are created by subclassing the `ivec_dat` class to provide a `compute_hook` method, and optionally to override `seek`. The following code implements `map_dat` for `IV.map` (which applies a scalar-to-scalar function to each element of an `ivec`).

```
  class ['a, 'b] map_dat (exemplar: 'b) (f: 'a -> 'b) (xin: 'a ivec) =
  object inherit ['b] ivec_dat
    val f = f
    val x = xin
    val xbuf = !xin#get_buf
    initializer
      buf <- V.const exemplar maxblocklen
```

```
    method compute_hook new_upto len =
      !x#compute new_upto;
      VL.map buf len f xbuf

    method seek new_upto =
      !x#seek new_upto;
      buf_upto <- new_upto
  end
```

IV.map has an 'a ivec as input and a 'b ivec as output, so map_dat is parameterized over both of these type variables, and inherits from 'b ivec_dat.

The compute_hook requests data from x, and calls VL.map: map f over len of xbuf, into buf.

IV.map assumes that f is free of side effects, so it provides a seek that will not request unnecessary elements of x, nor apply f to them.

The exemplar argument is a workaround for a technical problem. We need to create an array, buf. O'Caml requires that arrays be initialized during creation. All we know is that its type is 'b vec; we don't know what type 'b is, so we have no way to come up with any particular value of that type to use as the initializer value. We have to ask the programmer for one. 7.5 talks about how we might try to avoid this.

5.6 The building phase

Associated with map_dat is a builder function, map:

```
  let map (exemplar: 'b) f x =
    ref ((new map_dat exemplar f x) :> ('b ivec_dat))
```

It creates a new map_dat with the given arguments, coerces it to the ivec_dat of which it is a subtype, and returns a reference. Through this subtype coercion, all builder functions can return an 'a ivec.

In some cases the builder is more than just a wrapper:

```
  let join (prefix: 'a vec) x =
    if (V.length prefix) = 0 then x
    else ref ((new join_dat prefix x) :> ('a ivec_dat))
```

If the join is a no-op, there is no need to create a join_dat.

5.7 Recursive delay

We would like to allow cycles in the dataflow graph, as long as each cycle contains a delay. (Delay-free cycles are unrealizable, short of solving arbitrary fixed-point problems.) A cycle corresponds to a group of ivecs whose definitions are mutually recursive. Recall that delay has the signature (length: time) (basis: 'a vec) (x: 'a ivec) -> (y: 'a ivec), where length is in samples, and basis fills in for y before time 0. We would like to use delay to define, for example, this 1-cycle:

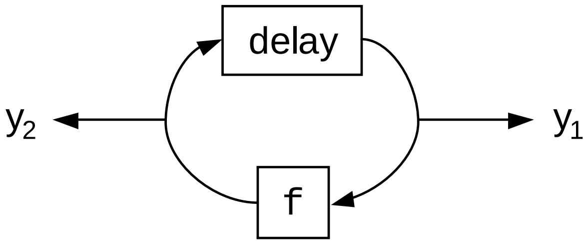
```
let rec nats = IV.map succ (IV.delay 1 [|0|] nats)
(* [|0|] is the basis: effectively, y.(-1) = 0 *)
```

This is not legal in O'Caml (nor in SML). Only certain types can be defined recursively: function types, chiefly, plus implementation-dependent extensions that don't solve our problem. Informally, the restriction is because the language can't rule out a definition like `let rec x = f x` that is well-typed but non-terminating, so the type has to have an element representing non-termination. Section 7.4 explores ways to make `ivec` such a type, but they do not ultimately prove attractive.

In lieu of this, Chronic fuses the feedback cycle into a single dataflow node, a higher-order node which takes the feedback path as an argument of type `'a ivec -> 'a ivec`. The function for recursive constant-length delay, `delay_rec`, has the signature `(length: time) (basis: 'a vec) (f: 'a ivec -> 'a ivec) -> (y: 'a ivec)`. It satisfies $y = \mathsf{delay}\ \mathit{length}\ \mathit{basis}\ (f\ y)$. Using `delay_rec`, we can write

```
let nats = IV.delay_rec 1 [|1|] (IV.map succ)
(* nats = [1, 2, 3, ...] *)
```

Note the distinction between $y_1 = \mathsf{delay}\ \mathit{length}\ \mathit{basis}\ (f\ y_1)$ and $y_2 = f(\mathsf{delay}\ \mathit{length}\ \mathit{basis}\ y_2)$, as diagrammed here:



On one hand, `delay_rec` may seem over-complicated in taking an `ivec` feedback function. Why not just a scalar feedback function `f: 'a -> 'a`? In this example, that would be sufficient; the scalar `f` would be `succ`. Even in cases where the scalar function has to maintain internal state, it can do so. But often we would like to use existing `ivec` operators in a feedback loop—an `IV.filter1` in the feedback of a comb filter, for example, is an element used in the reverberator of Section 4.3. A scalar feedback function is undesirable then for two reasons. First, it effectively forces block length to one, when the delay length should be the only constraint. Second, it would require us to have an `unmap` function of type `('a ivec -> 'b ivec) -> ('a -> 'b)`, whereas an `ivec` feedback function only requires the `IV.map` that we already have.

The feedback function for `delay_rec` has one `ivec` input, namely the feedback itself. The `ivec` it generates is purely recursive, without outside `ivec` input. `IV.delay_rec2` does feedback delay with one outside input (so its feedback function has two inputs). This means it can, for example, apply repeating echo to an `ivec` input.

The `_rec2` functions are implemented in terms of the `_rec` ones. However, it is not as simple as partially applying the feedback function. The following code is incorrect:

```
let delay_rec2 d basis (f: 'b ivec -> 'a ivec -> 'a ivec) (x: 'b ivec) =
  delay_rec d basis (f x)
```

Here, the expression `(f x): 'a ivec -> 'a ivec` creates a feedback function that refers to `x`. In fact, it is an error for a feedback function to refer to an `ivec` that is not one of its arguments. This situation is an error because of problems with differing-length blocks, as discussed in Section 5.8. That section also describes the reblocking primitives that are in fact used to build a correct `delay_rec2` in terms of `delay_rec`.

This constraint on feedback functions means that a `delay_rec3` might very well be needed too, and so on.

5.8 The block-subblock problem

Chronic's dataflow protocol breaks down when an `ivec` has consumers who ask for blocks with the same end time, but different starting times. For example, if `consumer1` is making length-5 requests and `consumer2` is making length-10 ones:

```
consumer1: !source#compute 5
consumer1: !source#compute 10
consumer2: !source#compute 10          (* uh oh *)
```

Although `consumer2` expects to see ten samples, namely 0–9, what it gets is only five, 5–9. Consequently it misattributes as 0–4 the samples that it does get, and then goes on to read from a region of `source`'s output buffer that was never written to. We call this the block-subblock problem.

The `compute` calls include no indication of the desired length, so `source` implicitly assumes that the length is the difference between this `upto` and the last one—the last one received from any consumer, not necessarily the consumer at hand. When `source` receives the request from `consumer2`, it takes it as a no-op.

This failure is not detected, and causes incorrect output. If the request included a length, it could at least trigger a fatal error. As matters stand, though, the only solution is to use Chronic according to a discipline that makes the failure impossible.

The simplest discipline is to have the ultimate consumer of the `ivec` DAG determine the block length used throughout. (This does rather vitiate the choice of consumer-controlled over producer-controlled block length—see Section 6.1.3.) This global property is achieved by composition of a local property: that an `ivec` operator propagates its consumer's choice of block length upstream unchanged.

Most of the primitives in `IV` satisfy this local property. `IV.delay_rec` and its relatives, however, can violate it. The following code fragment causes `consumer1` and `consumer2` to make exactly the `!source#compute` calls we saw above.

```
let source = IV.iterate succ 0
let consumer1 =
  let f y = IV.map2 (+) y source     (* source is a free variable *)
  in IV.delayz_rec 5 f
let consumer2 = IV.map (~-) source
let sink = IV.map2 (+) consumer1 consumer2
let _ = IV.print_ints 10 10 sink
```

`IV.delayz_rec` does not take any `ivec` arguments, so we might not expect it to be capable of violating the local property. But in this case, it has an implicit dependency on `source` smuggled in through the feedback function `f`. As `delayz_rec` operates here in five-sample blocks, it calls `f` on five-sample blocks, and `f` in turn pulls five-sample blocks from `source`. Then, of course, `consumer2` asks for a ten-sample block.

The workaround is to avoid creating these implicit dependencies. A `delay_rec` feedback function must not refer to `ivec`s aside from its own arguments; those arguments can be reblocked so as to be safe. So the correct way to write the code fragment is with `delayz_rec2`:

```
let source = IV.iterate succ 0
let consumer1 =
  let f source y = IV.map2 (+) y source     (* source is bound *)
  in IV.delayz_rec2 5 f source
let consumer2 = IV.map (~-) source
let sink = IV.map2 (+) consumer1 consumer2
let _ = IV.print_ints 10 10 sink
```

Now `delayz_rec2` can catch the length-5 requests and batch them together into a length-10, taking its cue from the length-10 request made of it. What `source` sees is all length-10:

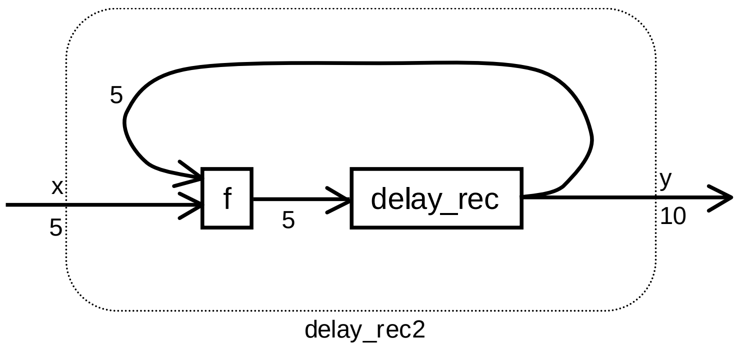
```
consumer1: !source#compute 10
consumer2: !source#compute 10
```

5.8.1 The reblocking primitives

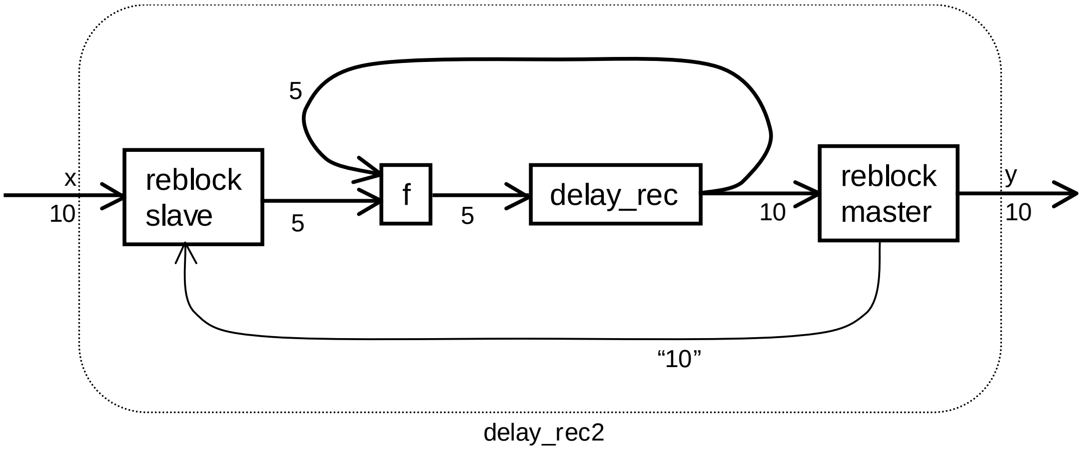
Ignoring the block-subblock problem, we could write `delay_rec2` in terms of `delay_rec` very simply:

```
let delay_rec2 d basis (f: 'b ivec -> 'a ivec -> 'a ivec) (x: 'b ivec) =
  delay_rec d basis (f x)
```

This definition leads to the following flow of `ivec` data. Each arc is labeled with the block length of the `ivec` that flows along it if we plug this definition into our two-consumer example.



A corrected version can be diagrammed like this:



Here the previous structure is bracketed by a pair of reblocking primitives. On its input side, the `reblock_slave` passes an `ivec` through unchanged, but reblocks it; in this case it batches length-5 requests into length-10 requests. The number "10" comes to it via a side channel from the `reblock_master`, which passes an `ivec` through without touching it at all, only noting and communicating its block length.

The actual implementation of `delay_rec2` in terms of `delay_rec`, which realizes this structure, can be seen in the `IV` source. It makes use of these two reblocking primitives, which are internal to `IV`:

```
(slave: 'a reblock_slave_dat ref, x': 'a ivec) =
  reblock_slave (x: 'a ivec)

(x': 'a ivec) =
  reblock_master (x: 'a ivec) (slave: 'a reblock_slave_dat ref)
```

They can be applied, for example, to wrap an `ivec`-to-`ivec` function so that it will preserve block length:

```
(* f may not preserve block length, but f' does. *)
let reblock (exemplar_a: 'a) (exemplar_b: 'b) (f: 'a ivec -> 'b ivec) =
  let f' (x: 'a ivec) =
    let (slave, slave_as_ivec) = reblock_slave exemplar_a x
    in reblock_master exemplar_b (f slave_as_ivec) slave
  in f'
```

5.8.2 Doing away with the block-subblock problem

To make the block-subblock situation no longer an error, each producer would have to be aware of all of its consumers, and each `compute` call would have to include a `from` time, an `upto`, and the identity of the consumer making the request. Furthermore, `compute` would return an offset into the output buffer. These additional features would allow the producer to keep track of where each consumer has read to, to expose to each one the correct range of samples in its output buffer, and to overwrite the buffer only when all consumers have reached a point of alignment. (They would be required to reach alignment at least once every `maxblocklen` samples.)

5.9 Speed for static dataflow

Measuring Chronic's performance is of limited use, because the style of programming illustrated by Chronic is of more interest than Chronic itself. Still, some mini-benchmarks can confirm and quantify our ideas of where the time goes in a Chronic execution.

The `const` benchmark does no per-sample work at all (`IV.const`'s `compute` function returns immediately), so it measures pure per-block overhead:

```
let x = IV.const 2.
in IV.ignore n blocklen x  in
```

The `map2` and `add2` benchmarks each include two `const`s. Beyond that, the difference between them is whether the per-sample arithmetic is inlined or is through a function call. `map2`:

```
let x = IV.const 2.  in
let y = IV.const 3.  in
let z = IV.map2 0. (+.) x y
in IV.ignore n blocklen z  in
```

Versus `add2`:

```
let x = IV.const 2.  in
let y = IV.const 3.  in
let z = IV.add2_float x y
in IV.ignore n blocklen z  in
```

The following C++ code corresponds to `add2` in inlining the add operation and in doing no per-sample work (no work at all, in fact) for the constant signals:

```
int t = 0;
while (t < n)
   int len = n - t < blocklen ? n - t : blocklen;
   for (i = 0; i < len; ++i)
      mix[i] = const2[i] + const3[i];

   t += len;
```

The O'Caml code was compiled with `ocamlopt -unsafe -noassert -inline 1` using version 3.01; the C++ was compiled with `g++ -O -funroll-loops` using version 2.95.2. Times are normalized relative to `add2` at block length 256. All tests were run on a '333-MHz Pentium II machine.

Normalized CPU consumption

blocklen	`const`	`map2`	`add2`	C++
1	12.0	42	31	1.75
2	6.0	26	15.4	1.05
4	3.0	17.2	10.8	0.63
8	1.5	13.4	6.2	0.55
16	0.76	10.6	2.7	0.45
32	0.36	9.5	1.69	0.46
64	0.173	9.1	1.40	0.43
128	0.101	8.8	1.22	0.38
256	*0.050*	*9.3*	*1*	*0.36*
512	0.024	10.2	1.10	0.38
1024	0.012	10.2	1.92	1.62

These numbers quantify two things we probably expected: first, function calls are costly. Between `map2` and `add2`, we can see that an O'Caml function call is about an order of magnitude slower than an arithmetic operation. To be efficient in O'Caml, the per-sample work must never include function calls. Second, Chronic's per-block overhead, which involves O'Caml object method calls, is high. The time consumed in `add2` at block length 1 is 30:1 overhead to computation, whereas in the C++ benchmark (which admittedly has no function calls and is realistic only as a model of generated code) it is approximately 4:1 (computed as $(1.75 - 0.36)/0.36$). Forcing Chronic to compute with a short block length, as feedback delay can do, is deadly.

The `add2` mini-benchmark, static dataflow computation with inlined arithmetic, is about three times slower than comparable C++ code. This is talking largely about the execution speeds of O'Caml versus C++, which is ultimately not a subject of interest here.

Chapter 6

Decisions behind Chronic

Chapter 5 described how Chronic is implemented. This chapter explains how that implementation was arrived at.

6.1 The structure of an `ivec`

One's first thought is that representing `ivec`s is trivial: an `ivec` is a lazy list. This has the correct semantics, but its asymptotic space use can be excessive.

```
let (x: float ivec) = IV.const 0.  in
let n = 1000  in
let blocklen = 10
in IV.print_float n blocklen x
```

A program like this ought to run indefinitely in $O(1)$ space, but with the lazy-list representation it takes $O(n)$ space (and eventually runs out of memory) because `x` is a reference to the entire `ivec` and never goes out of scope. An `ivec` ought to discard past data that will not be needed again.

We can solve this problem, but the resulting `ivec` representation will not be perfect either. This section will explore a number of representations, traversing the decision tree in Figure 1.

The tree is unbalanced because this chapter is not an exhaustive review, but an explanation of how Chronic got to be the way it is. We will go a bit down byways, but the bulk of this section is successive refinements to the `ivec` representation, each of which meets a new goal as well as the previous ones. In the end we arrive at the solution, using objects, given in the previous chapter. Some of the refinements are optimizations that we perform because our compiler cannot, so the intermediate solutions may be of interest when speed is less important, or when compilers improve.

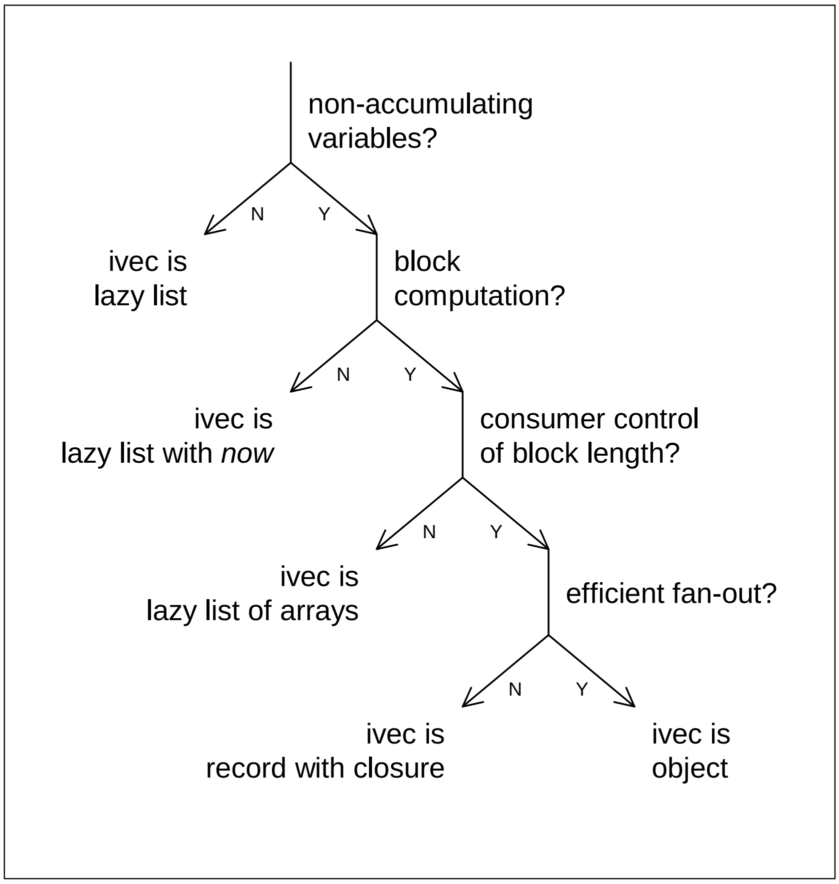


Figure 1: Decision tree of `ivec` representations.

6.1.1 Should variables be accumulating?

The term "lazy list" may be ambiguous. We use it to refer to a list whose constructor uses lazy evaluation (evaluates each closure no more than once). A list whose constructor is non-strict but which does not cache evaluations (normal-order evaluation without updating) is referred to instead as a "non-strict list". The list type found in a non-strict language will be a lazy list; the simplest non-strict list built in O'Caml will not be lazy:

```
type 'a nslist = Nslist of (unit -> 'a * 'a nslist)
```

Lazy lists result in *accumulating* variables, which accumulate past samples for their entire lifespan. This is not necessarily fatal. Working around it is a matter of programming discipline: never create a variable whose lifespan is indefinite. Nyquist programs (Dannenberg 1997c) operate under this discipline; it proves possible to work under, though it can surprise the unindoctrinated programmer. Nyquist encourages a style where signals are handled through functions rather than being evaluated to variables, and output is performed with special macros that avoid retaining references to data after it has been written.

For example, the code above can be fixed by substituting in for `x`: `IV.print_float n blocklen (IV.const 0.)`. This transformation eliminates the toplevel reference to the head of the zero `ivec` that was keeping it non-garbage.

6.1.1.1 Non-accumulating variables

If we prefer to have variables that are not accumulating, we can build them on top of either lazy lists or non-strict lists.

From lazy lists To build non-accumulating `ivec`s on top of lazy lists, give an `ivec` two parts: an infinite list representing the future, and a `now` value telling when the future starts from.

```
type 'a ivec = Iv of time * 'a lazylist
```

Certain functions, *accessor* functions, advance the `now` and discard a segment of the future. The functions which perform output to a file or to a DAC are accessors, as is `IV.take`.

Access must be strictly monotonic. The `Nonmonotonic` exception is raised when an access is attempted to an element before `now`:

```
IV.take: time -> 'a ivec -> 'a vec
(* squares = (0, [0, 1, 4, 9, ...]) *)
let first3 = IV.take 3 squares
(* first3 = [0, 1, 4]
 * squares = (3, [9, 16, 25, ...] *)
let _ = IV.print_ints 5 squares
(* raises Nonmonotonic exception. *)
```

An analogous fix to non-strict lists Even a direct use of non-strict lists as `ivec`s does not lead to accumulating variables. Instead of spending space to hold the result of forcing a suspension, they spend time to recompute it. These `ivec`s, instead of being asymptotically too large, are asymptotically too slow.

For example, this implementation of `IV.peekiv` re-traverses `x`; for an arithmetically-spaced `i` input, it takes quadratic time instead of linear:

```
IV.peek: time -> 'a ivec -> 'a
let peekiv i x = IV.map (fun i -> IV.peek i x) i
```

This time problem can be fixed just as lazy lists' space problem was, by adding a `now`.

6.1.1.2 Accumulating variables

If we were to go down the path of accumulating variables, we would want features described in the sections below, which lazy lists don't provide. Nyquist's implementation(Dannenberg 1997b) of accumulating variables shows how to provide block computation and efficient fan-out.

6.1.2 Block computation

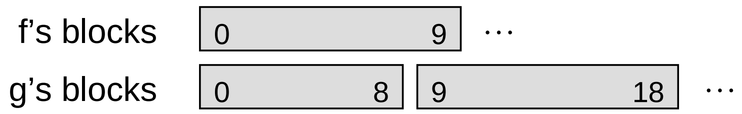
Working with individual samples is inefficient; the overhead only becomes tolerable when it is amortized over a block of samples. (This is true even for lower-level languages such as C, though the inefficiency may be more or less extreme.) For block computation, we define an `ivec` as a non-strict list of arrays, plus a `now`:

```
type 'a ivec = Iv of time * 'a array nslist
```

Although Chronic chooses not to use this definition, it should be possible to make it work. We will briefly describe how this might be done.

6.1.2.1 Block alignment

This definition means that the producer of the `ivec`, not the consumer, controls the block lengths. The consumer cannot afford to be inflexible about what block lengths it intends to generate, as this can cause unbounded accumulation of processing delay. For example, `let y = f (g x)`, and say that `f` intends to generate a first block of length 10, from times 0 through 9. But `g` may make its first block length 9, causing `f`'s first block to depend on `g`'s second, which may stretch a further 10 samples. This dependency on the future translates in a real-time context into processing delay. We set the maximum block length (here to 10) intending to bound the delay, but the worst-case delay turns out to be almost the maximum block length times the number of nodes in the processing network.



A change in the consumer's policy fixes this problem: let the consumer place block boundaries wherever the producer did. But what about a consumer, such as `IV.map2`, that has multiple `ivec` inputs? Its producers' block boundaries may not align.

It can deal with this by placing its output block boundaries wherever *any* of its inputs have one, or where *all* have one—the OR and the AND approaches. The former means that `map2` has to maintain internal state about partially-used blocks, which complicates its logic a good deal. The latter may mean that it can never place a block boundary at all.

But we can try to salvage the AND approach by placing constraints on `map2`'s inputs. Namely, every `ivec` must have block boundaries at all times that are multiples of some constant $T_{\text{forcebreak}}$. Chronic's wide palette of operations which shift or alter the timebase of an `ivec` makes this constraint less natural than it might otherwise be; these operations need to do the kind of partial-block bookkeeping mentioned for the OR approach.

6.1.2.2 Making recursive delay work

The `delay_rec` function is unable to accept arbitrarily long blocks from its feedback loop `f`—blocks cannot be longer than the delay length. If we give `delay_rec` no direct control over block lengths, it has to take a roundabout approach. It can generate length-1 blocks itself. Its output is what `f` operates on, so `f` will operate on these length-1 blocks, and generate length-1 blocks of its own, which `delay_rec` can accept.

Letting these length-1 blocks propagate down through the graph would defeat the purpose of using blocks at all. Instead, reblock the output into larger blocks.

```
(* generate x' with the same value as x, but with blocks of length _len_ *)
reblock: int -> 'a ivec -> 'a ivec
```

Length-1 blocks are guaranteed to work, but may be shorter than necessary. For greater efficiency, `delay_rec` can use blocks whose length is equal to the delay length; `vdelay_rec` can use the logic found in Chronic's `IV.vdelay_rec` to examine the time-varying delay length and use the longest possible block.

We have been assuming that `f` preserves block lengths. This is not necessarily true. If `f`'s environment contains an `ivec input` that it draws upon, `input` will probably be generating long blocks, and if we are using the AND approach from the previous section, these long blocks will force `f`'s output blocks to be long as well. To avoid this, `f` should be written in terms of `IV.reblock 1 input`.

A wrapper function with the same interface as `delay_rec` can add the output reblocking automatically. The input reblocking is not so transparent to the user.

6.1.3 Consumer control of block length

Giving the consumer of an `ivec` control of block length avoids the complications encountered in the previous section. An `ivec` has a closure which the consumer calls, specifying the block length.

```
  type 'a ivec = { now: time;
                   compute: ('a vec -> time -> 'a ivec) }
                   (* compute: out upto -> future *)
```

The `upto` argument to `compute` is the time up to which samples should be computed. That is, it is absolute time, not relative (so the block length is specified indirectly, as the difference between successive `upto`s). Using relative time—the length of the vector to be computed—would lead to erroneous results in the case of fan-out, when an `ivec` is referred to more than once, as in this example:

```
let c =
  let a = some expression  in
  let b1 = IV.map f1 a  in
  let b2 = IV.map f2 a  in
  IV.map2 g b1 b2

let out = V.const 0 10
let c' = c.compute out 10
```

Attempting to use relative time causes the following sequence of calls:

```
c.compute 10            (*  c will compute samples 0...9 *)
    b1.compute 10       (* b1 will compute samples 0...9 *)
        a.compute 10    (*  a will compute samples 0...9 *)
        compute using a[0...9]
    b2.compute 10       (* b2 will compute samples 0...9 *)
        a.compute 10    (*  a has done 0...9, now does 10...19 *)
        compute using a[10...19]
    compute using correct b1 and erroneous b2    (* wrong! *)
```

But absolute time fixes the problem:

```
c.compute 10            (*  c will compute samples 0...9 *)
    b1.compute 10       (* b1 will compute samples 0...9 *)
        a.compute 10    (*  a will compute samples 0...9 *)
        compute using a[0...9]
    b2.compute 10       (* b2 will compute samples 0...9 *)
        a.compute 10    (*  a will recompute samples 0...9 *)
        compute using a[0...9]
    compute using correct b1 and b2              (* right. *)
```

6.1.3.1 Limitations of consumer control

Consumer control of block length does not necessarily give complete freedom in block lengths. Chronic's `ivec` definition has consumer control, but Section 5.8 described how in practice it requires that the same block length be used throughout the dataflow graph, except in carefully constructed subgraphs wrapped in reblocking primitives. It seems that a similar reblocking construction should work for producer-controlled block length. Therefore it is not clear that Chronic's level of consumer control is preferable to producer control.

A more elaborate consumer-control protocol, on the other hand, might allow more freedom of block lengths. Section 5.8.2 sketches one possibility.

6.1.4 Efficient fan-out

Where two or more consumers read from a single producer, we would like the producer not to recompute for each one. Consider the flow graph from the previous section:

```
let c =
  let a = some expression  in
  let b1 = IV.map f1 a  in
  let b2 = IV.map f2 a  in      (* does a reuse its work for b2? *)
  IV.map2 g b1 b2
```

We would like the computation to run not as it did in that section, but rather as follows:

```
c.compute 10            (*  c will compute samples 0...9 *)
    b1.compute 10       (* b1 will compute samples 0...9 *)
        a.compute 10    (*  a will compute samples 0...9 *)
```

```
        compute using a[0...9]
    b2.compute 10        (* b2 will compute samples 0...9 *)
        a.compute 10     (*  a has done 0...9, reuses them *)
        compute using a[0...9]
    compute using b1 and b2
```

This has several implications. First, it becomes better for each `ivec` to own its output buffer, rather than writing into one provided by the `compute` caller (and having to cache the output for recopying).

Second, `now` becomes insufficient information. It says `a` has computed up to time 10, but it does not say where the sample for time 9 occurs in the output buffer. It is renamed `buf_upto`, and augmented with `buf_start`, the time corresponding to the buffer's first element: in the example at the "reuse" point, `a.buf_upto` is 10, and the first element of the buffer is the sample for time 0, the value of `a.buf_start`.

Third, the reuse is a side effect that `b1`'s call to `a.compute` has on `b2`'s call to it. To achieve this, `a` updates `buf_upto` in place when `b1` calls `a.compute`, and `b1` and `b2` hold references to a common `a` so the update is seen. Likewise, `a` updates in place the `compute` closure itself, because if it continued to return a future `ivec`, it would have to be careful to construct just one shared future `ivec` and give each fan-out consumer a reference to it.

We can imagine incorporating all of these features into an `ivec` definition like this:

```
type 'a ivec = { buf: 'a vec;
                 mutable buf_start: time;
                 mutable buf_upto: time;
                 mutable compute: (time -> unit) }
                 (* compute takes new buf_upto *)
```

Note that O'Caml automatically handles the record type by reference; also that `buf` does not need to be mutable, only its elements. The `compute` closure must contain a self-reference through which it updates the mutable values.

What Chronic actually does is to say that at this point we've rolled our own object, and we might as well just use O'Caml's object features. Then inheritance allows `ivec` to be factored into common and operation-specific parts, as described in Sections 5.5.2 and 5.5.3. Subtype coercion allows the factoring to be hidden. Method calls are slower than ordinary function calls, but for the purpose of development, eliminating duplication of common code is sufficient benefit to outweigh the cost of method calls.

We have now arrived at the `ivec` structure currently implemented in Chronic.

6.2 Why no IV.peek?

Chronic does not have any means for accessing a single sample from an `ivec`—there is no `IV.peek: time -> 'a ivec -> 'a`. Why not? Well, how would this function be used? Most likely, inside an `IV.map` or other function that would apply it infinitely often to a single `ivec`. This case is better handled by `IV.peekiv`. Less likely, `IV.peek` would be applied only finitely often, perhaps only once. This case can be handled through `IV.take` and `V.peek`.

`IV.peek` would be a compute-phase function, and compute-phase functions are mildly deprecated. They have side effects on `ivecs`, which can result in runtime `Nonmonotonic` errors. However, an infinite number of `peek`s can be rolled together into a single side-effect-free `peekiv`.

Chapter 7

Extensions to Chronic

Chronic was developed to a level that would allow us to evaluate how computer-music applications are written in it, as in Chapter 4. This leaves plenty of openings for future work. Some, such as lazy vectors and `ivec`s with history, are a mere matter of engineering; others, such as mathematical semantics and support for real-time operation, are research projects in themselves.

7.1 Practical necessities

Certain extensions to Chronic are really not optional; they would be essential for it to evolve beyond research-quality software. At the top of the list are support for sampling rates, and for vector and `ivec` comprehensions.

7.1.1 Sampling rates

In audio synthesis, typically a large fraction of the signals are not audio—they are control signals, having a lower bandwidth than audio, and can be represented with a lower sampling rate. Computer music software typically provides a notion of control-rate signals that can be used when full audio bandwidth is not required. Typically the control rate is at least an order of magnitude lower than audio rate, making operations on control signals almost free.

Chronic has no idea of sampling rates, so the programmer gets stuck with the bookkeeping. For example, `LIV.osc_sine` takes its frequency normalized to the sampling rate, so the programmer writes `LIV.lowpass2 (440. /. fs) 0.707 input` (or, even more distracting, code like `(440. *. fs_reciprocal)`.)

Chronic does presently allow multiple sampling rates, though implicitly. Here the `fast` signal is at eight times the rate of the `slow`:

```
let fast = ...
let slow = ...
let t = IV.tabulate (fun i -> i lsr 3)   (* logical shift right: /8 *)
let slow_as_fast = IV.peekiv t slow
let sum = IV.add2 fast slow_as_fast
```

The difference in rates is purely a consequence of how the `sum`'s demand for samples propagates through the `peekiv`. Chronic will notice nothing awry with the nonsensical expression `IV.add2 sum slow`—until it fails at run time.

I envision types `'a sig` and `'a isig`, corresponding to `vec` and `ivec`, which incorporate a sampling rate. Arbitrary sampling rates are attractive, but simplicity may dictate one audio rate and one control rate a submultiple of it, or perhaps a chain of submultiples. These signal types would allow `LIV.lowpass2`, for example, to take its cutoff frequency in Hertz, and normalize by the sampling rate of its signal input. `LIV.osc_sine`, on the other hand, has no signal input, and would need to take its output signal's sampling rate either as an argument or from a global audio rate.

7.1.2 Comprehensions

Section 5.9 demonstrates that an operation such as `map (+.)` is terribly inefficient; it spends very little of its time performing the floating-point addition itself. This is traceable to the fact that the O'Caml compiler does not specialize in order to inline. It will inline ordinary function calls, but applications of a function variable it will not inline, on the grounds that it does not know the function value of the variable. But in the case of `map (+.)`, the function happens to be known at compile time, and we would like to specialize `map`—partially evaluate it on its first argument—and inline `+.` into it.

One approach would be to add this feature to the compiler (or else hope that someone else does it). The reason no one has already done this is that it is not a trivial matter; overenthusiastic specialization can make code size explode.

A less involved approach is to specialize only certain expressions designated by the programmer. Furthermore, we can use the preprocessor to improve readability by writing these expressions as vector or `ivec` comprehensions. Instead of the mildly cumbersome:

```
IV.map2 (fun xi yi -> (xi +. yi) *. 0.5) x y
```

we can write something like:

```
 (xi +. yi) *. 0.5: xi in x, yi in y
```

or even the abbreviated:

```
 (x +. y) *. 0.5: in x, in y
```

The preprocessor will generate an object specialized to perform this particular operation on two `ivec` inputs.

Besides speed and readability, this has the advantage it doesn't depend on the completeness of a family of `map` functions. To tell the truth, `IV.map4i` was created specifically because the FOF code in Section 4.2 needed it.

7.2 Mathematical semantics

This dissertation approaches Chronic as a software system informally implementing certain ideas about temporal type constructors. Appendices C and D describe the software system down to the level of its O'Caml implementation.

An alternate approach is to construct a mathematical semantics of the primitive operations on temporal types, and treat the software system as an attempted realization of that semantics. A static semantics would allow for a basic type-soundness proof. A dynamic semantics might give insight into the exact behavior desired of `ivec`s, and the interaction of build-phase and compute-phase primitives; it might also dictate optimization and program transformation if it offered a broad enough formulation of equivalence. Cost semantics, particularly for space usage, would help greatly to guide the programmer. Semantics of Chronic is a promising line of investigation, but beyond the scope of this work.

7.3 Is history necessary?

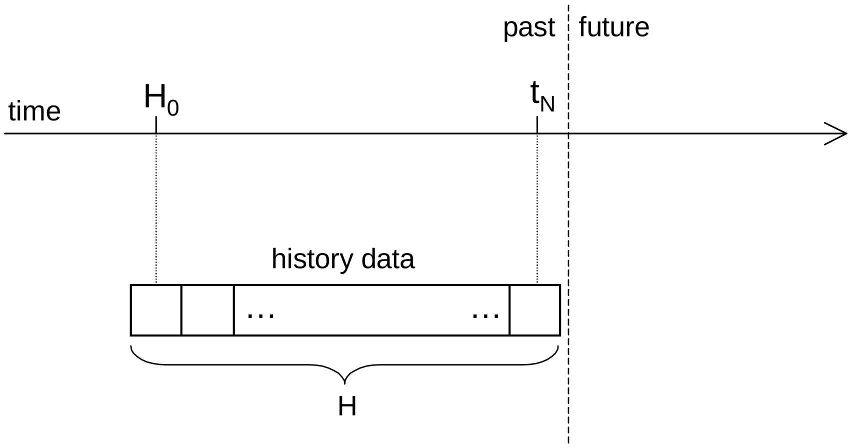
Having to access `ivec`s monotonically can be constricting. For example, if we want to take overlapped subvectors of an `ivec` (such as for short-time FFT spectral analysis), the straightforward way is non-monotonic:

```
let subvecs (len: time) (stride: time) (x: 'a ivec) =
  let offsets = IV.iterate (fun t -> t + stride) 0  in
  let offsetted = IV.map (fun ofs -> IV.drop ofs x) offsets  in
  IV.map (IV.take len) offsetted
(* no good: accesses x[0..len-1], then skips back to x[stride..] *)
```

7.3.1 An `ivec` with a history

How would it work for `ivec`s to allow non-monotonic access? Accesses can't possibly be allowed unboundedly far into the past, as this would force every `ivec` value to accumulate past data forever. So an `ivec`'s history is bounded. This can be measured either in terms of the history's length, H, or its starting time, H_0.

Define the `now` time t_N to be the high-water mark, $\max_i t_i$ if the t_is are the times accessed. For a strictly monotonic `ivec`, with no history, the next access must be to $t_N + 1$ or greater. For an `ivec` with history of length H, it must be to $H_0 = t_N - H + 1$ or greater. The span from H_0 through t_N is the history:



One might make the amount of history either adjustable on the fly, or fixed. Adjustable history is more general. Adjusting H upwards is awkward, because it calls for samples that have already been discarded, so H_0 should be the adjustable parameter, with adjustments non-strictly monotonic. Failing to update H_0 results in a history leak.

Fixed-length history means that whenever a new sample is computed, the oldest sample is dropped from the history, so its length is fixed. This is simpler for the programmer, because it can't leak. It is less general, but adjustable history is not fully general either; only infinite history is. Adjustable history allows the program to decide at runtime how much history it needs, but only when these decisions never involve moving H_0 backwards. For example, in variable-ratio downsampling, the downsampling filter's span over the input is proportional to the ratio, and can reach arbitrarily far into the past at any time—H_0 may have to go backwards.

We will shelve adjustable-history `ivecs` until coming across a compelling application that needs more than fixed-length history but less than infinite.

To build a fixed-length-history `ivec`, simply bundle its parts together:

```
  type 'a ivech = Ivh of ('a ivec) * ('a vec) * time
  (* future, past, length of past *)
```

Then we can provide operations like `peek` on `ivech`:

```
module IVH = struct
  let from_ivec (x: 'a ivec) past =
    Ivh (x, past, V.length past)

  let to_ivec (Ivh (xiv, _, _)) = xiv

  let peek i x =
    let Ivh (Iv (now, future), past, pastlen) = x  in
    let past0 = now - pastlen  in
    if (i < past0) then err_lost "IVH.peek"
    else if (i < now) then (x, past.(i - past0))
    else let (morepast, xi_at_head) = LL.split (i - now) future  in
```

```
        let (xi, future') = LL.split1 xi_at_head  in
        let bigpast = V.joinlist [past; morepast; V.from_scalar xi]  in
        let past' = V.drop (i - now + 1) bigpast  in
        let x' = Ivh (Iv (i+1, future'), past', pastlen)
        in (x', xi)
end
```

Use IVH to build a non-monotonic IV.peekiv:

```
  let peekiv_basis basis t (x: 'a ivec) =
    if (now t) > 0 then err_lost "peekiv_basis"
    else let f x t_i = IVH.peek t_i x
         in sidescan f (IVH.create x basis) t
```

7.3.2 History in the type system?

We now have two distinct types of infinite vectors—unless we entirely eliminate `ivec` in favor of `ivech`, which would penalize the common case of monotonic access. The `from_ivec` and `to_ivec` functions can interconvert the two, but do not eliminate the need to have a full slate of operations for each type: trying to define IVH functions in terms of IV functions loses history information, and defining IV in terms of IVH penalizes the common case.

7.3.3 Living without history

History can often be expressed in terms of `vdelay`. For example, a limited `peekiv_basis` is simple:

```
(* requires _strict_ monotonicity, except with (V.length basis) of slack.
 * (whereas the IVH peekiv_basis was non-strict, with the same slack.)
 * to be more permissive, you'd collapse non-unique t values and
 * re-expand afterwards with peekiv.
 *)
let peekiv_basis (basis: 'a vec) (t: time ivec) (x: 'a ivec) =
  let ([|t0|], t') = IV.spy 1 t  in
  let (xhead, xtail) = IV.split t0 x  in
  let diff = IV.mapi 0 (fun i t -> t0 + i - t) t'  in
  IV.vdelay diff (V.join basis xhead) xtail
```

Then this proves effective in representing other uses of history:

```
let subvecs (len: time) (stride: time) (x: 'a ivec) (exemplar: 'a) =
  let offsets = IV.icount stride  in
  let zigzag = IV.map (-) (IV.icount 0) (IV.stretch len offsets)  in
  let serialized = IV.peekiv_basis (V.const exemplar len) zigzag x  in
  IV.break (IV.const stride) serialized
```

Section 6.2) argued that programs using `IV.peek` can be rewritten in terms of `IV.peekiv`. Likewise, programs using `IVH.peek` can be rewritten in terms of `IV.peekiv_basis`. We have no real need of `IVH` at all. It can still be built on top of `IV`, as shown above, if having history makes a certain program better expressible, but we have left `IVH` out for now.

7.4 Recursive definition of `ivec` values

In Chronic as described in Chapter 3, feedback delay is written in terms of the `vdelay_rec` family of functions:

```
let (x_with_echoes: float ivec) =
  IV.delayz_rec echo_len (IV.map2 (fun xi yi -> xi +. 0.5*.yi) x)
```

This is not the most natural way of thinking. We would like to be able to write feedback delay directly, more like this:

```
let rec (x_with_echoes: float ivec) =
  IV.map2 (fun x y -> x +. 0.5*.y) x (IV.delayz echo_len x_with_echoes)
```

The value `x_with_echoes` is defined in terms of itself. Strict languages do not allow recursive values of arbitrary type, so if we want to write feedback delay directly, we need to make `ivec` one of those types that do. Function types, of course, may have recursive function values; we choose to make `ivec` a function type.

The O'Caml compiler has a `-rectypes` option, which allows arbitrary recursive types (unmediated by a constructor); normally this is only allowed with object types. We take advantage of this to define `ivec` as a non-strict list type (as in Section 6.1.1) with a function type outermost:

```
type 'a ivec = unit -> 'a * 'a ivec
```

Now we can define `x_with_echoes` almost the way we wanted:

```
let rec (x_with_echoes: float ivec) () =
  IV.map2 (fun x y -> x +. 0.5*.y) x (IV.delayz echo_len x_with_echoes) ()
```

The only difference is the `()` on the left-hand side, to make `x_with_echoes` a function, and the matching one on the right-hand side. That is, we have eta-expanded the original definition to get it into the proper form.

7.4.1 Where recursive definition breaks down

We have shown a recursively definable `ivec`. It behaves as a non-strict list, which lacks the properties that we developed in Section 6.1. What happens if we try to reintroduce these?

The `ivec` from Section 6.1.3, which had consumer-controlled block length but not efficient fan-out, can be given an function-outermost form:

```
type 'a ivec = 'a vec -> time -> (time * 'a ivec)
               (* out upto -> (now, future)  *)
```

Then it should be possible to have recursive definitions:

```
let rec (x_with_echoes: float ivec)
        out now =
  IV.map2 (fun x y -> x +. 0.5*.y) x (IV.delayz echo_len x_with_echoes)
  out now
```

The function-outermost restriction prevents us, however, from proceeding with efficient fan-out as in Section 6.1.4, where `ivec` is required to be `ref`-outermost. Furthermore, we ultimately want `ivec` to support not just a `compute` call, but also a `seek`—you would have to multiplex these into a single function type. This all is a good deal of trouble to get into for the reward of recursively-definable `ivecs`.

7.5 Will nothing rid us of these troublesome exemplars?

Most primitives that return some `'b ivec` require that the caller give them an exemplar of the type `'b`. For example, `IV.map` is of type `'b -> ('a -> 'b) -> 'a ivec -> 'b ivec`; the first argument is the exemplar. A call to `IV.map` might look like `IV.map 0. float_of_int my_ints`. The `0.` is arbitrary; any `float` value will do.

Having to pass in these exemplars is annoying. What are they for, and can't we get rid of them?

At initialization, `map_dat` has to set up its output buffer as a length-`maxblocklen` vector, but it doesn't care what's in the vector—it will be be overwritten before it is read. O'Caml, though, doesn't allow uninitialized arrays. We know the type `'b`, but we need an instance of it. The exemplar serves this purpose, so the output buffer can be initialized as `V.const exemplar maxblocklen`.

Some `IV` functions do not need an exemplar. `IV.const`, for example, can use `xi` instead. In fact, all of the usual `ivec` generators (functions that return an `ivec` but do not take one as an argument) can do without an exemplar. This suggests that perhaps they could package an exemplar with the `ivec` they generate, and other functions would propagate it along.

Then `map_dat` is handed an exemplar for `'a`, but it needs one for `'b`. It might try to apply the function being mapped, `f: 'a -> 'b`, but what if `f` is not a total function; what if it, say, divides by zero at this point?

The purest solution to this problem would be to make the array elements be of an `option` type, so they could always be initialized to `None`. However, as Chronic's inner loop dwells heavily on these arrays, for speed we would rather only take this approach within a type system that would make the compiler able to optimize the `option` away in the common case.

Another possibility is to add a "create uninitialized array" primitive to O'Caml. This is unsafe, but arrays are already allowed to be unsafe.

7.6 Non-time axes

Section 2.2 includes vectors that are indexed by frequency rather than by time. Some vectors and `ivecs` are indexed by the bare ordinal numbers, with no unit of measurement. A vector might also be indexed by spatial position along one of several axes.

Chronic's type system does not distinguish among these. Doing so might catch programmer errors and clarify code. We might consider adding a "units" field to each vector, `ivec`, and event—dimensionless, samples, samples^{-1}, pixels, furlongs, fortnights, and so forth. However, the case of multiple spatial axes suggests that "pixels" alone is not enough; you want to carry information about the axis. So make it an "axis" field.

One's taste for type distinctions may determine whether it seems better to work to add this information right off, or to examine time/frequency code such as the pitch shifter in Section 4.5, and spatial code that has not been written, to see how it looks with and without axis information.

7.7 Lazy vectors

Making `'a vec` a synonym for `'a array` means that vectors are strict in all of their elements. Sometimes we would prefer that vectors be lazy. Very long vectors, for example, need not be allocated and computed all at once. Furthermore, if Chronic is ever going to operate in real time (for more on which, see the next section), lazy vectors are essential.

Consider the dependencies in the FOF synthesis code from Section 4.2. Look at an output sample that aligns with the first sample of a grain. A strict-vector grain will become available to `vmix` only all at once, when its last sample is computed. The output sample depends on the grain's last sample, some time in the future. The FOF code is acausal. If that last sample depends on a control signal derived from real-time input, then the only way to run this code is to delay its output by the maximum length of any grain—if that bound even exists.

The grain should be a lazy vector, instead. This breaks the dependency of its first element on its last, alleviating the problem.

Similarly, what if we are reading Section 2.2's gestural control type, `(pitch vec * pressure vec) event ivec`, from a real-time controller? With strict vectors, we don't get to hear about the beginning of a note until it has already ended.

Some vectors are accessed monotonically, like `ivec`s. A lazy vector of this kind is simply an `ivec` with an ending time attached. Other vectors are random-access. We would like to distinguish the simple case when it occurs, but this probably means two separate types, and without subtyping, this becomes painful.

A random-access lazy vector might again begin with an `ivec`-like engine, but now be augmented with an unbounded history, a `rewind` method, and a `poke` method. The `rewind` method enables `peek` and `subvec` operations, but efficiency suggests `peekv` and `pokev` methods as well.

A different approach would be to build the lazy vector as a finite lazy list of arrays, or as a finite version of any of the `ivec` representations in Section 6.1. It may be that all of the reasons driving the design of the `ivec` still apply to the lazy vector. Consumer control of block length is valuable here because it translates to perfect laziness: no unnecessary forward dependencies, a valuable property. With producer control, I may request up to time t, but get up to time $t + d$, which means that my time t is forced to depend on the producer's time $t + d$.

7.8 Real-time computation

The previous section showed that making Chronic work in real time would require lazy vectors. This is not the only requirement. It also needs to change its handling of `event ivecs`, so that commonly-needed code can be written causally. There is also the broader problem that the programmer may write acausal code by accident.

`EIV.vfold` is an example of the `event ivec` problem. It needs to know when it can just proceed with folding the currently-live grains, versus when it has to take a new grain into account. It does this by pulling the next `vec event` off the `vec event ivec`, and examining its timestamp. Up to that time, proceed with current grains.

This is a forward dependency. Even if the `event` were lazy, it wouldn't help; `vfold` examines the timestamp, which cannot be known until the event comes into being.

One possibility is to fuse `event ivec` into an `event_ivec` type. This type would then support queries of the form "is the next event at a time less than t_1?", which `EIV.vfold` could poll with. Stepping t_1 by single samples means no forward dependency; larger steps will mean a certain amount. A pleasant side effect is that since an `event_ivec` must reasonably have monotonic event timestamps, we might leave `event ivec` for non-monotonic timestamps, giving us a way to carry that information.

However, the type distinction between `event_ivec` and `event ivec` is generally not desirable; it means code duplication and special handling for this new `event_ivec` type. We might instead leave the `ivec` alone and alter just the `event`. What `vfold` gets is a "potential event", which it polls with "is your timestamp less than t_1?" until the event happens. One might even use the new kind of event everywhere, replacing the present kind, to keep the type system simple.

The changes to `event ivecs` and to vectors are necessary because without them the programmer has no way to write causal code. Even with the changes, though, there is the risk of writing acausal code by accident. This is the dark side of the *sub specie æternitatis* model: when you're standing outside of time, a forward dependency is essentially no different than a backward dependency. Ideally, a type system would enforce causality. This seems difficult to do usefully, and remains an open problem.

Chapter 8

Related work

Chronic enters a crowded field. There are any number of ways to program with temporal data, from a variety of backgrounds. This survey of related work is 'Chronic-centric' in that it considers not how well each work meets its own goals, but how well it meets Chronic's goals: general and expressive music programming. Coverage becomes less complete the further from Chronic we get.

Some of the following are programming languages, some are domain-specific sublanguages within another language, and some are frameworks or libraries. These distinctions are important only through their influence on the way one programs.

One angle to consider is the type system. Are there temporal type constructors, or are temporal types provided as primitive types? I've argued that constructors are far more useful. Is temporal data even first-class? Infinite vectors are the kind that most often isn't.

A broader angle is the language in general. Chronic relies on an applicative style of programming, one suited to concise manipulation of constructed types. This style requires polymorphism and higher-order functions, or some substitute. What style does the language support? Does it restrict the programming model in any way—by being only *sub specie temporis*, or by lacking practical necessities like scoping or modularity?

The following taxonomy of related work is in terms of its temporal type system first, and then other language characteristics. Here "signal" is an umbrella term for any type usable as a finite or infinite one-dimensional signal, and "event" for any type usable as a note event.

8.1 Languages with signal and event types

Only Fran and ALDiSP have temporal type constructors of any kind. The other languages provide their signal and event types as atomic types.

8.1.1 Languages with temporal type constructors

8.1.1.1 Fran

Fran, for "Functional Reactive Animation", is a domain-specific language, embedded within Haskell, for graphical animation. Its temporal type constructors are `Behavior` and `Event`.

An α `Behavior` is, semantically, a function `Time` $\rightarrow \alpha$. `Event` has had two different meanings. Originally (Elliott and Hudak 1997), an α `Event` meant a non-strict pair `Time` $\times\, \alpha$. In later work (Elliott 1998), an α `Event` is instead a time-sorted sequence of such pairs.

Continuous-time `Behaviors` are strictly more general than `ivecs`, so why does Chronic use discrete time? The two type systems developed independently, one intended for animation and the other for music. Perhaps the most fundamental difference is that discrete-time animation is merely an expedient and more or less *ad hoc* approximation to continuous-time animation, whereas discrete-time signal processing is a mathematical discipline in its own right. Digital filters and the Fast Fourier Transform are inherently discrete-time. DSP makes pervasive use of the assumption of a constant sampling frequency.

Full-blown continuous-time signals are inefficient to compute with. A continuous-time signal has infinite information content over any finite segment, so it can hardly be stored in a delay line. A continuous-time delay operation cannot be reduced beyond the following:

```
type time = float
let time_zero = 0.
type 'a behavior = Behavior of (time -> 'a)

let delay (d: time) (Behavior x) =
  Behavior (fun t -> x (t -. len))
```

So if y is a delayed x, then $y(t) = x(t - d)$. Furthermore, if y is x with repeating echo applied, then $y(t)$ depends on $\{x(t), x(t-d), x(t-2d), \dots\}$. The only efficient way out is to memoize $y(t - d)$ values, which could take unbounded space unless t is constrained.

Once it is necessary to constrain time, a type that leaves it unconstrained becomes a weakness in the type system. The choice of constraints encoded into Chronic's types, non-negative integer time, is stronger than necessary, but a constant sampling rate is well-suited to its domain, music.

The question remains of whether to use a function-of-time semantics. Chronic's vectors can, semantically, be considered (partial) functions of time, but their realization is as arrays. This is more practical, given that the choice of integer time has made it possible. Chronic's `ivecs` depart from the function-of-time semantics for reasons explained in Section 6.1.

Fran programs could use discrete-time conventions that would let them do Chronic-style programming with their `Behaviors`. In practice, they do not seem to. This simply reflects that Fran and Chronic are domain-specific languages for different domains. Their use of continuous versus discrete time is a consequence, not a cause, of the rather different ways they propose to program with temporal data.

8.1.1.2 ALDiSP

ALDiSP (Freericks 1996) is an functional language intended for real-time DSP and asynchronous event handling. It includes both ordinary demand-driven lazy lists, and input-driven *pipes*, chains of suspensions, suitable for streams of asynchronous events. This event-stream constructor represents not the time-structured data, but a program that reacts to data. Each event is viewed *sub specie temporis*.

ALDiSP's compiler has the challenging task of compiling a complex semantics to a real-time program for a restrictive DSP platform. It takes an interesting but resource-intensive approach: abstract interpretation of the program covering all possible execution paths, to generate an explicit code form.

8.1.1.3 Theoretical work

Dannenberg, Desain, and Honing (1997) survey the problem of how to represent music in a programming language. This work does not talk in terms of type constructors, but is mentioned here for its wide-ranging discussion of the structures and relationships needed to represent music. The complexity of these structures is part of the motivation for temporal type constructors.

Chronic does not explicitly address all of the problems described here. For example, the *vibrato problem* is the problem of knowing what to do when a note having pitch variation is transformed so as to be longer: stretch its pitch contour (for a glissando), or generate an appropriate extrapolation (for vibrato)? And what is the appropriate extrapolation? Chronic does not build in any solutions to this, though as a language it could support any of the approaches described here.

8.1.2 Languages with events *sub specie æternitatis*

Languages may also have some notion of signals and of events *sub specie æternitatis*, but as atomic types, not constructed. Here events are first-class data that can be manipulated *sub specie æternitatis* (in the terminology of Section 2.3) within the language.

Arctic (Dannenberg, McAvinney, and Rubine 1986) is a first-order functional language for real-time programming. Arctic's `signal` resembles a Fran `Behavior`, but is specifically real- or boolean-valued rather than polymorphic. Arctic has two sorts of event-like constructs: times as parameters in signal instantiation, and conditions whose satisfaction by a boolean `signal` triggers, for example, an `until` expression to splice *before* and *after* signals. Neither of these constructs are inarguably first-class values, but they do exist in expressions and provide a *sub specie æternitatis* declaration of temporal behavior. Arctic seems to have been the first programming language to do this.

Nyquist (Dannenberg 1997c; Dannenberg 1997b) instantiates a sound (a `float vec`) from a behavior, which has a start time (among other times) attached to it. In this way a Nyquist behavior is something like a `(() -> float vec) event`. Behaviors can be, for example, sequenced one after another, or shifted in time. (These transformations are actually done by altering an environment from which *shift* and other values are read when the behavior is instantiated to a sound.)

SuperCollider (McCartney 1996) is a Smalltalk-like language. It is typeless, but by convention an event is an environment (a symbol table) that binds the parameters for a musical note. A lazy sequence of events forms a score; this is first class data, and can be generated or processed in any way desired.

8.1.3 Languages with events *sub specie temporis*

Other languages still have atomic signal and event types, but events can be manipulated only *sub specie temporis*.

The original Max programming language (Puckette 1988) did not have signals, but its descendants FTS (Puckette 1991), MSP (Cycling '74 2001), and PD (Puckette 1996) do. These are static dataflow languages, represented visually as flow networks. Events (commonly MIDI data) are processed in real time as they travel through the network.

Aura (Dannenberg and Brandt 1996) uses a similar architecture, with an emphasis on real-time capabilities. It, however, is a C++ framework rather than a visual language.

8.1.4 Languages where events are not first-class

Finally, some languages have signals and some kind of non-first-class events. The MUSIC-N family of computer music languages, including such modern descendants as Csound (Vercoe 1993) and MPEG-4 SAOL (Scheirer 1998; Vercoe, Gardner, and Scheirer 1998), has audio and certain spectral types atomically. Events exist only in a separate mini-language, the *score language*, which is very close to a static list of notes; it does not support variables, expressions, or functions.

The *orchestra language* defines instruments, which perform synthesis and processing when instantiated by score events. Within each instrument, the language is static dataflow, operating on audio and control signals by passing them through operators, called here "unit generators". This makes its model of time *sub specie temporis*.

The score/orchestra dichotomy means that musical events cannot be spoken of in the orchestra language, but also that control signals cannot be spoken of in the score. The arguments to an instrument, with which it is instantiated, are scalar values only—the score can't hold a pitch glissando or a bow-pressure envelope, because these signals can't live in the score. The score can reference stored function tables, but, conversely, these tables cannot be filled from a signal generator in the orchestra language. The score language contains a hard-wired set of table generators.

8.2 Languages with signals only

Certain languages do not have any built-in notion of events, but do have some kind of regularly-sampled data usable for signals: arrays, matrices, or streams. Of course, many languages have arrays, but this discussion is restricted to languages with a reasonably declarative style of programming with them. Signals here are often not polymorphic.

8.2.1 Languages with streams

Sisal (McGraw et al. 1985) is a first-order functional language designed for parallel scientific computation. It has performance competitive with Fortran's, and signal-processing code has been written using its stream types. SERT (Nico, Feo, and Yates 1997) extends Sisal with pragmas for specifying timing. These let real-time programs constrain latency and throughput on streams, and skew between streams, giving the compiler freedom to reorder computation within these constraints.

Lucid (Ashcroft and Wadge 1977) is a single-assignment language with polymorphic streams. It has only scalar operators, but an unusually flexible view of time for a stream-based language. The most direct implementation is lazy dataflow.

EIN (Lansky and Steiglitz 1995) is a mini-language for DSP; its key feature is the construction of feedback delay structures. The language is an augmented C that gets wrapped in a "for each sample" loop; the augmentation is a "tap to numbered delay line" construct that always taps from the value `y`, and correspondingly-numbered delay output variables. This is sufficient to represent many interesting delay structures (with fixed delay lengths), though not in a modular or spectacularly readable manner.

FAUST (Orlarey and Letz 2002) is a DSP language based on a block-diagram algebra. This algebra includes operators for parallel and serial composition, recursive delay, and composition of subdiagrams inside recursive delay. From this algebra a compiler can generate efficient code.

8.2.2 Reactive dataflow languages

A reactive system (Harel and Pneuli 1985) is one whose actions, presumed instantaneous, are driven solely by its environment. Reactive languages (Halbwachs 1993) are intended for developing this sort of system. They place a heavy emphasis on safety and formal verifiability, to the point of limiting the programming model to a finite automaton; in fact, visual languages such as Statecharts (Harel 1987), Argos (Maraninchi 1991), and SyncCharts (André 1996) are Mealy automata given hierarchical structure for expressiveness.

The input-driven nature of reactive systems makes it natural for these languages to use implicit time. Esterel (Berry and Gonthier 1992), an explicitly parallel imperative language for "control-dominated" or event-based systems, needs no notion of time other than *now*.

Lustre (Halbwachs, Caspi, and Pilaud 1991) and Signal (LeGuernic, Benveniste, Bournai, and Gautier 1985; Gauthier, LeGuernic, and Besnard 1987) are both synchronous dataflow languages for "data-dominated" or signal-based systems. Lustre has a *basic clock* which defines atoms of time; every value is then a *flow* sampled over time, with an associated clock specifying an arbitrary subset of basic ticks at which a sample exists. Operations on these flows include *delay-by-one* and zero-order *sub-* and *supersample*. Signal is quite similar, except that whereas Lustre is functional, Signal uses bidirectional constraints.

Chronic is not intended to be verifiable, and hence can have a broader scope: Turing-complete, rather than bounded-memory, and with full constructors rather than a flat type system.

Silage (Hilfinger 1985) comes from a different direction than the reactive languages, but it has a similar programming model: bounded loops, no recursion, synchronous dataflow. It supports fixed-point arithmetic and is used for DSP.

8.2.3 Languages with matrices

Matlab (The MathWorks, Inc. 1997) is a widely-used matrix programming language. As it is widely used, as a point of reference let's look a bit at how it serves as a computer music programming language. Its matrices are finite, and it is not intended for streaming processing of real-time audio or audio that does not fit in main memory. Matlab is commonly used for DSP, often for prototyping algorithms to be implemented finally in C. This it does well, since its vector arithmetic and DSP library can handle many operations, and explicit looping over samples can take care of the rest. Looping is expensive, though, since Matlab is normally interpreted (and must be to support higher-order functions, which are realized through evaluation of strings containing code text).

Beyond straight DSP and on to computer music, Matlab becomes more cumbersome. Like Chronic, it has no notion of sampling rates, requiring bookkeeping by the programmer; and whereas Chronic at least has implicit support for multiple rates (as in Section 7.1.1), Matlab does not—no control signals. As for temporal types, current versions of Matlab have added records and inhomogenous matrices to the type system, so temporal structures could be built, but Matlab would provide no primitives for dealing with them. As a result, many DSP designers use Matlab, but few composers.

NESL (Blelloch 1995) is a first-order functional language whose goal is the expression and efficient implementation of data-parallel algorithms. Its nestable vectors are equipped with vector primitives that operate in parallel, leading to competitive performance on vector multiprocessor platforms.

FISh (Jay and Steckler 1998) is an Algol-like language with multidimensional arrays, over which functions are polymorphic. This is achieved through *shape analysis*, compile-time partial evaluation of array dimensions, which must therefore be statically known.

SAC (Scholz 1998; Scholz 1996a; Scholz 1996b), "Single-Assignment C", is essentially a single-assignment C without pointers or globals but with matrix operations from the ψ-calculus (Mullin 1991). It is intended for high-performance scientific computation, for which its performance is roughly comparable to Fortran or Sisal.

SRL (Kopec 1985), for "signal representation language", is a first-order functional language (without general higher-order functions) with signals in the form of finite multidimensional arrays. What SRL terms a "signal type" is an OOP class inheriting ultimately from **basic-signal**, which upon receipt of its parameters instantiates a signal. Signals support, among other inquiries, a method to `fetch` a sample; this method might `fetch` a sample from a signal bound to a parameter, operate on it, and return the result. Array fetches and signal-valued signals provide the ability to write more involved algorithms in an elegant way: overlap-add convolution is an example given.

8.2.4 Libraries for low-level languages

Some libraries are general DSP libraries, providing fundamental DSP operations such as filtering, tone generation, and the FFT. Others are specifically intended for computer music. These will include higher-level functions such as reverberation or physical-modeling synthesis, and will be able to be controlled by some sort of a musical score. Libraries designed for computer music and operating on more-complex temporal data do not appear to exist.

We take the Intel Signal Processing Library (Intel 2002) as an example of a DSP library. Its functions operate on single samples and on vectors. For infinite vectors, the programmer calls the vector functions and carries state from one block of computation (one vector) to the next.

STK (Cook and Scavone 1995) is a C++ library for audio synthesis and processing. A control protocol called SKINI drives it with events from a score or real-time controller. The programming model is like that of the MUSIC-N languages, except that the synthesis language is the same as that in which the unit generators are implemented, giving it access to individual samples.

CMix (Lansky 1990) combines a set of executable utilities, a C library for audio synthesis and processing, and a mini-language with variables and control constructs used to define the score. This mini-language gives its programming model the additional ability to create scores algorithmically.

8.3 Languages with events only

The only languages considered here are ones which treat events *sub specie æternitatis*; languages not meeting this criterion exist in great number, including, for example, all discrete event simulation languages.

The Stream mini-language (Letz, Orlarey, and Fober 1995) deals with streams of MIDI events, constructed through lazy operations (`cons` and `merge`) from constants or from a stream representing MIDI input. Thus it puts forth the idea of writing real-time interactive programs *sub specie æternitatis*. [*mention Conal's related work?*]

Haskore (Hudak et al. 1996) is an "algebra of music" embedded in Haskell. It defines a recursive Haskell data type that represent a score in terms of notes and rests, pitch classes and octaves, beats and tempo, and operations on these such as transposition, serial and parallel composition, and annotation for the player. The score is defined in terms of discrete events, not sampled control signals.

Chapter 9

Conclusions

This concluding chapter first summarizes those preceding it; if they're fresh in your mind, you might skip the summary. Following this summary, I assess the successes, failures, and possible applications of my work, and pick out the ideas that have the broadest importance.

9.1 Summary

This dissertation has put forth a programming style targeted to the domain of computer music programming. This style, applicative programming with temporal type constructors, combines expressiveness and generality in a way that existing approaches, from C++ to specialized computer-music languages, have not.

9.1.1 Temporal type constructors

Temporal type constructors are type constructors that introduce temporal structure. The three proposed are event, vec, and ivec. An "α event" is an α at a single specified time. An "α vec" is a finite sequence of αs, and an "α ivec" is an infinite sequence of αs.

α event	an event whose timestamp is of type Time.
α vec	a finite vector, indexed by Time.
α ivec	an infinite vector, indexed by Time.

This set of constructors forms a rich variety of musically useful types. For example:

- Digital audio: Sample ivec
- Multichannel audio: Sample ivec vec
- Short-time spectral data: Complex vec ivec
- A musical score with instantaneous note events: (Pitch $\times$ Velocity) event ivec
- A score with continuous control of each note: (Pitch vec $\times$ Pressure vec) event ivec
- A stream of fragments of audio: Sample vec event ivec

This last type is a good example of synergy between vectors and events. It allows irregular segmentation of audio, with overlap, underlap, both, or neither. It can be used to cue and mix audio samples, or to represent sparse signals. It allows processing of selected pieces of a signal in different ways. A whole family of techniques, such as FOF vocal synthesis and formant-corrected pitch shifting, involve arranging small grains of sound. By use of temporal type constructors, Chronic can express these and many other temporal programming tasks with great clarity and power.

9.1.1.1 Models of time

In a type system that lacks temporal information, the only time is "now". Code runs to generate the current sample, or in response to the most recent event. Audio processing code is written as if signals were scalars—in Csound, the code to negate an audio stream is:

```
anegated = -ainput
```

and this scalar code is implicitly wrapped in a loop. Chronic uses values with temporal extent, and is explicit about the looping:

```
let (negated: int ivec) = IV.map (fun x -> -x) input
```

Chronic's explicitly temporal types allow the code to stand outside of time and do with it as it will, so we call its model of time *sub specie æternitatis*, and Csound's *sub specie temporis*. The *sub specie æternitatis* model is often more closely aligned with how the programmer thinks about an algorithm. Whenever we use a graph with a time axis, we are thinking in the *sub specie æternitatis* mode. Also, it is more powerful, as it allows reference to the future; for example, it allows the `IV.drop` operation, dropping the leading elements in an `ivec`.

9.1.2 Chronic

Chronic is an O'Caml library for applicative programming with temporal type constructors. It consists of core modules, each dealing with one kind of temporal type, and a standard library of computer-music primitives:

module	type
`E`	`event`
`V`	`vec`
`M`	`vec vec` (matrices)
`IV`	`ivec`
`EV`	`event vec`
`EIV`	`event ivec`
`L`	mostly scalars
`LV`	`vec`
`LIV`	`ivec`

`E` has functions to create and dismantle events. `V` and `IV` have the usual applicative-programming primitives: `iterate`, `map`, and so forth. `IV` also includes a family of recursive-delay operators that take a user-specified feedback path, for defining values like $y =$ delay *length basis* $(f\ y\ \ldots)$

`EV` deals with vectors of time-ordered events. It can convert them to piecewise-constant or piecewise-linear vectors, or, if the events are vectors themselves, vertically fold overlapping events. `EIV` is the `ivec` equivalent.

9.1.3 Applications written with Chronic

This work stands or falls on whether it works for writing computer music programs. Chapter 4 gives code examples written in Chronic for several plausible applications: musical scores, FOF synthesis, Moorer and Gardner reverberators, and a pitch shifter.

9.1.3.1 Musical scores

Chronic does not predefine any particular score format. This is a feature. The user can build, manipulate, and synthesize from any desired time-structured data format. At its simplest, this may be a hand-entered vector of note events. The examples demonstrate further possibilities: time-varying tempo maps, hierarchical structure, motivic cross-references with variations, algorithmic generation, incorporating external graphical editors. Note events may reference more-global control signals; they may even decide that the physical beginning of the sound protrudes before the timestamp on the note.

Leaving the score to be defined by the user works well if the language is capable of providing whatever score feature the user needs. The user can take advantage of a full programming language within the score language, giving it great power.

9.1.3.2 FOF synthesis

Chronic's style of programming applies beautifully to FOF (Rodet 1984), a granular synthesis algorithm used for a pitched signal with a spectral peak, such as the human voice. It involves a stream of sinusoidal grains with carefully controlled pitch, envelope, and onset spacing. The stream is irregularly timed, and each grain is a sequence of samples: this stream of grains is a `float vec event ivec`. In Chronic we can work directly with this type.

What's pleasing about the resulting code is that it expresses our idea of the FOF algorithm. It builds grains one by one and places them in time. It does not keep track of when *now* is and what grains are sounding and which should advance to the next stage of their envelopes. We think about the FOF algorithm *sub specie æternitatis*; finally we can write code for it that way too.

Lines of code is admittedly a crude metric, but it may be interesting to note that the Chronic implementation is about one fifth the size of the one written in C for Csound, or one third if we include the utility functions. More importantly, its expression of the algorithm is more direct and more comprehensible.

9.1.3.3 Two reverberators

A reverberator creates a decaying tail of echoes with dense but irregular spacing. We tried implementing two of the many schemes for doing this, Moorer's (Moorer 1979) and Gardner's (Gardner 1992).

Moorer uses two building blocks: the first-order allpass filter and the comb filter with lowpassed feedback. Neither of these is a primitive in Chronic, but both can be built from the recursive-delay primitive `IV.delayz_rec2` with an appropriate feedback function. The allpass filter does require a bit of block-diagram rearrangement to make it fit `IV.delayz_rec2`'s schema. Chronic handles the bank of parallel lowpassed-feedback combs nicely, as a vector of filters generating a vector of outputs, to be mixed down.

Gardner uses a more exotic structure: the nested allpass filter, where the delay line in a first-order allpass is replaced with another allpass system. This turns out to be no harder to build than a simple allpass. But when it comes time to put the whole reverberator together, Chronic stumbles over a more mundane point: `IV.delayz_rec2` presents output from the point after the delay and before the feedback function; here we need to tap from two points *inside* the feedback function. The workaround is to replicate outside the loop a large chunk of the feedback function, which is something approaching twice as slow, and far from pretty.

These reverberators can be represented in Chronic, but they point out the weaknesses of its approach to handling feedback delay networks.

9.1.3.4 A frequency-domain pitch shifter

To pitch-shift a signal is to transpose its musical pitch without changing its length. I implemented a polyphonic pitch shifter based on the short-time Fourier transform, which involves taking overlapping frames from the input data, multiplying each frame by a window function, and applying the FFT to each. The pitch shifter manipulates each frequency-domain frame. Then the inverse STFT involves an IFFT on each frame, windowing again, and mixing down the overlapping frames. The Chronic implementation is a close translation of Stephen Sprenger's C version (Sprenger 1999). It includes an FFT implementation and a complex-arithmetic library.

The pitch-shifting code is not dramatically clarified by being in Chronic instead of C. Chronic's advantage over C is that the pitch shifting occurs in the context of a high-level language. It can be factored into general-purpose spectral-manipulation functions, communicating in constructed types. Then the analysis functions can readily be broken out and used for a new purpose, such as generating a score where orchestral instruments track the partials of a voice sample, or applying different spectral manipulation within the the STFT analysis/synthesis framework.

Inadequate type systems discourage these reusable subcomponents, because they cannot represent the types that appear at subcomponent interfaces: "stream of short-time spectra", for example, is the output of the STFT analysis function. If your only type is "audio", you cannot represent the analysis or frequency manipulation or resynthesis functions. You are forced to fuse them all together into the pitch shifter, which has merely audio input and output.

9.1.4 Chronic's implementation

Most of the complexity in Chronic's implementation relates to `ivecs`. The rest is simple: time is an integer, events are pairs, and vectors are arrays—though Chapter 7 gives reasons to go beyond these representations.

9.1.4.1 Inside an `ivec`

It is useful to look at `ivecs` as being handled in two phases, *building* and *computation*. The `IV` API in Chapter 3 is for the building phase; the computation phase is private. In this way, an API in a functional style is layered over an implementation in a different style, which might be painful to work with directly: for speed, our implementation makes heavy use of imperative update, but the API does not reflect this. It is important that computation may invoke further building, so Chronic programs are not limited to static dataflow graphs.

An `'a ivec` is implemented as a reference to an object. This object is an `['a] ivec_dat`, or some subclassed object that through subtype coercion can be viewed as such. The building phase constructs a network of these objects referencing each other.

Computation is demand-driven dataflow, performed in variable-length blocks, with the consumer specifying the output block length. It proceeds through successive calls to `compute`, part of an `ivec_dat`'s public interface:

```
get_buf: unit -> 'a array
compute: time -> unit                        (* upto: time *)
seek:    time -> unit
```

The `seek` method advances time without generating output, so it is permitted to step more than `maxblocklen` ahead of the current time. It may, for some `ivec_dat` subclasses, do less work than `compute`.

The argument to `compute` is absolute time, not relative: it is the time up to which the caller wants data. It is non-strictly monotonic—that is, in successive calls, the time argument may stay the same, but may not decrease—in order to support fan-out to multiple readers. Multiple readers must all advance time in lockstep.

This synchronization requirement on multiple readers is made so they can share data; the source does not have to recompute for each reader. When readers cannot meet this requirement, sidestep it by giving each reader a private copy of the source. This is most simply done by wrapping it in a `unit -> ivec` function and re-instantiating.

9.1.4.2 Subclassing an `ivec_dat`

Internally, what `compute` does is to raise an exception if time has gone backwards, do nothing if time is unchanged, and call the private method `compute_hook` to do the work if time has advanced. This method is abstract in `ivec_dat`.

Then `map_dat`, for example, subclasses `ivec_dat`. Its `compute_hook` requests data from `x`, and maps `f` over `x`'s output buffer to fill `map_dat`'s own output buffer. Also, `map_dat` assumes that `f` is free of side effects, so it provides a `seek` that will not request unnecessary elements of `x`, nor apply `f` to them.

Associated with `map_dat` is a builder function, `IV.map`. It creates a new `map_dat` with the given arguments, coerces it to the `ivec_dat` of which it is a subtype, and returns a reference. Through this subtype coercion, all builder functions can return an `'a ivec`.

9.1.4.3 Recursive delay

We would like to allow cycles in the dataflow graph, as long as each cycle contains a delay. A cycle corresponds to a group of `ivec`s whose definitions are mutually recursive. We would like to use `delay` to define, for example, this 1-cycle:

```
let rec nats = IV.map succ (IV.delay 1 [|0|] nats)
(* [|0|] is the basis: effectively, y.(-1) = 0 *)
```

This is not legal in O'Caml, as `ivec` is not a type that can be defined recursively. In lieu of this, Chronic fuses the feedback cycle into a single dataflow node, a higher-order node which takes the feedback path as an argument of type `'a ivec -> 'a ivec`. Using `delay_rec`, we can write

```
let nats = IV.delay_rec 1 [|1|] (IV.map succ)
(* nats = [1, 2, 3, ...] *)
```

The `_rec2` functions are implemented in terms of the `_rec` ones. However, it is not as simple as partially applying the feedback function:

```
let delay_rec2 d basis (f: 'b ivec -> 'a ivec -> 'a ivec) (x: 'b ivec) =
  delay_rec d basis (f x)
```

In fact, it is an error for a feedback function to refer to an `ivec` that is not one of its arguments. The feedback function `(f x): 'a ivec -> 'a ivec` commits this error: `x` is a free variable. This situation is an error because of problems with differing-length blocks: Chronic's dataflow protocol breaks down when an `ivec` has consumers who ask for blocks with the same end time, but different starting times.

A property sufficient to avoid this is that each `ivec` operator propagates its consumer's choice of block length upstream unchanged. Any operator that violates this can be made safe by wrapping it in a master/slave pair of reblocking primitives which Chronic provides.

9.1.4.4 Performance on static dataflow arithmetic

Measuring Chronic's performance is of limited use, because the style of programming illustrated by Chronic is of more interest than Chronic itself. Still, some mini-benchmarks can confirm and quantify our ideas of where the time goes in a Chronic execution.

The core of the `map2` benchmark is `let z = IV.map2 0. (+.) x y`; the core of `add2` is `let z = IV.add2_float x y`. The difference between them is whether the per-sample arithmetic is inlined or is through a function call. Furthermore, we compare with a C++ benchmark.

Normalized CPU consumption

block length	`map2`	`add2`	C++
1	42	31	1.75
256	9.3	1	0.36

These numbers quantify two things we probably expected: first, function calls are costly. Between `map2` and `add2`, we can see that an O'Caml function call is about an order of magnitude slower than an arithmetic operation. The per-sample work must never include function calls. Second, forcing Chronic to compute with a short block length, as feedback delay can do, is deadly.

The `add2` mini-benchmark, static dataflow computation with inlined arithmetic, is about three times slower than comparable C++ code if we run at what turns out to be the optimal block length for both, 256. This is talking largely about the execution speeds of O'Caml versus C++, which is ultimately not a subject of interest here.

9.1.5 Decisions made in implementation

One's first thought is that representing `ivec`s is trivial: an `ivec` is a lazy list. This has the correct semantics, but it leaks memory when anyone maintains a `let`-binding to an `ivec`. An `ivec` ought to know how to discard past data that will not be needed again.

The design of the `ivec` went through several refinements. These are the successive goals, and the design to meet each:

1. Just get the basic semantics: lazy list.
2. Asymptotically efficient variables: lazy list with `now`.
3. Block computation: list of arrays.
4. Consumer controls block length: `compute` closure.
5. Efficient fan-out: an `ivec` is a reference to an object.

Memoized lazy lists leak memory. Unmemoized ones, instead of spending space to hold the result of forcing a suspension, spend time to recompute it. Neither is ideal. The trouble is that they are holding on to the past—so redefine an `ivec`'s lazy list as holding only the future; its first element is not at time zero, but some other time `now` bound into the `ivec`. This gives the right asymptotic behavior.

Working with individual samples is inefficient; the overhead only becomes tolerable when it is amortized over a block of samples. For block computation, we define an `ivec` as a lazy list of arrays, plus a `now`. In this definition, the producer of the `ivec` controls the block lengths, which causes certain complications: a consumer with multiple `ivec` inputs may receive unaligned block boundaries, or a recursive delay function may receive a block longer than its delay. Section 6.1.2.1 suggests how to deal with this.

Alternatively, give the consumer control over block length. An `ivec` has a `compute` closure which the consumer calls, specifying the block length. The argument to the closure is the time up to which samples should be computed—absolute time, not relative, so fan-out to multiple readers works correctly. Besides output, it returns an `ivec` holding the future.

When a producer fans out to two or more readers, we would prefer it not to recompute for each one. They need to handle it by reference, and it needs to memoize its output, at the least; we choose to give it ownership of an output buffer, which readers share. It also needs to update certain status variables in place, so the second consumer's call can see that the work has already been done. In fact, it does all updates in place, and returns `unit`. We decide to use O'Caml's object features, and make an `ivec` a reference to an object. Inheritance then allows `ivec` to be factored into common and operation-specific parts.

9.1.6 Extensions and future work

Certain extensions to Chronic are really not optional; they would be essential for it to evolve beyond research-quality software. At the top of the list are support for sampling rates, and for vector and `ivec` comprehensions.

In audio synthesis, typically a large fraction of the signals are not audio—they are control signals, having a lower bandwidth than audio, and can be represented with a lower sampling rate. Computer music software typically provides a notion of control-rate signals that can be used when full audio bandwidth is not required. I envision types `'a sig` and `'a isig`, corresponding to `vec` and `ivec`, which incorporate a sampling rate.

Section 5.9 demonstrates that an operation such as `map (+.)` is terribly inefficient; it spends very little of its time performing the floating-point addition itself. Also, the syntax is mildly cumbersome. We might use the O'Caml preprocessor to let us write vector or `ivec` comprehensions, something like `{ (x +. y) *. 0.5: in x, in y }` instead of `IV.map (fun xi yi -> (xi +. yi) *. 0.5) x y`. The preprocessor would generate an object specialized to perform this particular operation on two `ivec` inputs.

9.1.6.1 History?

Having to access `ivecs` monotonically can be unnatural. What about augmenting `ivecs` with history, to allow bounded non-monotonicity? There are several possible designs, but we conclude that fixed-length history is sufficient. As an example, on top of `IV` we build an `IVH` module that defines an `ivec` with history, `ivech`. It contains `IVH.peek`, which we use to write a non-monotonic `IV.peekiv_basis`.

To eliminate `ivec` in favor of `ivech` would penalize the common case, and it is awkward to have two distinct kinds of infinite vectors. We would rather do without history, if we can. In fact, `IV.peekiv_basis` can be written without `IVH`, in terms of `IV.peekiv` and `IV.vdelay`. Increasing a variable delay is like going back in time. Programs involving `IVH.peek` can be rewritten in terms of this `IV.peekiv_basis`.

9.1.6.2 Lazy vectors

Making `'a vec` a synonym for `'a array` means that vectors are strict in all of their elements. Sometimes we would prefer that vectors be lazy. In particular, a dependency on the first element of a strict vector is a dependency on its last, which may be in the future. Therefore, lazy vectors are necessary for real-time operation.

A lazy vector that is accessed monotonically can simply be an `ivec` with an ending time attached. A random-access lazy vector becomes more complicated.

9.1.6.3 Real-time computation

Besides lazy vectors, real-time capability would require a different handling of `event ivecs`, so that commonly-needed code can be written causally. There is also the broader problem that the programmer may write acausal code by accident.

`EIV.vfold` is an example of the `event ivec` problem. It needs to know when it can just proceed with folding the currently-live grains, versus when it has to take a new grain into account. It does this by pulling the next `vec event` off the `vec event ivec`, and examining its timestamp. This is a forward dependency.

One possibility is to fuse `event ivec` into an `event_ivec` type. This type would then support queries of the form "is the next event at a time less than t_1?", which `EIV.vfold` could poll with. However, the type distinction between `event_ivec` and `event ivec` is generally not desirable. We might instead leave the `ivec` alone and alter just the `event`. What `vfold` gets is a "potential event", which it polls with "is your timestamp less than t_1?" until the event happens. One might even use the new kind of event everywhere, replacing the present kind, to keep the type structure simple.

The changes to `event ivecs` and to vectors are necessary because without them the programmer has no way to write causal code. Even with the changes, though, there is the risk of writing acausal code by accident. This is the dark side of the *sub specie æternitatis* model: when you're standing outside of time, a forward dependency is essentially no different than a backward dependency. Ideally, a type system would enforce causality. This seems difficult to do usefully, and remains an open problem.

9.1.7 Related work

The following taxonomy of related work is in terms of its temporal type system first, and then other language characteristics. We consider not how well each work meets its own goals, but how well it meets Chronic's goals. For references, see Chapter 8.

9.1.7.1 Languages with signal and event types

Fran is a domain-specific language, embedded within Haskell, for graphical animation. It actually uses temporal type constructors, `Behavior` and `Event`. An α `Behavior` is a function `Time` $\rightarrow \alpha$, and an α `Event` is a time-sorted sequence of `Time` $\times\,\alpha$ pairs. This continuous time is rather different from Chronic's discrete time: no notion of "previous sample", no efficient delay lines, and no useful way of doing recursive delay in general. On the other hand, it makes motion in animation more accurate. The use of continuous versus discrete time is a consequence, not a cause, of the rather different ways that Fran and Chronic propose to program with temporal data.

ALDiSP is a functional language for DSP and asynchronous event handling. It includes a lazy-stream constructor for synchronous signals, and an input-driven pipe constructor for asynchronous events. This pipe constructor differs from a Chronic `event ivec` in that it represents not the data itself, but suspensions which will react to each event, *sub specie temporis*, in turn.

Everything else has its signals and events as atomic types, not constructed. Some have events *sub specie æternitatis*: in SuperCollider, for example, a score is a lazy sequence of events as symbol tables. Some have them *sub specie temporis*: the Max family processes events in real time as they travel through a static dataflow network. Some have events that are not first-class: the MUSIC-N family places them in a separate score mini-language that has no variables, expressions, or functions.

9.1.7.2 Languages with signals only

Some languages do not have events built in, but do have a reasonably declarative style of programming on some kind of regularly-sampled data: arrays, matrices, or streams. Signals here are often not polymorphic.

Many languages of this class are intended for numerical computation: Sisal and NESL (both parallel), Matlab, SAC, and Fish. SRL is a language specifically for signal processing. Of these, only Sisal has infinite signals, and only NESL has a type system into which Chronic's style could fit.

Another class is the reactive dataflow languages, intended for developing environment-driven systems with a heavy emphasis on safety and formal verifiability, to the point of limiting the programming model to a finite automaton. Chronic is not intended to be verifiable, and hence can have a broader scope.

Libraries for low-level languages may be aimed at DSP in general, or computer music specifically. The Intel Signal Processing Library is example of a DSP library. Its functions operate on single samples and on vectors; for infinite vectors, the programmer calls the vector functions and manages state. STK, a computer music library, has signal-processing and synthesis functions, and also drives them with events from a score or controller. Libraries that handle events *sub specie æternitatis* do not appear to exist.

9.1.7.3 Languages with events only

Practically any language can be used to program with events. One interesting case is Stream, which puts forth the idea of writing real-time interactive programs *sub specie æternitatis*. Haskore, an "algebra of music" embedded in Haskell, defines a score in terms of discrete events from common music notation, plus operations such as transposition and serial or parallel composition.

9.2 Conclusions

9.2.1 Goals

The point of this work was to raise the expressive level of computer music languages without sacrificing generality. If other domains benefit as well, so much the better.

Dannenberg (2001) has suggested an illuminating way to look at the MUSIC-N languages (remember Section 8.1.4). We talk about the score/orchestra dichotomy, the way a MUSIC-N language is partitioned into two sublanguages, but there is a third: the unit generator implementation language. We should not overlook the unit generator language, because the user cannot. Any user whose ideas involve a new DSP operation, or a new kind of analysis or control data, will have to write in this language.

The score/orchestra dichotomy is widely considered harmful, as neither half gains enough from specialization to justify the barrier placed between them. Post-MUSIC-N computer music languages such as Nyquist and SuperCollider have abolished it. Abolished the dichotomy? Made a dichotomy from a trichotomy, I would say. Chronic aims to finish the job.

9.2.2 Ideas

Look at the MUSIC-N languages. Their way to be high-level is to have not the audio sample but the audio signal as a primitive type. They are high-level, to a point—this high and no higher: audio signals, but not streams of cepstral frames. This low and no lower: audio signals, but not audio samples. New kinds of data are too high-level to be expressible, and new DSP is too low.

This is why Chronic has instead the sample as the primitive type, and a constructor that builds the signal from the sample. The type system starts as low as possible, and builds as high as one needs. We propose three temporal type constructors as a basic set: *vector*, *infinite vector*, and *event*.

These constructors mean that even infinite time structures can be represented and viewed from a vantage point outside of time, *sub specie æternitatis*, and operated upon in a purely applicative style. The idea in a nutshell is applicative programming with constructed temporal types. Chronic is an example implementation of this proposed programming style.

9.2.3 Choices

Chronic is embedded in O'Caml, but it could have been an actual variant of the host language, or an entirely new language front-end. These would have had advantages, but the ease of working with the O'Caml embedding allowed the many design iterations that went into this research.

9.2.4 Applications

9.2.4.1 Possible implementations

Chronic's style of programming certainly does not have to be implemented as Chronic is, embedded in O'Caml. Implementation as its own language is another obvious possibility.

However, an embedding in C++ might prove more useful. C++ is widely used, and its template mechanism is (with some syntactic pain) Turing-complete at compile time (Veldhuizen 1995b). This means it could address the problems with feedback delay in Section 9.2.5, which come from a lack of compile-time power. C++ templates also give us vector comprehensions as we wanted in Section 7.1.2 (Veldhuizen 1995a) and preserve features we need, such as lambda expressions (Järvi and Powell 2000).

O'Caml's `camlp4` preprocessor can use O'Caml functions to rewrite the program arbitrarily, so it could certainly do any of these job as well, and probably more elegantly. C++ does have the advantage, though, in widespread use, and in execution speed.

9.2.4.2 Problem domains

What are some examples of algorithms involving non-trivial temporal types, where Chronic can be brought to bear? This was the criterion for the list back in Section 1.1. Now, with an understanding of the uses of temporal types, and having seen the examples from Chapter 4, we can flesh out the possibilities from that list:

- Building computer-music scores graphically and algorithmically from composer-defined score events, control gestures and diagrams, and nested assemblages of these; synthesizing from these scores. We have seen many of the techniques in Section 4.1. Some interface to graphical editors of data would be needed, of course. One might even imagine using this score model in an interface that did not expose code to the user.
- Analysis/synthesis techniques, where audio is analyzed and then resynthesized from the analysis data, which may undergo transformations before resynthesis. We can represent and manipulate the analysis data.
- Feature abstraction from audio signals, such as score transcription from audio. We have the capability to represent both the signal processing and the abstracted features inside the same algorithm.
- Signal processing that is adaptive to abstracted characteristics of the signal: speech effects sensitive to word or phoneme boundaries, drum-loop manipulation based on beat detection. This follows on from the previous item.
- Other signal processing that involves significant data structure: psychoacoustic compression, granular synthesis, filterbanks, wavelet transforms, operations based on linear algebra. Again, we can represent both the structure, which may come from analysis or from synthesis, and the signal processing.
- Speech processing with time warping and multiple levels of abstraction. Speech recognition is a kind of multiple-pass feature extraction process. We can view all of the levels, including some which are realigned in time, *sub specie æternitatis*.
- Applications in other domains. Perhaps simulation, especially when it combines event-driven and continuous time; or image processing, when it combines 2-D signal processing with computational geometry or abstraction of features.

9.2.5 Failures

Chronic has problems with representing delay. First, very short feedback delay is no problem to write, but it forces a short inner loop, and therefore runs slowly. Second, section 4.4 shows that it has no good representation of a multitap feedback delay. The feedback-delay primitives all implement one particular dataflow graph, a single cycle with a delay and a user-definable element, tapped at a particular point. This was enough to do nested allpass filters, but it is not enough in general. In general, we would like to render arbitrary dataflow graphs as code: perhaps by naming nodes, and writing down their relations each in terms of the others, with recursion but no delay-free cycles. This is difficult when Chronic is a library in O'Caml.

The user can in fact write a new `ivec_dat` subclass (Section 5.5.2) that efficiently implements any desired feedback delay structure, but this misses the point. The style of programming involved in subclassing `ivec_dat` is low-level—imperative and heavy on loops. This is what we are trying to get away from.

Secretly, Chronic as implemented does have a kind of unit generator language (it's all in O'Caml—a unit generator *style*). We just assume that our "unit generators" are general enough that the user never thinks of writing more. With feedback delay this fails; we need a more general feedback delay primitive to put it right.

9.2.6 Successes

Chapter 4 shows how Chronic's style expresses certain algorithms almost as directly and clearly as we could wish for. When an algorithm involves non-trivial temporal types, and especially when it thinks about them *sub specie æternitatis*, Chronic's benefits over other programming styles are dramatic.

In other cases, the benefits are less in the expression of the algorithm than in its integration into a high-level language. Even if an algorithm does not itself benefit dramatically, we appreciate its easy high-level communication, through first-class temporal types, with other components.

Chronic does away with separate score, orchestra, and unit generator languages; its one unified language can operate well in all of these realms. Unification means that data at all levels of abstraction can be interconverted freely. A score, interpreted by a performance model, gives control signals; a synthesizer renders audio from the control signals. Or audio is analyzed to get a low-level description, which an inverse performance model turns into a score. Then data at any level may be manipulated, as part of an analysis/resynthesis system, or used to influence processing at a different level.

Unification also gives freedom in code structure and control. When separate languages exist, they are given fixed roles—the score language runs first and the orchestra language operates on its results; the unit generator language implements certain functions and the orchestra language calls them. Then a unit generator never calls into the orchestra, and the orchestra never affects the score. Supporting that kind of interaction is just too hard with separate languages. With one language, of course, it comes for free.

Having the score language be a full-blown programming language allows computation and data structure in the score. The orchestra, likewise, can build the dataflow graph programmatically and rebuild it during computation.

Chronic's programming model allows for some unusual code reuse. The `vfold` operation, for example, can be factored out and reused, where in code *sub specie temporis* the operation is inextricably woven in at each use.

A simplistic implementation of Chronic could be terribly slow, but Chronic manages to be tolerably efficient. This is not trivial, because of the dynamic nature of the computation: techniques applicable to static dataflow do not work here. Chronic develops an `ivec` implementation that permits block computation, in-place update, and sharing output among multiple consumers.

9.2.7 Lessons

Implementing Chronic within O'Caml was expedient, but it is not sufficient for proper treatment of feedback delay. To make this kind of embedded implementation work, we need more to happen at compile time: code generation to create custom inner loops. This might be achievable with O'Caml's `camlp4` preprocessor.

Consumer control of block length may not be worth the trouble. It was meant to let `ivec` functions operate on their own natural block length, but Chronic ends up having to isolate non-standard block lengths inside reblocking wrappers because they can break its dataflow protocol.

Leaving the types' semantics open proved to be a useful hedge. I had not realized that an event's timestamp was not necessarily a sample count, for example, until writing the code for Section 4.1.

A programming style with temporal type constructors is more expressive than existing low-level styles, and more capable than the high-level ones. The expressiveness comes about through the construction of complex time-structured types suited to the problem at hand; the generality is because every type can be deconstructed down to the individual audio sample. Many interesting musical DSP programs, and perhaps other kinds of programs, become easier to write and to read.

Why should types be explicitly temporal? The underlying data bears some relation to time; temporal types move this information from code into data. Time information in the code is implicit in how it handles its time-anonymous data, whereas time information in the data is explicit and available to the compiler. Furthermore, if the data is aware of time, the code can step outside of it, *sub specie æternitatis*.

Appendix A

A very short introduction to DSP

This section serves only as the most abbreviated presentation of some of the ideas and techniques of digital signal processing. DSP is a vast field; for some detail refer for example to Steiglitz' primer (Steiglitz 1996) or Oppenheim and Schafer's classic text (Oppenheim and Schafer 1975).

A.0.8 Signals

Mathematically, a signal is a function over one or more independent variables. We deal with one-dimensional signals: $S : \mathcal{T} \to \mathcal{R}$. (The domain is conventionally referred to as "time", if only for specificity.)

Our signals are digital, discrete in both time and amplitude. That is, $\mathcal{T}$ is $\mathbb{N}$ or $\mathbb{Z}$, and $\mathcal{R}$ is some finite set. CD audio, for example, uses $R = (\mathbb{Z}_{2^{16}} \times \mathbb{Z}_{2^{16}})$, i.e. stereo pairs of 16-bit integers. We will typically think of R as $\mathbb{R}$, but actually use floating point.

A.0.9 Sampling and reconstruction

So a digital signal $x(n)$ is a sequence of reals. How does this relate back to the analog world? An analog signal $a(t)$, with $\mathcal{T} = \mathbb{R}$, can be *sampled* to create a digital signal $x(n)$:

$$x(n) = a(Tn)$$

The sampling interval T is the reciprocal of the sampling frequency F, which for example in CD audio is 44100 Hz (sec^{-1}). The inverse operation, reconstructing intermediate values from the sampled ones, is more involved; it can be done if F is more than twice as high as any frequency in $a(t)$. Time-domain digital audio is a sequence of reals sampled from an analog signal.

A.0.10 The frequency domain

In the analog world, a periodic signal can be expanded to an infinite Fourier series:

$$f(x) = \sum_{n=0}^{\infty} a_n \cos nx + \sum_{n=0}^{\infty} b_n \sin nx$$

A periodic digital signal has a discrete Fourier series which is finite:

$$f(x) = \sum_{n=0}^{N-1} a_n \cos nx + \sum_{n=0}^{N-1} b_n \sin nx$$

One can combine the sine/cosine pairs to get a single sinusoid $c_n \sin(nx + \phi_n)$, described by its amplitude c_n and phase ϕ_n. This sinusoid-basis representation is useful computationally; furthermore, it corresponds to the way we hear.

A.0.11 Filters

A general single-input single-output system is an function taking an $x(n)$ and generating a $y(n)$. A well-developed subset is that of filters, systems that are linear and time-invariant (LTI), which can be described by two sequences $a(n)$ and $b(n)$:

$$y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k)$$

A filter given a sinusoid $x_\omega(n) = c \sin(\omega n + \phi)$ will output one of the same frequency, but difference phase and amplitude: $y_\omega(n) = c' \sin(\omega n + \phi')$. Amplitude gain and phase shift are functions of frequency.

Filters are usually designed in terms of these amplitude and phase responses. A filter to block hum radiated from AC lines, for example, has unity-gain amplitude response except near 60 Hz, where it has a gain ideally of zero.

A.0.12 Implementation

Implementing LTI systems in the time domain is a matter of realizing some dataflow network as code. The network has no delay-free cycles, but otherwise can have a variety of topologies, each involving different delay-feedback paths.

Frequency-domain calculations usually use the fast Fourier transform (FFT), which is far beyond the scope of this section. Suffice it to say that this work involves vector operations, complicated indexing, and out-of-order access to signals.

A.0.13 DSP vs. computer music

Programming for computer music makes heavy use of digital signal processing, but they should not be conflated. Computer music systems are usually not time-invariant, but make use of internal state, modulation, and dynamic rerouting of signals. Nor do they restrict themselves to linear synthesis and processing techniques. Thus the heart of DSP, linear time-invariant systems, applies only to individual elements.

Furthermore, computer music is not just an application of signal processing. Musical events are a prominent exception, and then there are the more elaborate temporal types in section 2.2.

Appendix B

A very short introduction to O'Caml

Readers who have not used O'Caml but do have some experience with strongly-typed functional languages can use this section as a reference for the specifics of O'Caml. Readers whose backgrounds are in Java or Lisp or SuperCollider will, I hope, pick up enough to follow along on the Chronic code examples, but O'Caml's style of programming is different enough that real comfort may come only with writing some exercises. In any case, the ultimate reference is the O'Caml manual, avilable on-line (Leroy 2001).

B.1 Naming values

Values are defined with the `let` expression. The following expression has a value of 5:

```
let x = 3
in x + 2
```

Multiple values with dependencies are defined with nested `let` expressions:

```
let x = 3  in
let y = x + 2
in 5 * y
```

A `let` expression at the global level may omit the `in` clause, making the value available to all following expressions in the global scope:

```
let x = 3
let y = x + 2
```

B.2 Types: basic types and function types

In O'Caml every value has a type. In the code `let x = 3`, the value `x` has the type `int`. The compiler infers this by looking at the right-hand side value `x` is given, namely the integer 3. The programmer may, for documentation or sanity checking, include a type specification. The compiler will confirm it.

```
let (x: int) = 3              (* compiles successfully *)
let (x: string) = 3           (* fails with a type error *)
```

The basic types are `int`, `float`, `bool`, `char`, and `string`. O'Caml's core library has various operators, such as these:

```
let n = 3 + 5
let x = 3. +. 5.        (* float operators are +. *. et cetera *)
let b = (n > 9) || (x > 6.)
let c = char_of_int 65
let s = "hello " ^ "world"
```

Other types are built out of the basic types. For example, the type `int -> bool` will mean a function taking an integer and returning a boolean.

B.3 Functions, named and anonymous

Function values can also be defined with `let` expressions.

```
let is_positive x = x > 0
in is_positive ~-3              (* ~- is unary negation *)
```

The function `is_positive` takes an integer argument and returns an boolean, so its type is `int -> bool`. Function application is done simply by juxtaposition.

While `let` can name a function, the `fun` keyword declares an anonymous function. Here we define an integer-doubling function anonymously, and apply it:

```
  (fun x -> 2 * x) 3
```

Anonymous functions are more useful in practice for passing to other functions.

B.4 Functions, part 2: recursion and multiple arguments

Recursive functions are marked with the `rec` keyword:

```
let rec factorial n =
  if (n = 0) then 1
  else n * (factorial (n-1))
```

The O'Caml `if` *`test`* `then` *`true_val`* `else` *`false_val`* construct is an expression, returning a value, much like *`test`* `?` *`true_val`* `:` *`false_val`* in C.

A function may take multiple arguments:

```
let add x y = x + y
```

The type given for `add` is `int -> int -> int`. This multiple-arrow type may seem bizarre; if you like, you can ordinarily think of it as `int int -> int`. But sometimes it proves useful to go back to the original type, which really means `int -> (int -> int)`. That says that if you pass `add` one integer, it will return an `int -> int` function:

```
let add3 = add 3            (* add3: int -> int *)
let five = add3 2
```

B.5 Types, part 2: tuples and polymorphism

The expression `(3, true)` is called a pair, or 2-element tuple, and has type `int * bool`. Tuples can have any length and any combination of constituent types. Order matters.

Tuples can be taken apart in a `let` binding:

```
let (int_part, bool_part) = (3, true)
   (* or *)
let (int_part, _) = (3, true)      (* _ is "don't care" *)
```

Tuples let us see something new that functions can do:

```
let pair_with_true x = (x, true)
```

The function `pair_with_true` has type `'a -> ('a * bool)`. The `'a` is a type variable, which can take on any type. This represents the fact that `pair_with_true` is polymorphic. If you pass it an `int`, it returns an `int * bool`. If you pass it a `string`, it returns a `string * bool`.

Note that O'Caml uses the word "polymorphic" differently than C++ does. C++ 'polymorphism' is over related object types (behind pointers). O'Caml polymorphism is over any type whatsoever.

B.6 Types, part 3: declared types, arrays, and records

Not only values can be declared; types can be declared too.

```
type int_and_bool = int * bool
type 'a predicate = 'a -> bool
```

The first declaration is a simple shorthand synonym. The second shows how a type declaration can include type variables.

Types can have multiple alternatives. One way to operate on these types is to break down the alternatives with `match`.

```
type number = Rational of int*int | Float of float
let pi = Rational (355, 113)
let pi = Float 3.14159

let number_to_float x =
  match x with
    (Float f)         -> f
  | (Rational (p, q)) -> (float_of_int p) /. (float_of_int q)
```

Arrays are finite sequences indexed by non-negative numbers. An `'a array` can contain only `'a` elements (but `'a` may include alternatives). For example, an `int array`:

```
let squares = [| 0; 1; 4; 9; 16 |]
let squares_sub_2 = squares.(2)
```

The O'Caml standard library's `Array` module contains functions for creating and working with arrays. For example, `squares` can be defined with `Array.init`, which evaluates a given function at non-negative integers:

```
let squares = Array.init 5 (fun i -> i*i)
```

Records are like tuples with named fields.

```
type person =  name: string; age: int
let bob =  name = "Bob"; age = 42
let bob_age = bob.age
```

B.7 Imperative features

Arrays are mutable; the values they contain can be changed.

```
let squares = Array.init 5 (fun i -> i*i)
let _ = squares.(2) <- 17
let squares_sub_2 = squares.(2)
```

The array-mutation operation `squares.(2) <- 17` is, like everything else in O'Caml, an expression with a value. That value is of type `unit`, a type which has only one value, called `()`. This is what expressions evaluate to when they don't care what they evaluate to. As you can see, we ignore it.

Another way to ignore a value is to use the sequencing construct. This is something like C's comma operator:

```
let squares_sub_2 = (
  squares.(2) <- 17;
  squares.(2)
)
```

O'Caml even has `for` and `while` loops:

```
let squares = Array.init 5 (fun i -> i)
let _ =
  for i = 0 to 4 do
    squares.(i) <- squares.(i) * squares.(i)
  done
```

Records can also have mutable elements:

```
type person =  name: string; age: mutable int
let age p =
   p.age <- p.age + 1
```

Finally, an `'a ref` is a cell containing a mutable `'a`. The dereference operator is `!`, and update is `:=`.

```
let x = 3
let xref = ref x
let _ = (
  Printf.printf "%d\n" (!xref);    (* 3 *)
  xref := 8;
  Printf.printf "%d\n" (!xref);    (* 8 *)
  Printf.printf "%d\n" x           (* 3 *)
)
```

B.8 Objects

Class definitions use the `val` keyword for member data, and the `method` keyword for member functions. (A member function is unusual in that one can declared without any arguments, not even a `()`. The `initializer` runs at object creation time.

```
class mammal init_fur_length = object (self)
  val mutable fur_length = init_fur_length

  initializer
    if (init_fur_length > 10)
    then fur_length <- 10

  method get_fur_length = fur_length
  method set_fur_length new_fur_length =
    fur_length <- new_fur_length
end
```

The `new` keyword instantiates an object of this class; here we give 2 for `init_fur_length`. An object's methods are accessed using the `#` separator, like this:

```
let platypus = new mammal 2
let len = platypus#get_fur_length
```

The `inherit` keyword defines inheritance relationships. The `:>` operator upcasts an object of a child class to an object of an ancestor class.

```
class cow init_fur_length (num_of_spots: int) = object
  inherit mammal init_fur_length

  method get_num_of_spots = num_of_spots
end

let bossie = new cow 10 0              (* spotless yak *)
let bossie_as_mammal = (bossie :> mammal)
```

B.9 Summary of types

expression	type
`()`	`unit`
`2 + 3`	`int`
`2.5 *. 2.`	`float`
`3 = 5`	`bool`
`"hello"`	`string`
`fun x -> 3 + (int_of_float x)`	`float -> int`
`fun x y -> x + (int_of_float y)`	`int -> float -> int`
`(7, true)`	`int * bool`
`[ 7; 42 ]`	`int list`
`[| 7; 42 |]`	`int vec`

Appendix C

Chronic reference

Chapter 3 explained Chronic's overall organization, and its division into core and library modules:

module	type
`E`	`event`
`V`	`vec`
`M`	`vec vec` (matrices)
`IV`	`ivec`
`EV`	`event vec`
`EIV`	`event ivec`
`L`	mostly scalars
`LV`	`vec`
`LIV`	`ivec`

This chapter gives a full listing of the functions within each module.

C.1 The `event` module

To use the `E` module and other `event` operations, issue an `open Event` command. This gives access to the following functions:

C.1.1 Functions within the `E` module

Create and dismantle events

`split` Break an event into its timestamp and value.
`(ev_time: time, ev_value: 'a) = split (ev: 'a event)`

`v` Extract an event's value.
`(ev_value: 'a) = v (ev: 'a event)`

`t` Extract an event's time.
`(ev_time: time) = t (ev: 'a event)`

`diff` Calculate the time difference between two events.
`(difference: time) = diff (first: 'a event) (second: 'b event)`

`lift` Lift f to operate on events, leaving timestamps unchanged.
`(f': 'a event -> 'b event) = lift (f: 'a -> 'b)`

Furthermore, some event operators are offered outside of `E`, at the top level:

C.1.2 Top-level operators

Create and dismantle events

`@@` From a value and a time, form an event.
`(v_at_time_t: 'a event) = (v: 'a) @@ (t: time)`

`?@@` Extract an event's time.
`(ev_time: time) = ?@@ (ev: 'a event)`

`?@:` Extract an event's value.
`(ev_value: 'a) = ?@: (ev: 'a event)`

Adjust event times

`@+` Copy an event, incrementing its time.
`(e': 'a event) = (e: 'a event) @+ (t_increment: time)`

`@-` Copy an event, decrementing its time.
`(e': 'a event) = (e: 'a event) @- (t_decrement: time)`

`@!` Copy an event, attaching a new time.
`(e': 'a event) = (e: 'a event) @! (t': time)`

C.2 The **vec** module

Recall that vector elements can be addressed through O'Caml array syntax, with zero-based indexing:

```
let x = V.tabulate (fun x -> x*x) 4     (* x is [| 0; 1; 4; 9 |] *)
let _ = (x.(2) <- 17)                    (* becomes [| 0; 1; 17; 9 |] *)
let x2 = x.(2)                           (* i.e. x.(2) is now 17 *)
```

To access the `V` module, issue an `open Vec` command. It contains the following functions:

Create vectors

`const` Create a vector of length n whose elements are all x_i.
`(x: 'a vec) = const (xi: 'a) (n: time)`

`copy` Copy an existing vector.
If `'a` is a reference type, `copy` will alias.
`(x2: 'a vec) = copy (x: 'a vec)`

`tabulate` Apply `f` to the integers $[0, \dots, n-1]$.

	`(x: 'a vec) = tabulate (f: time -> 'a) (n: time)`
`iterate`	Return $[x_0, f(x_0), f\big(f(x_0)\big), \ldots, f^{(n-1)}(x_0)]$.
	`(x: 'a vec) = iterate (f: 'a -> 'a) (x0: 'a) (n: time)`

Examine vectors

`length`	Give the length of a vector.
	`(n: time) = length (x: 'a vec)`
`isempty`	Tell whether a vector is empty.
	`(emptyP: bool) = isempty (x: 'a vec)`

Convert to and from vectors

`from_scalar`	Make a one-element vector.
	`(x: 'a vec) = from_scalar (x0: 'a)`
`from_list`	Make a vector from a list.
	`(x: 'a vec) = from_list (xlist: 'a list)`
`to_list`	Make a list from a vector.
	`(xlist: 'a list) = from_list (x: 'a vec)`

Output

`print_ints`	Print a vector of integers.
	`() = print_ints (x: int vec)`
	Side effect: writes to `stdout`.
`print_floats`	Print a vector of floats.
	`() = print_floats (x: float vec)`
	Side effect: writes to `stdout`.

Input

`scanf_floats`	Read floating-point values as text from `stdin`.
	`(x: float vec) = scanf_floats ()`
`scanf_floats_from`	Read floating-point values as text from file *filename*.
	`(x: float vec) = scanf_floats_from (filename: string)`

Take vectors apart

`subvec`	Copy *len* elements, starting at *n*, from *x*.
	`(y: 'a vec) = subvec (n: int) (len: int) (x: 'a vec)`
`blit`	Write *len* elements from *source* to *dest*, with offsets *isource* and *idest*.
	`() = blit (source: 'a vec) (isource: int)`

(dest: 'a vec) (idest: int) (len: int)
Side effect: writes to *dest*.

`take` Copy the first n elements of x.
`(y: 'a vec) = take (n: int) (x: 'a vec)`

`drop` Copy all but the first n elements of x.
`(y: 'a vec) = drop (n: int) (x: 'a vec)`

`take_end` Copy the last n elements of x.
`(y: 'a vec) = take_end (n: int) (x: 'a vec)`

`drop_end` Copy all but the last n elements of x.
`(y: 'a vec) = drop_end (n: int) (x: 'a vec)`

`split` Split x into a head (of length n) and a tail.
`(xhead: 'a vec, xtail: 'a vec) = split (n: int) (x: 'a vec)`

Join vectors together

`join` Join two vectors.
`(xy: 'a vec) = join (x: 'a vec) (y: 'a vec)`

`ravel2` Flatten a (ragged) matrix, joining its rows end-to-end.
`(flat: 'a vec) = ravel2 (mat: 'a vec vec)`

Index into vectors

`peek` Read x_i: equivalent to `x.(i)`.
`(xi: 'a) = peek (i: int) (x: 'a vec)`

`poke` Write to x_i: equivalent to `x.(i) <- e`.
`() = poke (i: int) (e: 'a) (x: 'a vec)`
Side effect: writes to x.

`poke_copy` As `poke`, but applied to a copy of the target.
`(x': 'a vec) = poke_copy (i: int) (e: 'a) (x: 'a vec)`

`peekv` Read indexed by a vector: $[x_{i_0}, x_{i_1}, \ldots]$.
`(y: 'a vec) = peekv (iv: int vec) (x: 'a vec)`

`pokev` Write indexed by a vector: $x_{i_0} \leftarrow e_0, x_{i_1} \leftarrow e_1, \ldots$.
`() = pokev (iv: int vec) (ev: 'a vec) (x: 'a vec)`
Side effect: writes to x.

`pokev_copy` As `pokev`, but applied to a copy of the target.
`(x': 'a vec) = pokev_copy (iv: int vec) (ev: 'a vec) (x: 'a vec)`

`fill` Fill from *start* up to (but not including) *upto*.
`() = fill (start: int) (upto: int) (e: 'a) (x: 'a vec)`
Side effect: writes to x.

`first` Read the first element.
`(x0: 'a) = first (x: 'a vec)`

`last` Read the last element.
`(xlast: 'a) = last (x: 'a vec)`

Map over vectors

`apply` Call f on each element of x.
`() = apply (f: 'a -> unit) (x: 'a vec)`
Side effect: calls f for its side effect.

`applyi` Call f on each element's index and value.
`() = applyi (f: int 'a -> unit) (x: 'a vec)`
Side effect: calls f for its side effect.

`map` Map f over each element of x.
`(y: 'b vec) = map (f: 'a -> 'b) (x: 'a vec)`

`mapi` Map f over each element's index and value.
`(y: 'b vec) = mapi (f: int 'a -> 'b) (x: 'a vec)`

`map2` Map f over corresponding elements of x and y.
`(z: 'c vec) = map2 (f: 'a 'b -> 'c) (x: 'a vec) (y: 'b vec)`

`map3` Map f over corresponding elements of x, y and z.
```
(w: 'd vec) = map3 (f: 'a 'b 'c -> 'd)
                   (x: 'a vec) (y: 'b vec) (z: 'c vec)
```

`map_slide2` Return $[f(x_{\mathrm{pre}}, x_0), f(x_0, x_1), \ldots, f(x_{|x|-2}, x_{|x|-1})]$.
`(y: 'b vec) = map_slide2 (f: 'a 'a -> 'b) (xpre: 'a) (x: 'a vec)`

Scan along vectors

`scan` Return $y = [y_0, f(y_0, x_0), f(y_1, x_1), \ldots, f(y_{|x|-2}, x_{|x|-2})]$, where $|x| = \mathsf{length}\, x$.
`(y: 'a vec) = scan (f: 'a 'b -> 'a) (y0: 'a) (x: 'b vec)`

`scanx` "Scan extended": return $y = [y_0, f(y_0, x_0), f(y_1, x_1), \ldots, f(y_{|x|-1}, x_{|x|-1})]$.
`(y: 'a vec) = scanx (f: 'a 'b -> 'a) (y0: 'a) (x: 'b vec)`

`scan'` Return $y = [f(y_{\mathrm{pre}}, x_0), f(y_0, x_1), \ldots, f(y_{|x|-2}, x_{|x|-1})]$.
`(y: 'a vec) = scan (f: 'a 'b -> 'a) (ypre: 'a) (x: 'b vec)`

`sidescan` Return y, where $(a_0, y_0) = f(a_{\mathrm{pre}}, x_0)$; $(a_1, y_1) = f(a_0, x_1)$;
$\ldots\ (a_{|x|-1}, y_{|x|-1}) = f(a_{|x|-2}, x_{|x|-1})$.
```
(y: 'c vec) = sidescan (f: 'a 'b -> ('a, 'c))
                       (apre: 'a) (x: 'b vec)
```

Fold vectors

`foldl` Fold x leftwards.
`(y: 'b) = foldl (f: 'b 'a -> 'b) (y0: 'b) (x: 'a vec)`

`foldr` Fold x rightwards.
`(y: 'b) = foldl (f: 'a 'b -> 'b) (x: 'a vec) (y0: 'b)`

`foldl0` Fold x leftwards, without using an identity element *y0*.
`foldl0 f x` is equivalent to `foldl f x.(0) (V.drop 1 x)`.
`(y: 'a) = foldl0 (f: 'a 'a -> 'a) (x: 'a vec)`

Functions whose output length is unpredictable

`filter` Selects elements of x that satisfy f.
`(y: 'a vec) = filter (f: 'a -> bool) (x: 'a vec)`

`compress` Selects elements of x according to a corresponding boolean vector.
`(y: 'a vec) = compress (mask: bool vec) (x: 'a vec)`

`find` Returns indices of elements of x that satisfy f.
`(y: int vec) = find (f: 'a -> bool) (x: 'a vec)`

Zip and unzip

`zip` From two vectors, makes a vector of pairs.
`(ab: ('a, 'b) vec) = zip (a: 'a vec) (b: 'b vec)`

`zip3` From three vectors, makes a vector of triples.
`(abc: ('a, 'b, 'c) vec) = zip3 (a: 'a vec) (b: 'b vec) (c: 'c vec)`

`unzip` From a vector of pairs, makes a pair of vectors.
`(a: 'a vec, b: 'b vec) = unzip (ab: ('a, 'b) vec)`

`unzip3` From a vector of triples, makes a triple of vectors.
`(a: 'a vec, b: 'b vec, c: 'c vec) = unzip3 (abc: ('a, 'b, 'c) vec)`

Specialized functions

`scale` The same as `map (fun xi -> k *. xi) x`
`(kx: float vec) = scale (k: float) (x: float vec)`

`iterate_mul` The same as `iterate (fun xi -> k *. xi) x0 n`
`(x: float vec) = iterate_mul (k: float) (x0: float) (n: int)`

`mul3k` The same as `map3 (fun xi yi zi -> k *. xi *. yi *. zi) x y z`
```
(kxyz: float vec) = mul3k (k: float) (x: float vec)
                          (y: float vec) (z: float vec)
```

C.3 The `matrix` module

At present only finite-matrix operations are supported. These functions are found in `M.V`, the vector submodule of the matrix module. To access the `M.V` module, issue an `open Matrix` command. It contains the following functions:

Create

`tabulate` Evaluates $f\ i\ j$ on $0 \le i < m$ and $0 \le j < n$.
`(a: 'a vec vec) = tabulate (f: int int -> 'a) (m: int, n: int)`

Examine

`rows` The number of rows in a.
`(m: int) = rows (a: 'a vec vec)`

`cols` The number of columns in a.
`(n: int) = cols (a: 'a vec vec)`
Assumes a rectangular matrix.

`dims` Dimensions of a, rows and columns.
`(m: int, n: int) = dims (a: 'a vec vec)`
Assumes a rectangular matrix.

Extract vectors

Indexing is zero-based, not the traditional one-based matrix indexing.

`diag` Extract a diagonal of a.
An *ofs* of 0 gives the main diagonal, $\{a_{00}, a_{11}, \ldots\}$;
an *ofs* of 1 gives an upper diagonal, $\{a_{01}, a_{12}, \ldots\}$;
an *ofs* of -1 gives a lower diagonal, $\{a_{10}, a_{21}, \ldots\}$.
`(x: 'a vec) = diag (ofs: int) (a: 'a vec vec)`
Assumes a rectangular matrix.

`crossdiag` Extract a cross-diagonal of a.
An *ofs* of 0 gives $\{a_{00}\}$; an *ofs* of 1 gives $\{a_{01}, a_{10}\}$; etc.
Note that *ofs* is non-negative here, as opposed to in `diag`.
`(x: 'a vec) = crossdiag (ofs: int) (a: 'a vec vec)`
Assumes a rectangular matrix.

Output

Matrices are written one whitespace-delimited line per row, starting from row 0.

`print_ints` Print a to `stdout`.
`() = print_ints (a: int vec vec).`

`print_floats` Print a to `stdout`.
`() = print_floats (a: float vec vec).`

Input

Matrices are read from the same format used for output.

`scanf_floats_from` Read floats from *filename* until EOF.
`(a: float vec vec) = scanf_floats_from (filename: string)`

`scanf_floats` Read floats from `stdin` until EOF.
`(a: float vec vec) = scanf_floats (filename: string)`

C.4 The `ivec` module

An `'a ivec` is a infinite vector whose elements are of type `'a`. Indexing is by zero-based time.

Some `ivec` functions require an *exemplar* argument, an example of one output element. For example, `map trunc` converts a `float ivec` to an `int ivec`, so it needs an `int` exemplar—any `int`; 42 is as good as any other. Section 5.5.3 gives the technical reason for this awkwardness.

The following functions are inside the `IV` module:

Create ivecs

`const`	Create an `ivec` whose elements are all x_i. `(x: 'a ivec) = const (xi: 'a)`
`tabulate`	Apply `f` to the integers $[0, 1, \ldots]$. `(x: 'a ivec) = tabulate (f: time -> 'a)` Assumes that f has no side-effects.
`iterate`	Return $[x_0, f(x_0), f\big(f(x_0)\big), \ldots]$. `(x: 'a ivec) = iterate (f: 'a -> 'a) (x0: 'a)`

`iterate_niladic`	Call `f ()` repeatedly, and return the results.
	Takes an *exemplar* argument as described above.
	`(x: 'a ivec) = iterate_niladic (exemplar: 'a)`
	`(f: unit -> 'a)`
	Side effect: calls f for its side effect.

Take ivecs apart; put ivecs together.

`take`	Copy the first n elements of x. `(y: 'a vec) = take (n: int) (x: 'a ivec)`
`drop`	Copy all but the first n elements of x. `(y: 'a ivec) = drop (n: int) (x: 'a ivec)`
`split`	Split x into a head (of length n) and a tail. `(xhead: 'a vec, xtail, 'a ivec) = split (n: int) (x: 'a ivec)`
`join`	Join a vector and an `ivec`. `(xy: 'a ivec) = join (x: 'a vec) (y: 'a ivec)`
`spy`	Examine the first n elements of x, returning a usable copy x' of x (which is consumed). `(xhead: 'a vec, x': 'a ivec) = spy (n: int) (x: 'a ivec)`
`break`	Break x into pieces. $length_i$ is the length of y_i. `(y: 'a vec ivec) = break (lengths: time ivec) (x: 'a ivec)` *Not implemented.*
`ravel`	Join elements of x. `(y: 'a ivec) = ravel (x: 'a vec ivec)`

Map over ivecs

`map`	Map f over each element of x. Takes an *exemplar* argument as described above. `(y: 'b ivec) = map (exemplar: 'b) (f: 'a -> 'b) (x: 'a ivec)` Assumes that f has no side-effects.
`mapi`	Map f over each element's index and value. Takes an *exemplar* argument as described above. `(y: 'b ivec) = map (exemplar: 'b) (f: int 'a -> 'b) (x: 'a ivec)` Assumes that f has no side-effects.

`map2`	Map f over corresponding elements of x and y.
	`(z: 'c ivec) = map2 (exemplar: 'c) (f: 'a 'b -> 'c)`
	`(x: 'a ivec) (y: 'b ivec)`
	Assumes that f has no side-effects.

`add2_float`	Add corresponding elements of x and y. `(z: float ivec) = add2_float (x: float ivec) (y: float ivec)`

`map4i`	Map f over corresponding elements of x, y and z.
	Takes an *exemplar* argument as described above.
	`(e: 'e ivec) = map4i (exemplar: 'e) (f: int 'a 'b 'c 'd -> 'e)`
	`(a: 'a ivec) (b: 'b ivec)`
	`(c: 'c ivec) (d: 'd ivec)`
	Assumes that f has no side-effects.

`map_slide2`	Return $[f(x_{\mathrm{pre}}, x_0), f(x_0, x_1), \ldots]$.
	Takes an *exemplar* argument as described above.
	`(y: 'b ivec) = map_slide2 (exemplar: 'b) (f: 'a 'a -> 'b)`
	`(xpre: 'a) (x: 'a ivec)`
	Assumes that f has no side-effects.

Scan along ivecs

`scan`	Return $y = [y_0, f(y_0, x_0), f(y_1, x_1), \ldots]$. `(y: 'a ivec) = scan (f: 'a 'b -> 'a) (y0: 'a) (x: 'b ivec)`
`scan'`	Return $y = [f(y_0, x_0), f(y_1, x_1), \ldots]$. `(y: 'a ivec) = scan' (f: 'a 'b -> 'a) (y0: 'a) (x: 'b ivec)`

`sidescan`	Return y, where $(a_0, y_0) = f(a_{\mathrm{pre}}, x_0)$; $(a_1, y_1) = f(a_0, x_1)$; $\ldots$.
	`(y: 'c ivec) = sidescan (f: 'a 'b -> ('a, 'c))`
	`(apre: 'a) (x: 'b ivec)`

Index into ivecs

`peekiv`	Read indexed by an `ivec`: $[x_{i_0}, x_{i_1}, \ldots]$. Takes an *exemplar* argument as described above. `(y: 'a ivec) = peekiv (exemplar: 'a) (i: int ivec) (x: 'a ivec)`
`peekivr`	Read indexed by an `ivec` of reals: $[x_{\lfloor t_0 \rfloor}, x_{\lfloor t_1 \rfloor}, \ldots]$.

Takes an *exemplar* argument as described above.
`(y: 'a ivec) = peekivr (exemplar: 'a) (t: float ivec) (x: 'a ivec)`

Delay ivecs

Delay operations raise the question of what came before time zero. For example, if you delay x by `IV.const 3`, what are the first three samples of the result? Here the *basis*, which answers that question, must be three or more samples long.

The feedback function supplied to one of the recursive-delay operators must never refer to `ivecs` other than its arguments. For example, `fun x -> IV.map2 (+) x foo` may not be passed to `delay_rec`—instead, pass `fun x foo -> IV.map2 (+) x foo` to `delay_rec2`. This error is not detected, and produces undefined results. Section 5.7 explains the problem.

`vdelay` — Delay x by a time-varying amount d.
```
(y: 'a ivec) = vdelay (d: time ivec) (basis: 'a vec)
                      (x: 'a ivec)
```

`vdelay_rec` — The same as `y = vdelay d basis (f y)`, if that were legal.
```
(y: 'a ivec) = vdelay_rec (d: time ivec) (basis: 'a vec)
                          (f: 'a ivec -> 'a ivec)
```
The function f may not refer to `ivecs` other than its argument.

`vdelay_rec2` — The same as `y = vdelay d basis (f x y)`, if that were legal.
```
(y: 'a ivec) = vdelay_rec2 (d: time ivec) (basis: 'a vec)
                           (f: 'b ivec -> 'a ivec -> 'a ivec)
                           (x: 'b ivec)
```
The function f may not refer to `ivecs` other than its arguments.

`delay` — Delay x by a fixed amount d.
```
(y: 'a ivec) = delay (d: time) (basis: 'a vec)
                     (x: 'a ivec)
```

`delay_rec` — The same as `y = delay d basis (f y)`, if that were legal.
```
(y: 'a ivec) = delay_rec (d: time) (basis: 'a vec)
                         (f: 'a ivec -> 'a ivec)
```
The function f may not refer to `ivecs` other than its argument.

`delay_rec2` — The same as `y = delay d basis (f x y)`, if that were legal.
```
(y: 'a ivec) = delay_rec2 (d: time) (basis: 'a vec)
                          (f: 'b ivec 'a ivec -> 'a ivec)
                          (x: 'b ivec)
```
The function f may not refer to `ivecs` other than its arguments.

`vdelayz` — Like `vdelay` with a *basis* of all `0.0`.
Requires $d \leq d_{max}$.
```
(y: float ivec) = vdelayz (d: time ivec) (dmax: time)
                          (x: float vec)
```

`vdelayz_rec` — Like `vdelay_rec` with a *basis* of all `0.0`.
Requires $d \leq d_{max}$.
```
(y: float ivec) = vdelay_rec (d: time ivec) (dmax: time)
                             (f: float ivec -> float ivec)
```

	The function f may not refer to `ivec`s other than its argument.
`delayz`	Like `delay` with a *basis* of all `0.0`. `(y: float ivec) = delay (d: time) (x: float vec)`

`delayz_rec`	Like `delay_rec` with a *basis* of all `0.0`.
	`(y: float ivec) = delay_rec (d: time)`
	`(f: float ivec -> float ivec)`
	The function f may not refer to `ivec`s other than its argument.

`delayz_rec2`	Like `delay_rec2` with a *basis* of all `0.0`.
	`(y: float ivec) = delayz_rec (d: time) (f: float ivec ->`
	`float ivec -> float ivec)`
	`(x: float ivec)`
	The function f may not refer to `ivec`s other than its arguments.

Digital filters

`filter1`	Applies first-order IIR filter (in direct-II form) to x.
	Satisfies $y_n = b_0 x_n + b_1 x_{n-1} - a_1 y_{n-1}$. (Note sign of a_1.)
	Transfer function is $H(z) = (b_0 + b_1 z^{-1})/(1 + a_1 z^{-1})$.
	`(y: float ivec) = filter1 (b0: float, b1: float) (a1: float)`
	`(x: float ivec)`

`filter2`	Applies second-order IIR filter (in direct-II form) to x.
	Satisfies $y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} - a_1 y_{n-1} - a_2 y_{n-2}$. (Note sign of a_i.)
	Transfer function is $H(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2})/(1 + a_1 z^{-1} + a_2 z^{-2})$.
	`(y: float ivec) = filter2 (b0: float, b1: float, b2: float)`
	`(a1: float, a2: float)`
	`(x: float ivec)`

Output

Functions which operate on the beginning of an `ivec` have the side effect of consuming the `ivec`. It cannot be referenced again afterwards. In the terminology of Section 5, these are *compute-phase* functions.

`apply_upto`	Apply f to the initial elements of x.
	`() = apply_upto (upto: int) (blocklen: int)`
	`(f: 'a -> ()) (x: 'a ivec)`
	Side effect: Consumes x; calls f for its side effect.

`print_ints`	Print the beginning of an `int ivec` to `stdout` as text. `() = print_int (upto: int) (blocklen: int) (x: int ivec)` *Side effect*: Consumes x; writes to `stdout`.
`print_floats`	Print the beginning of a `float ivec` to `stdout` as text. `() = print_float (upto: int) (blocklen: int) (x: float ivec)` *Side effect*: Consumes x; writes to `stdout`.
`write_floats`	Write the beginning of a `float ivec` to `stdout` as binary.

	`() = print_float (upto: int) (blocklen: int) (x: float ivec)` *Side effect*: Consumes x; writes to `stdout`.
`ignore`	Compute and throw away the beginning of an `'a ivec`. `() = print_float (upto: int) (blocklen: int) (x: 'a ivec)` *Side effect*: Consumes x.

Input

`scanf_floats`	Read floating-point values as text from `stdin`. After end-of-file, gives zeroes. `(x: float ivec) = scanf_floats ()`
`scanf_floats_from`	Read floating-point values as text from file *filename*. After end-of-file, gives zeroes. `(x: float ivec) = scanf_floats_from (filename: string)`

Specialized functions

`zero`	The same as `const 0.` `(zero: float ivec)`
`count_from`	The same as `iterate (fun x -> x +. delta) from` `(x: float ivec) = count_from (from: float) (delta: float)`
`count`	The same as `count 0. delta` `(x: float ivec) = count (delta: float)`
`icount_from`	The same as `iterate (fun x -> x + delta) from` `(x: int ivec) = icount_from (from: int) (delta: int)`
`icount`	The same as `icount 0 delta` `(x: int ivec) = icount (delta: int)`

C.5 The `event ivec` module

Recall that the `event ivec` functions require that the event timestamps be non-decreasing. This is checked only at run time. To access the `EIV` module, issue an `open evivec` command. The module contains the following functions:

`pwc`	Convert a series of breakpoint values to a piecewise-constant `ivec`. E.g. `pwc 3 [| 1 @@ 2; 2 @@ 4; ...|]` = `[| 3; 3; 1; 1; 2; ...|]`. `(xpwc: 'a ivec) = pwc (x0: 'a) (xevs: 'a event ivec)`
`pwl`	Convert a series of breakpoint values to a piecewise-linear `ivec`. E.g. `pwl 3. [| 1. @@ 2; 2. @@ 4; ...|]` = `[| 3.; 2.; 1.; 1.5; 2.; ...|]`. `(xpwl: float ivec) = pwl (x0: float) (xevs: float event ivec)`
`chain`	Call f to generate a new `ivec` for each event, and chain these `ivecs` together, carrying the current x value through each call to f.

E.g. `chain (fun x e -> IV.iterate (fun x -> x+(E.v e)) x)`
`1 [| 2 @@ 0; 10 @@ 3 |]`
= `[| 1; 3; 5; 7; 17; 27; ... |]`
`(x: 'a ivec) = chain (f: 'a -> 'b event -> 'a ivec)`
`(x0: 'a) (evs: 'b event ivec)`
Not implemented.

`vfold` Vertically fold overlapping grains in *evs*.
Use f as the folding function and *id* as its identity.
The fold is in `foldl` order if the values are old-to-new.
E.g. `vfold (+) [| [|1; 1; 1|] @@ 2; [|2; 2; 2|] @@ 3; ...|]` =
`[| 0; 0; 1; 3; 3; 2; ...|]`.
`(folded: 'b ivec) = vfold (f: 'b -> 'a -> 'b) (id: 'b)`
`(evs: 'a vec event ivec)`

`vmix` Vertically add overlapping grains in *evs*.
Equivalent to `vfold (+.) 0.`
`(mixed: float ivec) = vmix (evs: float vec event ivec)`

C.6 The event vec module

Recall that the `event vec` functions require that the event timestamps be non-decreasing. This is checked only at run time. To access the `EV` module, issue an `open evvec` command. The module contains the following functions:

`pwc` Convert a series of breakpoint values to a piecewise-constant `vec`.
Treats the data as open on the right end — see `close`.
E.g. `pwc 3 [| 1 @@ 2; 2 @@ 4 |]` = `[| 3; 3; 1; 1 |]`.
`(xpwc: 'a vec) = pwc (x0: 'a) (xevs: 'a event vec)`

`pwl` Convert a series of breakpoint values to a piecewise-linear `vec`.
Treats the data as open on the right end, which probably makes the output shorter than you want — see `close`.
E.g. `pwl 3. [| 1. @@ 2; 2. @@ 4 |]` =
`[| 3.; 2.; 1. |]`.
`(xpwl: float vec) = pwl (x0: float) (xevs: float event vec)`

`close` Append a closing event.
E.g. `close [| 1 @@ 2; 2 @@ 4 |]` = `[| 1 @@ 2; 2 @@ 4; 2 @@ 5 |]`.
Now, `pwc 3 (close [| 1 @@ 2; 2 @@ 4 |])` = `[| 3; 3; 1; 1; 2 |]`,
and `pwl 3. (close [| 1. @@ 2; 2. @@ 4 |])` =
`[| 3.; 2.; 1.; 1.5; 2. |]`.
`(evs_closed: 'a event vec) = close (evs: 'a event vec)`

`pwc_list` Convert a list of pairs to a vector of events, `close` it, and apply `pwc`.
E.g. `pwc_list 3 [(2, 1); (4, 2)]` = `[| 3; 3; 1; 1; 2 |]`.
`(xpwc: 'a vec) = pwc_list (x0: 'a) (xevlist: (time * 'a) vec)`

`pwl_list` Convert a list of pairs to a vector of events, `close` it, and apply `pwl`.
E.g. `pwc_list 3. [(2, 1.); (4, 2.)]` = `[| 3.; 2.; 1.; 1.5; 2. |]`.

```
(xpwl: 'a vec) = pwl_list (x0: 'a) (xevlist: (time * 'a) vec)
```

C.7 The standard library

To access the `L`, `LV`, and `LIV` modules, issue an `open chronlib` command. The library contains the following functions:

C.7.1 General library

C.7.1.1 Math

`pi` The value of π.
`(pi: float)`

`subunity` A value very slightly less than 1.0.
`(subunity: float)`

`sinc` The sinc function, $\big(\sin(\pi x)\big)/(\pi x)$.
`(sinc_x: float) = sinc (x: float)`

C.7.1.2 Conversions between units

`amp_to_db` Convert a multiplicative factor to a dB offset.
`(db: float) = amp_to_db (amp: float)`

`db_to_amp` Convert a dB offset to a multiplicative factor.
`(amp: float) = db_to_amp (db: float)`

`oct_to_hz` Convert an octave value to a frequency.
Assumes A5 is 440 Hz.
`(hz: float) = oct_to_hz (octave: float)`

C.7.1.3 Window functions

`f_raised_cosine` Evaluate the raised-cosine window function over the domain $[-1, 1]$.
`(y: float) = f_raised_cosine (x: float)`

`f_hamming` Evaluate the Hamming window function over the domain $[-1, 1]$.
`(y: float) = f_hamming (x: float)`

`f_blackman3` Evaluate the three-term Blackman-Harris -67dB window function over the domain $[-1, 1]$.
`(y: float) = f_blackman3 (x: float)`

`f_blackman4` Evaluate the four-term Blackman-Harris -92dB window function over the domain $[-1, 1]$.
`(y: float) = f_blackman4 (x: float)`

C.7.1.4 Interpolation

`interp1` Use linear interpolation to get a value for x at index t.
`(x_t: float) = interp1 (t: float) (x: float vec)`

C.7.1.5 Filter coefficients

`lowpass2` Generate coefficients for a second-order lowpass filter with normalized cutoff fc and resonance q.
`((b0: float, b1: float, b2: float), (a1: float, a2: float)) = lowpass2 (fc: float) (q: float)`

`para_eq` Generate coefficients for a peaking filter with normalized center frequency fc, gain (or cut) in dB, and half-gain width in octaves.
`((b0: float, b1: float, b2: float), (a1: float, a2: float)) = para_eq (fc: float) (gain: float) (width: float)`

C.7.2 Vector library

C.7.2.1 Math

`dot` Dot product of x and y.
`(x_dot_y: float) = dot (x: float vec) (y: float vec)`

C.7.2.2 Conversions between units

`oct_to_hz` Convert a vector of octave values to frequencies.
`(hz: float vec) = oct_to_hz (oct: float vec)`

C.7.2.3 Window functions

`window` Generate an n-point window by evaluating f over $[-1, 1]$.
`(fwin: 'a vec) = window (f: float -> 'a) (n: int)`

`cyclic_window` Generate an n-point cyclic window by evaluating f over $[-1, 1)$.
`(fwin: 'a vec) = cyclic_window (f: float -> 'a) (n: int)`

`raised_cosine` Generate an n-point raised-cosine window.
`(win: float vec) = raised_cosine (n: int)`

`cyclic_raised_cosine`
Generate an n-point cyclic raised-cosine window.
`(win: float vec) = cyclic_raised_cosine (n: int)`

`hamming` Generate an n-point Hamming window.
`(win: float vec) = hamming (n: int)`

`blackman3` Generate an n-point three-term Blackman-Harris -67dB window.
`(win: float vec) = blackman3 (n: int)`

`blackman4` Generate an n-point four-term Blackman-Harris -92dB window.

```
(win: float vec) = blackman4 (n: int)
```

C.7.2.4 Interpolation

`resample0` Resample x with the specified frequency and initial phase, using zero-order (sample and hold) resampling.

```
(y: 'a vec) = resample0 (f: float) (phase0: float)
                        (x: 'a vec)
```

`resample0_to` Zero-order resample x with the specified frequency and initial phase, truncating or zero-padding to the specified length.

```
(y: float vec) = resample0_to (len: int) (f: float)
                              (phase0: float)
                              (x: float vec)
```

`resample_linear` Resample x with the specified frequency and initial phase, using first-order (linear) resampling.

```
(y: 'a vec) = resample_linear (f: float) (phase0: float)
                              (x: 'a vec)
```

C.7.2.5 Table generation and lookup

`tablei` Look up each $[0, 1]$ element of the phase vector, using linear interpolation.

```
(y: float vec) = tablei (table: float vec) (phase: float vec)
```

`table_of_sine_harmonics`

Generate a length-n table of sine harmonics according to *amplitudes* vector (whose 0^{th} element is the fundamental).

```
(table: float vec) = table_of_sine_harmonics
                     (amplitudes: float vec) (n: int)
```

`table_of_sine`

Generate a length-n sine table.

```
(table: float vec) = table_of_sine (n: int)
```

C.7.2.6 Oscillators

`phasor` Generate n samples of a phasor ranging over $[0, 1)$, with the specified frequency and initial phase.

```
(x: float vec) = phasor (n: int) (f: float) (ph0: float)
```

`phasor_v` Generate n samples of a phasor ranging over $[0, 1)$, with the specified time-varying frequency and initial phase.

```
(x: float vec) = phasor_v (n: int) (f: float vec) (ph0: float)
```

`osci` Generate n samples of a linearly-interpolated table oscillator, with the specified frequency and initial phase.

```
(x: float vec) = osci (n: int) (table: float vec)
                      (f: float) (ph0: float)
```

`osci_v` Generate n samples of a linearly-interpolated table oscillator, with the specified time-varying frequency and initial phase.

```
(x: float vec) = osci_v (n: int) (table: float vec)
                        (f: float vec) (ph0: float)
```

`osc_sine` Generate n samples of a sinusoid with specified frequency and initial phase (using a neutrally stable filter).

```
(x: float vec) = osc_sine (n: int) (f: float) (ph0: float)
```

`noise_white` Generate n samples of white noise uniform over $[-1, 1)$.

```
(x: float vec) = noise_white (n: int)
```

C.7.2.7 Envelopes

`env_lin_ar` Generate an envelope lasting *dur* samples, with linear attack and release.

```
(env: float vec) = env_lin_ar (dur: time)
                              (attack: float) (release: float)
```

`declick` Apply linear fade-in and fade-out to declick the ends of x.

```
(x': float vec) = declick (x: float vec)
```

C.7.2.8 Filters

`filter1` Apply a first-order direct-II filter to x, with the specified coefficients. See `IV.filter1` for sign convention.

```
(y: float vec) = filter1 ((b0: float, b1: float), a1: float)
                         (x: float vec)
```

`filter2` Apply a second-order direct-II filter to x, with the specified coefficients. See `IV.filter2` for sign convention.

```
(y: float vec) = filter2 ((b0: float, b1: float, b2: float),
                          (a1: float, a2: float))
                         (x: float vec)
```

`lowpass2` Apply to x a second-order resonant lowpass filter with the specified normalized frequency and Q.

```
(y: float vec) = lowpass2 (fc: float) (q: float) (x: float vec)
```

`para_eq` Apply to x a peaking filter with normalized center frequency *fc*, gain (or cut) in dB, and half-gain width in octaves.

```
(y: float vec) = para_eq (fc: float) (gain: float)
                         (width: float) (x: float vec)
```

C.7.3 `ivec` library

C.7.3.1 Vectors and `ivecs`

`extend` Convert x to an `ivec` by repeating its last element.

```
(x': 'a ivec) = extend (x: 'a vec)
```

`finitize` Convert f from a function on `ivec`s to a function on vectors.
`(f': 'a vec -> 'b vec) = finitize (f: 'a ivec -> 'b ivec)`

C.7.3.2 Conversions between units

`oct_to_hz` Convert an `ivec` of octave values to frequencies.
`(hz: float ivec) = oct_to_hz (oct: float ivec)`

C.7.3.3 Oscillators

`phasor_wrap` Generate an `ivec` of the times (with sub-sample precision) at which a phasor with the specified time-varying frequency and initial phase will wrap back to zero phase.
`(x: float ivec) = phasor_wrap (f: float ivec) (ph0: float)`

C.7.3.4 Filters

`lowpass2` Apply to x a second-order resonant lowpass filter with the specified normalized frequency and Q.
```
(y: float ivec) = lowpass2 (fc: float) (q: float)
                           (x: float ivec)
```

`para_eq` Apply to x a peaking filter with normalized center frequency fc, gain (or cut) in dB, and half-gain width in octaves.
```
(y: float ivec) = para_eq (fc: float) (gain: float)
                          (width: float) (x: float ivec)
```

`allpass1` Apply to x a first-order allpass with gain g and delay length *len*.
```
(y: float ivec) = allpass1 (len: time) (gain: float)
                           (x: float ivec)
```

`allpass1_nest` Apply to x a first-order allpass with a processing element *fnest* nested within it after the delay element.
```
(y: float ivec) = allpass1_nest
                    (fnest: float ivec -> float ivec)
                    (len: time) (gain: float)
                    (x: float ivec)
```

Appendix D

Code listings

To back up Appendix C's informal description of the Chronic API, this chapter includes a full code listing of Chronic. Alternatively, Chronic is available in machine-readable form through `http://www.cs.cmu.edu/~eli/thesis/`.

The files listed here are the following:

	filename	module	type
core			
	`time.ml`		`time`
	`globals.ml`		
	`event.ml`	`E`	`'a event`
	`vec.ml`	`V`	`'a vec`
	`matrix.ml`	`M`	`'a vec vec` (matrices)
	`ivec.ml`	`IV`	`'a ivec`
	`evvec.ml`	`EV`	`'a event vec`
	`evivec.ml`	`EIV`	`'a event ivec`
library			
	`chronlib.ml`	`L`	mostly scalars
		`LV`	`'a vec`
		`LIV`	`'a ivec`
internal			
	`vecl.ml`	`VL`	`'a vec` by side-effect
	`util.ml`		

D.1 time.ml

```
type time = int
let time_0 = 0
let time_max = max_int
```

D.2 globals.ml

```
open Time;;

exception Unimplemented
exception Length of (time * time)
exception Length_underspecified

let floatfuzz = 1.0e-9
```

D.3 event.ml

```
open Time;;

type 'a event = Event of time * 'a

let (@@) v t = Event (t, v)
let (@!) (Event (t, v)) t' = Event (t',     v)
let (@+) (Event (t, v)) t' = Event (t + t', v)
let (@-) (Event (t, v)) t' = Event (t - t', v)
let (@*) (Event (t, v)) k  = Event (t * k,  v)
let (@/) (Event (t, v)) k  = Event (t / k,  v)
let (?@@) (Event (t, v)) = t
let (?@:) (Event (t, v)) = v

module E = struct
  let split (Event (t, v)) = (t, v)
  let v     (Event (_, v)) = v
  let t     (Event (t, _)) = t
  let diff  (Event (t1, _)) (Event (t2, _)) = t1 - t2

  let lift f =
    fun (Event (t, v)) -> Event (t, f v)
end
```

D.4 vec.ml

```
(* polymorphic vectors
 *
 *)

open Printf

open Globals
open Util

type 'a vec = 'a array

module V = struct
(** create *)
  let const xi n = Array.make n xi
  let copy x = Array.copy x

  let tabulate f n = Array.init n f
  let iterate f x0 n =
    let xi = ref x0 in
    let tabf i =
      let oldxi = !xi in (xi := f oldxi; oldxi)
    in tabulate tabf n

(** examine *)
  let length = Array.length
  let isempty x = (length x) == 0

(** convert *)
  let from_scalar x0 = const x0 1
  let from_list lis = Array.of_list lis
  let to_list x = Array.to_list x

(** take apart; put together *)
  let subvec n len x = Array.sub x n len
  let take n x =     Array.sub x 0                n
  let drop n x =     Array.sub x n                ((length x) - n)
  let take_end n x = Array.sub x ((length x) - n) n
  let drop_end n x = Array.sub x 0                ((length x) - n)
  let split n x = (Array.sub x 0 n, Array.sub x n ((length x) - n))
  let blit source isource dest idest len =
    Array.blit source isource dest idest len

  let join = Array.append
  let joinlist = Array.concat
  let delay n x0 x = join (const x0 n) x

  let ravel2 a = Array.concat (to_list a)

(** rearrange *)
  let reverse x =
    let n = length x
    in tabulate (fun i -> x.(n-1-i)) n

  let rotate_left n x =
    if (n >= 0)
    then join (drop n x) (take n x)
    else join (take_end (-n) x) (drop_end (-n) x)

(** index into *)
  let peek n x = x.(n)
  let poke n e x = x.(n) <- e
  let poke_copy n e x =
    let x' = copy x in (x'.(n) <- e; x')
  let peekv nv x = tabulate (fun i -> x.(nv.(i))) (length nv)
  let pokev nv ev x = Array.iteri (fun i ni -> x.(ni) <- ev.(i)) nv
  let pokev_copy nv ev x =
    let x' = copy x in (pokev nv ev x'; x')
  let fill start upto xi x = Array.fill x start (upto-start) xi
  let first x = x.(0)
  let last  x = x.((length x) - 1)

(** map *)
  let apply f x = Array.iter f x
  let applyi f x = Array.iteri f x
  let map f x = Array.map f x
  let mapi f x = Array.mapi f x
  let map2 f x y = mapi (fun i xi -> f xi y.(i)) x
  let map3 f x y z = mapi (fun i xi -> f xi y.(i) z.(i)) x

  (* [f xpre x0, f x0 x1, ..., f x(n-1) xn] *)
  let map_slide2 f xpre x =
    let ximinus1 = ref xpre in
    let tabf i =
      let xi = x.(i) in
      let fval = f !ximinus1 xi in (ximinus1 := xi; fval)
    in tabulate tabf (length x)

(** scan *)
  (* f: a b -> c.  _a_ is iterated upon, _b_ is running input, _c_ is output *)
  let sidescan f a0 b =
    let a = ref a0 in
    let tabf i =
      let (a', ci) = f !a b.(i) in (a := a'; ci)
    in tabulate tabf (length b)

  (* scan (+) 100 [1 3 17] = [100 101 104] *)
  let scan f y0 x = sidescan (fun a b -> (f a b, a)) y0 x

  (* scan' (+) 100 [1 3 17] = [101 104 121] *)
  let scan' f ypre x =
    sidescan (fun a b -> let fab = f a b in (fab, fab)) ypre x

  (* "scan extended" -- result is one longer than x *)
(*
  let scanx f a0 x =
    let a = ref a0 in
    let tabf i =
      let olda = !a in (a := f olda x.(i); olda)
    in tabulate tabf (1 + (length x))
*)
  let scanx f a0 x =
    let xlen = length x in
    let y = const a0 (1+xlen) in        (* QQQ uninit *)
    let a = ref a0 in
    (for i = 0 to (xlen-1) do
      (y.(i) <- !a;
       a := f !a x.(i))
     done;
     y.(xlen) <- !a;
```

```
    y)

(** fold *)
  let foldl f y0 x = Array.fold_left  f y0 x
  let foldr f x y0 = Array.fold_right f x y0

  (* foldl0 f x = foldl f x.(0) (drop 1 x) *)
  let foldl0 f x =
    let xlen = length x  in
    let a = ref x.(0)  in
    for i = 1 to xlen-1 do
      a := f !a x.(i)
    done;
    !a

(** filter, find, and friends *)

let filter f x =
  if (isempty x) then x else
  let xlen = length x in
  let temp = const x.(0) xlen in        (* the result, with junk on the end *)
  let j = ref 0 in
  for i = 0 to xlen-1 do
    if (f x.(i)) then (
      temp.(!j) <- x.(i);
      incr j
    )
  done;
  take !j temp

let find f x =
  if (isempty x) then [||] else
  let xlen = length x in
  let temp = const 0 xlen in            (* the result, with junk on the end *)
  let j = ref 0 in
  for i = 0 to xlen-1 do
    if (f x.(i)) then (
      temp.(!j) <- i;
      incr j
    )
  done;
  take !j temp

  let compress b x = peekv (find id b) x

(** zip and unzip *)
  let zip x y = tabulate (fun i -> (x.(i), y.(i))) (length x)
  let zip3 x y z = tabulate (fun i -> (x.(i), y.(i), z.(i))) (length x)

  let unzip xy = (map fst xy, map snd xy)
  let unzip3 xyz =
    (map (fun (e, _, _) -> e) xyz,
     map (fun (_, e, _) -> e) xyz,
     map (fun (_, _, e) -> e) xyz)

(*  any faster?
  let unzip xy =
    let x = const (fst xy.(0)) (length xy) in        (* QQQ uninit *)
    let f i (xi, yi) = (x.(i) <- xi; yi) in
    let y = mapi f xy
    in (x, y)
*)

(** strings *)
  let break_string_on dividers s =
    let rec loop lis from =
      try
        let start = index_not_in_from dividers from s  in
        let stop =
          try index_in_from dividers start s
          with Not_found -> String.length s  in
        let token = String.sub s start (stop-start)
        in loop (token::lis) stop
      with
        Not_found -> lis
    in from_list (List.rev (loop [] 0))

  let break_string_on_whitespace = break_string_on " \t"

  let sscanf_floats s =
    map (float_of_string) (break_string_on_whitespace s)

(** read/write *)
(*
  let printf eltfmt x =
    let eltfmt_newline = (eltfmt ^ "\n") in
    let f xi = printf eltfmt_newline xi
    in Array.iter f x
*)
  let print_ints    x = (Array.iter (fun xi -> printf "%d " xi) x; printf "\n")
  let print_floats  x = (Array.iter (fun xi -> printf "%g " xi) x; printf "\n")
  let print_strings x = (Array.iter (fun xi -> printf "%s " xi) x; printf "\n")

  (* read ASCII as floats from channel _ch_ until EOF *)
  let scanf_floats_from_channel ch =
    map float_of_string (from_list (getlines_from_channel ch))

  (* read ASCII as floats from _filename_ until EOF *)
  let scanf_floats_from filename =
    scanf_floats_from_channel (open_in filename)

  (* read ASCII as floats from stdin until EOF *)
  let scanf_floats () =
    scanf_floats_from_channel stdin

(** util: sequences *)

  let count_by from by n =
    tabulate (fun i -> from +. by *. (float i)) n

  let count_by_upto from by upto =      (* up to but not to *)
    let k = (upto -. from) /. by  in
    let nk = int_of_float k  in
    let n = nk + if ((k -. (float nk)) < floatfuzz) then 0 else 1
    in count_by from by n

  (* ew.  I want overloading back.  yeah yeah modules modules. *)
  let int_count_by from by n =
    tabulate (fun i -> from + by*i) n

  let int_count_by_upto from by upto =      (* up to but not to *)
    let k = (upto - from) / by  in
```

```
    let n = k + if (0 = ((upto-from) mod by)) then 0 else 1
    in int_count_by from by n

  let int_count min max = int_count_by min 1 (max-min+1)

(** util: specialized versions *)

  let scale k (x: float vec) =
    let out = copy x  in
    let xlen = length x  in
    for i = 0 to xlen-1 do
      out.(i) <- k *. out.(i)
    done;
    out

  let iterate_mul k (x0: float) n =
    let x = const x0 n  in              (* QQQ uninit *)
    let xi = ref x0  in
    for i = 0 to n-1 do
      x.(i) <- !xi;
      xi := !xi *. k
    done;
    x

  let mul3k k x y z =
    let n = length x  in
    let out = const 0. n  in            (* QQQ uninit *)
    for i = 0 to n-1 do
      out.(i) <- k *. x.(i) *. y.(i) *. z.(i)
    done;
    out

end


(****************************** testing ******************************)

let test () =
  let odds = V.iterate (fun i -> i+2) 1 5  in
  let squares = V.tabulate (fun i -> i*i) 4  in
  let sq3 = V.subvec 1 3 squares  in
  let evens = V.map (fun i -> i+1) odds  in
  let map_slide2_test = V.map_slide2 (fun x y->y-x) 17 squares  in
  let sidescan_test =
    V.sidescan (fun a b -> (a+b, string_of_int a)) 100 odds  in
  let zipped = V.zip [|1; 2|] [|10; 20|]  in
```

```
  let _ = assert (V.length [|1; 2; 3|] = 3)  in
  let _ = assert (V.const 3 2 = [|3; 3|])  in
  let _ = assert (squares = [|0; 1; 4; 9|])  in
  let _ = assert (odds =  [|1; 3; 5; 7; 9|])  in

  let _ = assert (V.take 3 odds = [|1; 3; 5|])  in
  let _ = assert (V.drop 3 odds = [|7; 9|])  in
  let _ = assert (sq3 = [|1; 4; 9|])  in
  let _ = assert (V.join [|1; 2|] [|3|] = [|1; 2; 3|])  in
  let _ = assert (V.reverse  [|1; 2; 3|] = [|3; 2; 1|])  in
  let _ = assert (V.rotate_left 1 [|1; 2; 3|] = [|2; 3; 1|])  in
  let _ = assert (V.rotate_left (-1) [|1; 2; 3|] = [|3; 1; 2|])  in

  let _ = assert (evens = [|2; 4; 6; 8; 10|])  in
  let _ = assert (V.map2 (-) evens odds = [|1; 1; 1; 1; 1|])  in
  let _ = assert (map_slide2_test = [|~-17; 1; 3; 5|])  in
  let _ = assert (V.foldl (-) 100 sq3 = List.fold_left  (-) 100 [1;4;9])  in
  let _ = assert (V.foldr (-) sq3 100 = List.fold_right (-) [1;4;9] 100)  in
  let _ = assert (V.foldl0 (+) sq3 = 1+4+9)  in
  let _ = assert (V.scan  (+) 100 sq3 = [|100; 101; 105|])  in
  let _ = assert (V.scanx (+) 100 sq3 = [|100; 101; 105; 114|])  in
  let _ = assert (V.scan' (+) 100 sq3 = [|101; 105; 114|])  in
  let _ = assert (sidescan_test = [|"100";"101";"104";"109";"116"|])  in
  let _ = assert (zipped = [|(1, 10); (2, 20)|])  in
  let _ = assert (V.unzip zipped = ([|1; 2|], [|10; 20|]))  in

  let evenP i =  (i mod 2) = 0  in
  let _ = assert (V.filter evenP squares = [|0; 4|])  in
  let _ = assert (V.find   evenP squares = [|0; 2|])  in

  let _ = assert (V.break_string_on "\t" "foo bar" = [|"foo"; "bar"|])  in
  let _ = assert (V.break_string_on "xy" "xxfoxboxyfoox" =
                  [|"fo"; "bo"; "foo"|])  in
  let _ = assert (V.sscanf_floats "  1.0 -2.0" = [|1.; -2.|])  in

  let _ = assert (V.int_count_by_upto 10 2 15 = [|10; 12; 14|])  in
  let _ = assert (V.int_count_by_upto 10 2 14 = [|10; 12|])  in
  let _ = assert (V.int_count 5 9 = [|5; 6; 7; 8; 9|])  in

(*
  let _ = assert (V.compress [|true;false;false;true|] squares = [|0; 9|])  in
*)
  (prerr_string "V ok\n"; flush stderr; true)

let _ = assert (test ())                (* a.out.opt strips it all away *)
```

D.5 matrix.ml

```
open Globals
open Util
open Vec


module M = struct

  (** finite matrices *)
  module V = struct

(** examine *)
    let rows a = V.length a
    let cols a = V.length a.(0)    (* obviously assumes rectangularness *)
    let dims a = (rows a, cols a)

(** create *)
    let tabulate f (m, n) =
      V.tabulate (fun i -> V.tabulate (fun j -> f i j) n) m

(** extract vectors *)

    let row i a = a.(i)
    let col i a = V.map (V.peek i) a

    (* ofs 0 gives main diagonal   [a00 a11 ...],
     * ofs 1 gives upper diagonal  [a01 a12 ...],
     * ofs -1 gives lower diagonal [a10 a21 ...], etc.
     * the matrix is expected to be rectangular. *)
    let diag ofs a =
      let (m, n) = dims a  in
      let k0 = max 0 (-ofs)  in
      let kmax = min m (n - ofs)  in
      V.tabulate (fun k' -> let k = k0+k' in a.(k).(k + ofs)) (kmax - k0)

    (* ofs 0 gives [a00], ofs 1 gives [a01 a10], etc.
     * different from diag ofs!  "main cross-diagonal" is less clearly defined.
     * the matrix is expected to be rectangular. *)
    let crossdiag ofs a =
      let (m, n) = dims a  in
      let k0 = max 0 (ofs - n + 1)  in
      let kmax = min m (ofs + 1)  in
      V.tabulate (fun k' -> let k = k0+k' in a.(k).(ofs - k)) (kmax - k0)

(** transpose, rotate, reflect *)
    let transpose a =
      tabulate (fun i j -> a.(j).(i)) (cols a, rows a)

    (* flip left and right (inner axis) *)
    let flip_lr a = V.map V.reverse a

    (* flip up and down (outmost axis) *)
    let flip_ud a = V.reverse a

    (* clockwise when matrix is laid out in the standard matrix way *)
    let rotate_cw a = flip_lr (transpose a)

    (* counterclockwise when matrix is laid out in the standard matrix way *)
    let rotate_ccw a = flip_ud (transpose a)

(** map *)
    let map f a = V.map (V.map f) a

(** output *)
    let print_ints a =
      Array.iter (fun row -> V.print_ints row) a

    let print_floats a =
      Array.iter (fun row -> V.print_floats row) a

(** input *)
  (* read lines of whitespace-delimited floats from channel _ch_ until EOF *)
  let scanf_floats_from_channel ch =
    V.map V.sscanf_floats (V.from_list (getlines_from_channel ch))

  (* read from _filename_ until EOF *)
  let scanf_floats_from filename =
    scanf_floats_from_channel (open_in filename)

  (* read from stdin until EOF *)
  let scanf_floats () =
    scanf_floats_from_channel stdin

  end

end


(****************************** testing *****************************)

let test () =
  let a = [| [| 1; 2; 3 |]; [| 10; 20; 30 |] |]  in

  let _ = assert (M.V.diag 0 a = [|1; 20|])  in
  let _ = assert (M.V.diag 2 a = [|3|])  in
  let _ = assert (M.V.diag (-1) a = [|10|])  in
  let _ = assert (M.V.diag 3 a = [| |])  in
  let _ = assert (M.V.diag (-2) a = [| |])  in

  let _ = assert (M.V.crossdiag 1 a = [|2; 10|])  in
  let _ = assert (M.V.crossdiag 3 a = [|30|])  in
  let _ = assert (M.V.crossdiag 0 a = [|1|])  in
  let _ = assert (M.V.crossdiag (-1) a = [| |])  in
  let _ = assert (M.V.crossdiag 4 a = [| |])  in

  let _ = assert (M.V.transpose a = [| [|1; 10|]; [|2; 20|]; [|3; 30|] |])  in
  let _ = assert (M.V.flip_lr a = [| [| 3; 2; 1 |]; [| 30; 20; 10 |] |])  in
  let _ = assert (M.V.flip_ud a = [| [| 10; 20; 30 |]; [| 1; 2; 3 |] |])  in
  let _ = assert (M.V.rotate_cw a = [| [|10; 1|]; [|20; 2|]; [|30; 3|] |])  in
  let _ = assert ((M.V.rotate_ccw (M.V.rotate_cw a)) = a)  in

  (prerr_string "M ok?\n"; flush stderr; true)

let _ = assert (test ())              (* a.out.opt strips it all away *)
```

D.6 ivec.ml

```
(* polymorphic infinite vectors
 *
 *)

open Globals
open Time
open Vec
open Vecl
open Util
open Printf


let err_rewind locname =
  (prerr_endline ("rewind error in IV." ^ locname); raise Exit)

let maxblocklen = 256
let optimal_blocklen = 256

let idref = ref 0
class virtual ['a] ivec_dat = object (self)
  val id = (incr idref; !idref)
  method get_id = id
  method virtual get_name: string

  (* buf contains [out_3 out_4 out_5]
   * buf_start = 3
   * buf_upto  = 6
   *)
  val mutable buf = ([||] : 'a array)          (* subclass must init *)
  val mutable buf_start = 0
  val mutable buf_upto = 0
  method get_buf = buf

  method dbgP = true                   (* override in subclass *)
  method dbg_entry new_upto =
    if (self#dbgP) then
      (eprintf "IV.%s %d compute: buf_upto = %d, new_upto = %d\n"
         self#get_name id buf_upto new_upto;  flush stderr)
  method dbg_recalc_hook = ()           (* abstract *)
  method dbg_recalc (new_upto: int) =
    if (self#dbgP) then
      (eprintf "IV.%s %d recalc: buf_start = %d, buf_upto = %d\n"
         self#get_name id buf_start buf_upto;  flush stderr;
       self#dbg_recalc_hook)
  method dbg_seek new_upto =
    eprintf "IV.%s %d seek: buf_upto = %d, new_upto = %d\n"
            self#get_name id buf_upto new_upto;
    flush stderr

(*  initializer
    eprintf "IV.%s %d created\n" self#get_name id   *)

  method compute_hook new_upto len = ()     (* abstract *)
  method compute new_upto =
    (assert (new_upto - buf_upto <= maxblocklen);
(*   self#dbg_entry new_upto;    *)
     if (new_upto = buf_upto) then ()
     else if (new_upto < buf_upto) then err_rewind (self#get_name ^ "#compute")
     else (buf_start <- buf_upto;
           buf_upto <- new_upto;     (* is it important to do this before
                                        the compute_hook call?  delay_rec
                                        would prefer it to be afterwards. *)
(*         self#dbg_recalc new_upto;  *)
           self#compute_hook new_upto (buf_upto - buf_start)
          )
    )

  (* performs intermediate calculation -- override if you don't need to *)
  method seek new_upto =
(*  self#dbg_seek new_upto; *)
    while (buf_upto < new_upto) do
        let upto = min (buf_upto + maxblocklen) new_upto
        in self#compute upto
    done

end

type 'a ivec = ('a ivec_dat) ref


module IV = struct

(** internal use *)

  let err_lost locname =
    (prerr_endline ("reference to lost past in IV." ^ locname); raise Exit)

  let err_internal locname =
    (prerr_endline ("internal error in IV." ^ locname); raise Exit)

(** create *)

  class ['a] const_dat xi = object inherit ['a] ivec_dat
    method get_name = "const"
    val xi = xi
    initializer
      buf <- V.const xi maxblocklen

    method seek new_upto = buf_upto <- new_upto
    method compute_hook new_upto len = ()   (* init buf, then leave it alone *)
  end
  let const (xi: 'a) = ref ((new const_dat xi) :> ('a ivec_dat))

  let zero = const 0.

  class ['a] iterate_dat f x0 = object inherit ['a] ivec_dat
    method get_name = "iterate"
    val f = f
    val mutable x0 = x0
    initializer
      buf <- V.const x0 maxblocklen

    method compute_hook new_upto len =
      x0 <- VL.iterate_and_spill buf len f x0
  end
  let iterate f (x0: 'a) = ref ((new iterate_dat f x0) :> ('a ivec_dat))

  class ['a] iterate_niladic_dat junk f = object inherit ['a] ivec_dat
    method get_name = "iterate_niladic"
```

```
    val f = f
    initializer
      buf <- V.const junk maxblocklen

    method compute_hook new_upto len =
      VL.iterate_niladic buf len f
  end
  let iterate_niladic junk f =
(*
    let f' = deepcopy f in              (* QQQ uninit.  aiee! but only once *)
    let (junk: 'a) = f' () in
*)
    ref ((new iterate_niladic_dat junk f) :> ('a ivec_dat))

  class ['a] tabulate_dat f = object inherit ['a] ivec_dat
    method get_name = "tabulate"
    val f = f
    initializer
      buf <- V.const (f 0) maxblocklen             (* assumes f is pure *)

    method seek new_upto = buf_upto <- new_upto    (* f pure here too *)
    method compute_hook new_upto len =
      VL.tabulate_offset_i buf len f buf_start
  end
  let tabulate (f: int -> 'a) = ref ((new tabulate_dat f) :> ('a ivec_dat))

(** take apart; put together *)

  (* take prefix of x.  consumes x. *)
  let take upto (x: 'a ivec) =
    let xbuf = !x#get_buf in
    let t = ref (min upto optimal_blocklen) in
    !x#compute !t;
    let head = V.const xbuf.(0) upto in     (* QQQ uninit *)
    (* all this absurd code duplication is so we could get xbuf.(0) for that *)
    V.blit xbuf 0 head 0 !t;
    while (!t < upto) do
      let len = min (upto - !t) optimal_blocklen in
      t := len + !t;
      !x#compute !t;
      V.blit xbuf 0 head (!t - len) len
    done;
    head

  (* take and convert into list.  again, consumes x. *)
  let take_li upto x =
    let vec = take upto x
    in V.to_list vec

  class ['a] drop_dat junk n (xin: 'a ivec) = object inherit ['a] ivec_dat
    method get_name = "drop"
    val n = n
    val x = xin
    val xbuf = !xin#get_buf
    initializer
      !xin#seek n;
      buf <- V.const junk maxblocklen

    method seek new_upto =
      !x#seek (new_upto + n);
      buf_upto <- new_upto
    method compute_hook new_upto len =
      (* XXX shouldn't we seek to n here *)
      !x#compute (new_upto + n);
      V.blit xbuf 0 buf 0 len
  end
  let drop_taking_exemplar junk n (x: 'a ivec) =
    if (n = 0) then x
    else ref ((new drop_dat junk n x) :> ('a ivec_dat))
  let drop n (x: 'a ivec) =
    if (n = 0) then x
    else let x0 = (take 1 x).(0)
         in drop_taking_exemplar x0 n x

  (* return (take n x, drop n x).  consumes x. *)
  let split n (x: 'a ivec) =
    if (n = 0) then ([||], x)
    else let taken = take n x  in
         let dropped = drop_taking_exemplar taken.(0) n x
         in (taken, dropped)

  class ['a] join_dat vin (xin: 'a ivec) = object inherit ['a] ivec_dat
    method get_name = "join"
    val v = V.copy vin
    val x = xin
    val mutable vi = 0
    val vlen = V.length vin
    val xbuf = !xin#get_buf
    initializer
      buf <- V.const vin.(0) maxblocklen
      (* v can't be length-0; wrapper catches that case. *)

    method seek new_upto =
      if (new_upto <= vlen)
      then vi <- new_upto
      else (vi <- vlen;
            !x#seek (new_upto - vlen));
      buf_upto <- new_upto

    method compute_hook new_upto len =    (* XXX reblock? *)
      if (vi = vlen)
      then (!x#compute (new_upto - vlen);
            V.blit xbuf 0 buf 0 len)
      else let vremaining = vlen - vi in
           let vpiece = min len vremaining in
           V.blit v vi buf 0 vpiece;
           vi <- vi + vpiece;
           if (vpiece < len)
           then (!x#compute (new_upto - vlen);
                 V.blit xbuf 0 buf vpiece (len - vpiece);
                 vi <- vlen)
  end
  let join (prefix: 'a vec) x =
    if (V.length prefix) = 0 then x
    else ref ((new join_dat prefix x) :> ('a ivec_dat))

  let spy n x =
    let (head, tail) = split n x
    in (head, join head tail)

(** map *)
```

```
class ['a, 'b] map_dat (junk: 'b) (f: 'a -> 'b) (xin: 'a ivec) =
object inherit ['b] ivec_dat
  method get_name = "map"
  val f = f
  val x = xin
  val xbuf = !xin#get_buf
  initializer
    buf <- V.const junk maxblocklen

  method seek new_upto =
    !x#seek new_upto;
    buf_upto <- new_upto
  method compute_hook new_upto len =
    !x#compute new_upto;
    VL.map buf len f xbuf
end
let map (junk: 'b) f x =
  ref ((new map_dat junk f x) :> ('b ivec_dat))

class ['a, 'b] mapi_dat (junk: 'b) (f: int -> 'a -> 'b) (xin: 'a ivec) =
object inherit ['b] ivec_dat
  method get_name = "mapi"
  val f = f
  val x = xin
  val xbuf = !xin#get_buf
  initializer
    buf <- V.const junk maxblocklen

  method seek new_upto =
    !x#seek new_upto;
    buf_upto <- new_upto
  method compute_hook new_upto len =
    !x#compute new_upto;
    VL.mapi_offset_i buf len f buf_start xbuf
end
let mapi (junk: 'b) f x =
  ref ((new mapi_dat junk f x) :> ('b ivec_dat))

class ['a, 'b, 'c] map2_dat (junk: 'c) (f: 'a -> 'b -> 'c)
    (xin: 'a ivec) (yin: 'b ivec) =
object inherit ['c] ivec_dat
  method get_name = "map2"
  val f = f
  val x = xin
  val y = yin
  val xbuf = !xin#get_buf
  val ybuf = !yin#get_buf
  initializer
    buf <- V.const junk maxblocklen

  method seek new_upto =
    !x#seek new_upto;
    !y#seek new_upto;
    buf_upto <- new_upto
  method compute_hook new_upto len =
    !x#compute new_upto;
    !y#compute new_upto;
    VL.map2 buf len f xbuf ybuf
end
let map2 (junk: 'c) f x y =
  ref ((new map2_dat junk f x y) :> ('c ivec_dat))

class add2_float_dat (xin: float ivec) (yin: float ivec) =
object inherit [float] ivec_dat
  method get_name = "add2_float"
  val x = xin
  val y = yin
  val xbuf = !xin#get_buf
  val ybuf = !yin#get_buf
  initializer
    buf <- V.const 0. maxblocklen

  method seek new_upto =
    !x#seek new_upto;
    !y#seek new_upto;
    buf_upto <- new_upto
  method compute_hook new_upto len =
    !x#compute new_upto;
    !y#compute new_upto;
    VL.add2 buf len xbuf ybuf
end
let add2_float x y =
  ref ((new add2_float_dat x y) :> (float ivec_dat))

class ['a, 'b, 'c, 'd, 'e] map4i_dat (junk: 'e) f
    (x1in: 'a ivec) (x2in: 'b ivec) (x3in: 'c ivec) (x4in: 'd ivec) =
object inherit ['e] ivec_dat
  method get_name = "map4i"
  val f = f
  val x1 = x1in
  val x2 = x2in
  val x3 = x3in
  val x4 = x4in
  val x1buf = !x1in#get_buf
  val x2buf = !x2in#get_buf
  val x3buf = !x3in#get_buf
  val x4buf = !x4in#get_buf
  initializer
    buf <- V.const junk maxblocklen

  method seek new_upto =
    !x1#seek new_upto;
    !x2#seek new_upto;
    !x3#seek new_upto;
    !x4#seek new_upto;
    buf_upto <- new_upto
  method compute_hook new_upto len =
    !x1#compute new_upto;
    !x2#compute new_upto;
    !x3#compute new_upto;
    !x4#compute new_upto;
    VL.map4i_offset_i buf len f buf_start
      x1buf x2buf x3buf x4buf
end
let map4i (junk: 'e) f x1 x2 x3 x4 =
  ref ((new map4i_dat junk f x1 x2 x3 x4) :> ('e ivec_dat))

class ['a, 'b] map_slide2_dat (junk: 'b) f xpre (xin: 'a ivec) =
object inherit ['b] ivec_dat
  method get_name = "map_slide2"
  val f = f
  val mutable oldx = xpre
```

```
    val x = xin
    val xbuf = !xin#get_buf
    initializer
      buf <- V.const junk maxblocklen

    method seek new_upto =
      !x#seek (new_upto - 1);
      !x#compute new_upto;
      oldx <- xbuf.(0);
      buf_upto <- new_upto
    method compute_hook new_upto len =
      !x#compute new_upto;
      oldx <- VL.map_slide2 buf len f oldx xbuf
  end
  (* [f xpre x0, f x0 x1, ...] *)
  let map_slide2 (junk: 'b) f xpre x =
    ref ((new map_slide2_dat junk f xpre x) :> ('b ivec_dat))

(** scan *)

  (* f: a b -> c.  _a_ is iterated upon, _b_ is running input, _c_ is output *)
  class ['a, 'b, 'c] sidescan_dat (junk: 'c) f (a0: 'a) (bin: 'b ivec) =
  object inherit ['c] ivec_dat
    method get_name = "sidescan"
    val f = f
    val mutable a0 = a0
    val b = bin
    val bbuf = !bin#get_buf
    initializer
      buf <- V.const junk maxblocklen

    method compute_hook new_upto len =
      !b#compute new_upto;
      a0 <- VL.sidescan_with_spill buf len f a0 bbuf
  end
  let sidescan (junk: 'c) f a0 b =
    ref ((new sidescan_dat junk f a0 b) :> ('c ivec_dat))

  (* [a0, f a0 b0, f (f a0 b0) b1, ...] *)
  let scan junk f a0 b = sidescan junk (fun a b -> (f a b, a)) a0 b

  (* [f a0 b0, f (f a0 b0) b1, ...] *)
  let scan' junk f a0 b = sidescan junk (fun a b -> pair_of (f a b)) a0 b

(** filter, find, and friends *)

(** peek *)

  (* t values strictly monotonic *)
  class ['a] peekiv_dat (junk: 'a) (tin: time ivec) (xin: 'a ivec) =
    object inherit ['a] ivec_dat
    method get_name = "peekiv"
    val t = tin
    val x = xin
    val tbuf = !tin#get_buf
    val xbuf = !xin#get_buf
    initializer
      buf <- V.const junk maxblocklen

    method seek new_upto =
      !t#seek new_upto;
      buf_upto <- new_upto
    method compute_hook new_upto len =  (* XXX reblock?  queer case. *)
      !t#compute new_upto;
      for i = 0 to len-1 do
        let ti = tbuf.(i)
        in (!x#seek ti;
            !x#compute (ti+1);
            buf.(i) <- xbuf.(0)
           )
      done
  end
  let peekiv (junk: 'a) t x =
    ref ((new peekiv_dat junk t x) :> ('a ivec_dat))

  let peekivr junk t x = peekiv junk (map 0 int t) x

(** convert *)

  let from_list li terminator =
    join (V.from_list li) (const terminator)

(** i/o *)

  (* apply f to initial elements of x.  consumes x.  *)
  let apply_upto upto blocklen (f: 'a -> unit) (x: 'a ivec) =
    let t = ref 0 in
    let xbuf = !x#get_buf in
    while (!t < upto) do
      let len = min (upto - !t) blocklen in
      t := len + !t;
      !x#compute !t;
      for i = 0 to len-1 do
        f xbuf.(i)
      done
    done
(* you'd like to just "ignore upto blocklen (map junk f x)",
 * but where do you get the junk? *)

  (* apply f to blocks of x.  consumes x.
     f gets blocklen and buf; it should use elts 0..blocklen-1 of buf. *)
  let block_apply_upto upto blocklen (f: int -> 'a vec -> unit) (x: 'a ivec) =
    let t = ref 0 in
    let xbuf = !x#get_buf in
    while (!t < upto) do
      let len = min (upto - !t) blocklen in
      t := len + !t;
      !x#compute !t;
      f len xbuf
    done

  (* print int ivec on stdout as text.  consumes x. *)
  let print_ints upto blocklen x =
    let f xi = (print_int xi; print_string "\n")
    in apply_upto upto blocklen f x

  (* print float ivec on stdout as text.  consumes x. *)
  (* should measure overhead of doing this with apply_upto... *)
  let print_floats upto blocklen x =
    let t = ref 0 in
    let xbuf = !x#get_buf in
    while (!t < upto) do
```

```
      let len = min (upto - !t) blocklen in
      t := len + !t;
      !x#compute !t;
      for i = 0 to len-1 do
        print_float xbuf.(i);
        print_string "\n"
      done
    done

  (* write float ivec on stdout as binary.  consumes x. *)
  (* XXX godawful slow because of output_binary_float *)
  let write_floats upto blocklen x =
    let t = ref 0 in
    let xbuf = !x#get_buf in
    while (!t < upto) do
      let len = min (upto - !t) blocklen in
      t := len + !t;
      !x#compute !t;
      for i = 0 to len-1 do
        output_binary_float stdout xbuf.(i)
      done
    done

  (* for timing tests.  again, consumes x. *)
  let ignore upto blocklen (x: 'a ivec) =
    let t = ref 0 in
    let xbuf = !x#get_buf in
    while (!t < upto) do
      let len = min (upto - !t) blocklen in
      t := len + !t;
      !x#compute !t
    done

  (* read ASCII as floats from channel _ch_ until EOF, then zeroes *)
  let scanf_floats_from_channel ch =
    let eof = ref false  in
    let f () =
      if (!eof) then 0.
      else try float_of_string (input_line ch)
           with End_of_file -> (
             eof := true;
             0.)
    in iterate_niladic 0. f

  (* read ASCII as floats from _filename_ until EOF, then zeroes *)
  let scanf_floats_from filename =
    scanf_floats_from_channel (open_in filename)

  (* read ASCII as floats from stdin until EOF, then zeroes *)
  let scanf_floats () =
    scanf_floats_from_channel stdin

(** reblocking *)

  (* you never want to have two consumers making different-length requests
   * of the same producer.  the workaround is to reblock one of the consumers
   * to match the other.  you identify the section of the DAG that's altering
   * block lengths, and bracket it with reblocking operators.
   *
   * a reblock_slave accepts requests of any block length, while only issuing
   * requests of a certain block length.
```

```
   * a reblock_master controls one or more reblock_slaves upstream of it,
   * setting their "certain block length".
   *
   * we hope the client programmer never has to see these functions.
   * instead, they should be used to wrap any operators that need it.
   *
   *)

class ['a] reblock_slave_dat junk xin = object inherit ['a] ivec_dat
  method get_name = "reblock_slave"
  val x = xin
  val xbuf = !xin#get_buf
  val mutable blocklen = 1

  val mutable xbuf_start = 0
  val mutable xbuf_upto = 0
  val mutable xbuf_read_upto = 0
  val mutable buf_write_upto = 0
  initializer
    buf <- V.const junk maxblocklen

  method set_blocklen bl =
    blocklen <- bl
    (* changing blocklen is safe; it's only used to advance xbuf_upto. *)

  (* haven't worked through what to do, so use default conservative seek. *)

  val mutable copy_upto = 0
  method compute_hook new_upto len =
    while (xbuf_read_upto < new_upto) do
      (* copy from xbuf to buf *)
      copy_upto <- (min new_upto xbuf_upto);     (* enough x to fulfill? *)
      let xbuf_ofs = xbuf_read_upto - xbuf_start  in
      let buf_ofs  = buf_write_upto - buf_start   in
      let len = copy_upto - xbuf_read_upto
      in V.blit xbuf xbuf_ofs buf buf_ofs len;   (* do the copy*)
      xbuf_read_upto <- copy_upto;
      buf_write_upto <- copy_upto;               (* update times *)

      (* get more x if we need it *)
      if (copy_upto < new_upto) then (           (* we ran out of x *)
        xbuf_start <- xbuf_upto;
        xbuf_read_upto <- xbuf_upto;
        xbuf_upto <- xbuf_upto + blocklen;       (* (only use of blocklen) *)
        !x#compute xbuf_upto
       )
    done
end

(* returns (slave as reblock_slave_dat ref, slave as ivec) *)
let reblock_slave (junk: 'a) x =
  let slave = new reblock_slave_dat junk x
  in (ref slave, ref (slave :> ('a ivec_dat)))

class ['a] reblock_master_dat junk xin slave = object inherit ['a] ivec_dat
  method get_name = "reblock_master"
  val x = xin
  val slave = slave
  val xbuf = !xin#get_buf
  initializer
    buf <- V.const junk maxblocklen
```

```
    method get_buf = xbuf               (* we pass x right on through *)

    (* haven't worked through what to do, so use default conservative seek. *)

    method compute_hook new_upto len =
      !slave#set_blocklen len;
      !x#compute new_upto
  end

  let reblock_master (junk: 'a) x slave =
    ref ((new reblock_master_dat junk x slave) :> ('a ivec_dat))

  (* f may not preserve block length, but f' does. *)
  let reblock (junka: 'a) (junkb: 'b) (f: 'a ivec -> 'b ivec) =
    let f' (x: 'a ivec) =
      let (slave, slave_as_ivec) = reblock_slave junka x
      in reblock_master junkb (f slave_as_ivec) slave
    in f'

(** delay *)

  (* for experimentation in test-feedback1.ml
   * its compute_hook new_upto asks x for (new_upto - 1),
   * whereas vdelay propagates new_upto along unchanged.
   *)
  class ['a] xdelay1_dat basis xin = object inherit ['a] ivec_dat
    method get_name = "xdelay1"
    val mutable x = xin
    val mutable xbuf = !xin#get_buf
    initializer
      buf <- V.const basis maxblocklen

    method compute_hook new_upto len =
      buf_upto <- buf_start;   (* so reentrant call doesn't think "rewind" *)
      if (len != 1) then (eprintf "len = %d at new_upto = %d\n" len new_upto;
                          flush stderr;  raise Exit);
      buf.(0) <-
        if (new_upto <= 1) then basis
        else (!x#compute (new_upto - 1);
              xbuf.(0));
      buf_upto <- new_upto              (* okay, now we're done *)

    method init_input (xnew: 'a ivec) =
      x <- xnew;
      xbuf <- !xnew#get_buf
  end
  let xdelay1_init basis x =             (* see test-feedback1.ml *)
    let as_d1 = new xdelay1_dat basis x
    in (ref (as_d1 :> ('a ivec_dat)),
        fun new_x -> as_d1#init_input new_x)
  let xdelay1 basis x = fst (xdelay1_init basis x)

  (* for experimentation in test-feedback1.ml
   *)
  class ['a] xdelay_dat basis xin = object inherit ['a] ivec_dat
    method get_name = "xdelay"
    val mutable x = xin
    val mutable xbuf = !xin#get_buf
    val delay = V.length basis
    initializer
    buf <- V.const basis.(0) maxblocklen

  method compute_hook new_upto len =
    buf_upto <- buf_start;   (* so reentrant call doesn't think "rewind" *)

    let basis_left = delay - buf_start  in

    (* copy data from basis, if basis_left > 0 *)
    for i = 0 to (min len basis_left)-1 do
      buf.(i) <- basis.(i + buf_start)
    done;

    (* compute x *)
    let x_upto = new_upto - delay  in
    if (x_upto > 0) then (
      !x#compute x_upto;
      let ofs = max 0 basis_left  in
      for i = ofs to len-1 do
        buf.(i) <- xbuf.(i - ofs)
      done
     );

    buf_upto <- new_upto              (* okay, now we're done *)

  method init_input (xnew: 'a ivec) =
    x <- xnew;
    xbuf <- !xnew#get_buf
end
let xdelay_init basis x =             (* see test-feedback1.ml *)
  let as_xd = new xdelay_dat basis x
  in (ref (as_xd :> ('a ivec_dat)),
      fun new_x -> as_xd#init_input new_x)
let xdelay basis x = fst (xdelay_init basis x)

(* for experimentation in test-feedback2.ml
 * accepts block requests of any length; makes requests <= delaylen
 * XXX not complete I don't think
 *)
class ['a] x2delay_dat basis xin = object inherit ['a] ivec_dat
  method get_name = "x2delay"
  val mutable x = xin
  val mutable xbuf = !xin#get_buf
  val delay = V.length basis
  val past = Delayline.create basis
  val mutable nu = 0
  val mutable oldnu = 0
  initializer
    buf <- V.const basis.(0) maxblocklen

  method compute_hook new_upto len =
    buf_upto <- buf_start;   (* so reentrant call doesn't think "rewind" *)

    nu <- buf_start;         (* subblock upto *)
    while (nu < new_upto) do
      oldnu <- nu;
      nu <- min (nu + delay) new_upto;   (* step by at most delay *)

      let xu = nu - delay in    (* x upto for subblock *)
      if (xu > 0) then (
        !x#compute xu;
        for i = oldnu to (nu-1) do
```

```
        buf.(i - buf_start) <- xbuf.(i - oldnu)
      done
     );
   done;

   buf_upto <- new_upto               (* okay, now we're done *)

  method init_input (xnew: 'a ivec) =
    x <- xnew;
    xbuf <- !xnew#get_buf
end
let x2delay_init basis x =             (* see test-feedback1.ml *)
  let as_xd = new x2delay_dat basis x
  in (ref (as_xd :> ('a ivec_dat)),
      fun new_x -> as_xd#init_input new_x)
let x2delay basis x = fst (x2delay_init basis x)

(* variable-length delay *)
class ['a] vdelay_dat din basis (xin: 'a ivec) =
object inherit ['a] ivec_dat
  method get_name = "vdelay"
  val d = din
  val past = Delayline.create basis
  val x = xin
  val dbuf = !din#get_buf
  val xbuf = !xin#get_buf
  initializer
    buf <- V.const basis.(0) maxblocklen

  method compute_hook new_upto len =
    !d#compute new_upto;
    !x#compute new_upto;
    for i = 0 to len-1 do
      let di = dbuf.(i)
      in if (di = 0)                  (* special case to support 0-delay *)
         then buf.(i) <- xbuf.(i)
         else buf.(i) <- Delayline.peek past di;
      Delayline.put past xbuf.(i)
    done
end
let vdelay (d: time ivec) (basis: 'a vec) x =
  ref ((new vdelay_dat d basis x) :> ('a ivec_dat))

(* variable-length feedback delay.
 * equivalent to  y = vdelay d basis (f y)  if that were legal.
 *)
class ['a] vdelay_rec_dat din (basis: 'a vec) f =
object (self) inherit ['a] ivec_dat
  method get_name = "vdelay_rec"
  val d = din
  val dbuf = !din#get_buf
  val past = Delayline.create basis
  val mutable fself = const basis.(0)            (* uninitialized *)
  val mutable fself_buf = [||]
  (* each subblock reuses the beginning of buf; buf' holds everything *)
  val mutable buf' = [||]
  initializer
    buf  <- V.const basis.(0) maxblocklen;
    buf' <- V.const basis.(0) maxblocklen;
    fself <- f (ref (self :> ('a ivec_dat)));
    fself_buf <- !fself#get_buf

  val mutable nu = 0               (* our internal new_upto, for subblocks *)
  val mutable old_nu = 0
  method compute_hook new_upto len =
    nu <- buf_start;
    buf_upto <- buf_start;            (* no subblocks done yet *)
    while (buf_upto < new_upto) do    (* loop over subblocks *)
(*        eprintf "\tbuf_upto %d, new_upto %d, nu %d\n" buf_upto new_upto nu;
          flush stderr;  *)

      (* how high can nu be without creating feedback within the subblock? *)
      old_nu <- nu;
      while (nu < new_upto  &&  dbuf.(nu-old_nu) - (nu-old_nu) > 0) do
        nu <- nu + 1
      done;

      (* copy from delay line to buf, where f will see it *)
(*        eprintf "\tnu %d, old_nu %d\n" nu old_nu;  flush stderr;  *)
      for i = 0 to (nu - old_nu - 1) do
        buf.(i) <- Delayline.peek past (dbuf.(i) - i)    (* fudge by i *)
(*              ; eprintf "\tbuf.(%d) <- %g\n" i buf.(i); flush stderr  *)
      done;
      (* and copy into buf' *)
      V.blit buf 0 buf' (old_nu - buf_start) (nu - old_nu);
(*        printf "\tbuf': ";  V.print_floats (V.take 4 buf');  flush stdout; *)

      (* compute f(self) *)
      buf_upto <- nu;                 (* so f's call to us will be a no-op *)
      !fself#compute nu;

      (* let our buf be f(self)'s *)
      for i = 0 to (nu - old_nu - 1) do
        Delayline.put past fself_buf.(i)
(*              ; eprintf "\tfself_buf.(%d) = %g\n" i fself_buf.(i)
            ; flush stderr  *)
      done
    done;

    V.blit buf' 0 buf 0 len           (* copy buf' to buf *)
(*      ; eprintf "\tlen %d\n" len;  flush stderr
      ; printf "\tbuf: ";  V.print_floats (V.take 4 buf);  flush stdout *)
end
let vdelay_rec (d: time ivec) (basis: 'a vec) (f: 'a ivec -> 'a ivec) =
  ref ((new vdelay_rec_dat d basis f) :> ('a ivec_dat))

(* with a two-arg feedback function: (input, old output) -> new output
 * equivalent to  y = vdelay d basis (f x y)  if that were legal.
 *)
let vdelay_rec2 xjunk (d: time ivec) (basis: 'a vec)
    (f: 'b ivec -> 'a ivec -> 'a ivec) (x: 'b ivec) =
  let (slave, slave_as_ivec) = reblock_slave xjunk x  in
  let f1 y = f slave_as_ivec y  in
  let yjunk = basis.(0)
  in reblock_master yjunk (vdelay_rec d basis f1) slave

(* constant-length delay *)
let delay d basis x = vdelay (const d) basis x

(* constant-length feedback delay
 * equivalent to  y = delay d basis (f y)  if that were legal.
 *)
```

```
  let delay_rec d basis f = vdelay_rec (const d) basis f

  (* constant-length delay with two-arg feedback *)
  let delay_rec2 xjunk d basis f x = vdelay_rec2 xjunk (const d) basis f x

  (* variable-length delay; zero basis *)
  let vdelayz d dmax x = vdelay d (V.const 0. dmax) x

  (* variable-length feedback delay; zero basis *)
  let vdelayz_rec d dmax f = vdelay_rec d (V.const 0. dmax) f

  (* constant-length delay; zero basis *)
  let delayz d x = delay d (V.const 0. d) x

  (* constant-length feedback delay; zero basis *)
  let delayz_rec d f = delay_rec d (V.const 0. d) f

  (* constant-length delay with two-arg feedback; zero basis *)
  let delayz_rec2 d f x = delay_rec2 0. d (V.const 0. d) f x

(** filter primitives *)

(*---
  (* first-order IIR filter in direct-I form.
   * y(n) = b1 x(n) + b2 x(n-1) - a2*y(n-1)      (note sign of a2)
   * Y(z) = (b1 + b2 z^-1) / (1 + a2 z^-1)
   *)
  class ['a] filter1_dat b1 b2 a2 x1init y1init xin =
  object inherit ['a] ivec_dat
    method get_name = "filter1"
    val x = xin
    val b1 = b1
    val b2 = b2
    val a2 = a2
    val xbuf = !xin#get_buf
    val mutable x1 = x1init
    val mutable y1 = y1init
    initializer
      buf <- V.const 0. maxblocklen

    method compute_hook new_upto len =
      !x#compute new_upto;
      for i = 0 to len-1 do
        buf.(i) <- b1 *. xbuf.(i) +. b2 *. x1 -. a2 *. y1;
        x1 <- xbuf.(i);
        y1 <- buf.(i)
      done
      (* would be faster to touch x1 and y1 only at block boundaries *)
  end
---*)

  (* first-order IIR filter in direct-II form.
   * y(n) = b0 x(n) + b1 x(n-1) - a1 y(n-1)      (note sign of a1)
   * H(z) = (b0 + b1 z^-1) / (1 + a1 z^-1)
   *)
  class ['a] filter1_dat (b0i, b1i) a1i x1init y1init xin =
  object inherit ['a] ivec_dat
    method get_name = "filter1"
    val x = xin
    val b0 = b0i
    val b1 = b1i
  val a1 = a1i
  val xbuf = !xin#get_buf
  initializer
    buf <- V.const 0. maxblocklen

  val mutable z1 =
    let numerator = b1i *. x1init -. a1i *. y1init
    in if ((abs_float numerator) < 1.0e-6) then 0.
    else numerator /. (b1i -. b0i *. a1i)
  method compute_hook new_upto len =
    !x#compute new_upto;
    for i = 0 to len-1 do
      let z0 = xbuf.(i) -. a1*.z1
      in (buf.(i) <- b0*.z0 +. b1*.z1;
          z1 <- z0)
    done
    (* would be faster to touch z1 only at block boundaries *)
end

let filter1 ((b0, b1), a1) x =
  ref ((new filter1_dat (b0, b1) a1 0. 0. x) :> ('a ivec_dat))

(* time-varying first-order IIR filter in direct-I form.
 * y(n) = b0(n) x(n) + b1(n) x(n-1) - a1(n) y(n-1)      (note sign of a1)
 * Y(z) = (b0 + b1 z^-1) / (1 + a1 z^-1)
 *)
class ['a] filter1v_dat (b0i, b1i) a1i x1init y1init xin =
object inherit ['a] ivec_dat
  method get_name = "filter1v"
  val x = xin
  val b0 = b0i
  val b1 = b1i
  val a1 = a1i
  val xbuf = !xin#get_buf
  val b0buf = !b0i#get_buf
  val b1buf = !b1i#get_buf
  val a1buf = !a1i#get_buf
  val mutable x1 = x1init
  val mutable y1 = y1init
  initializer
    buf <- V.const 0. maxblocklen

  method compute_hook new_upto len =
    !x#compute new_upto;
    !b0#compute new_upto;
    !b1#compute new_upto;
    !a1#compute new_upto;
    for i = 0 to len-1 do
      buf.(i) <- b0buf.(i)*.xbuf.(i) +. b1buf.(i)*.x1 -. a1buf.(i)*.y1;
      x1 <- xbuf.(i);
      y1 <- buf.(i)
    done
    (* would be faster to touch x1 and y1 only at block boundaries *)
end

let filter1v ((b0, b1), a1) xinit yinit x =
  ref ((new filter1v_dat (b0, b1) a1 xinit yinit x) :> ('a ivec_dat))

(* second-order IIR filter in direct-II form.
 * y(n) = b0 x(n) + b1 x(n-1) + b2 x(n-2)
 *                - a1 y(n-1) - a2 y(n-2)    (note signs)
```

```
   * H(z) = (b0 + b1 z^-1 + b2 z^-2) / (1 + a1 z^-1+ a2 z^-2)
   *)
  class ['a] filter2_dat (b0, b1, b2) (a1, a2) (z1, z2) xin =
  object inherit ['a] ivec_dat
    method get_name = "filter2"
    val x = xin
    val b0 = b0
    val b1 = b1
    val b2 = b2
    val a1 = a1
    val a2 = a2
    val xbuf = !xin#get_buf
    initializer
      buf <- V.const 0. maxblocklen

    val mutable z1 = z1
    val mutable z2 = z2
    method compute_hook new_upto len =
      !x#compute new_upto;
      for i = 0 to len-1 do
        let z0 = xbuf.(i) -. a1*.z1 -. a2*.z2
        in (buf.(i) <- b0*.z0 +. b1*.z1 +. b2*.z2;
            z2 <- z1;
            z1 <- z0)
      done
      (* would be faster to touch z1 and z2 only at block boundaries *)

  end

  let filter2 (btriple, apair) x =
    ref ((new filter2_dat btriple apair (0., 0.) x) :> ('a ivec_dat))

  (* filter2 with initial conditions for delay line *)
  let filter2_ic (btriple, apair) (z1, z2) x =
    ref ((new filter2_dat btriple apair (z1, z2) x) :> ('a ivec_dat))

(** specialized instances *)

  class count_by_dat from by = object inherit [float] ivec_dat
    method get_name = "count_by"
    val mutable x = from
    val by = by
    initializer
      buf <- V.const 0. maxblocklen

    method seek new_upto =
      x <- x +. by*.(float (new_upto - buf_upto));
      buf_upto <- new_upto

    method compute_hook new_upto len =
      for i = 0 to len-1 do
        buf.(i) <- x;
        x <- x +. by
      done
  end
  let count_by from by = ref ((new count_by_dat from by) :> (float ivec_dat))
  let count from = count_by from 1.

  let scale k x = map 0. (fun xi -> k*.xi) x

(**********

  class ['a] fun_dat junk? args = object inherit ['a] ivec_dat
    method get_name = "fun"
    val
    val mutable
    initializer
      buf <- V.const junk maxblocklen

    method seek new_upto = buf_upto <- new_upto          ??
    method compute_hook new_upto len =

  end

  let fun junk? args =
    ref ((new fun_dat junk? args) :> ('a ivec_dat))

**********)

end


(****************************** testing *****************************)

let test () =
  let _ = assert ((IV.take_li 3 (IV.const 3)) = [3; 3; 3])  in
  let count = IV.iterate (fun x -> x+1) 0  in
  let _ = assert ((IV.take_li 5 count) = [0; 1; 2; 3; 4])  in

  let nilf =
    let state = ref 1 in
    fun () -> let oldstate = !state in (state := 2*oldstate; oldstate)  in
  let _ = assert (IV.take_li 4 (IV.iterate_niladic 0 nilf) = [1; 2; 4; 8]) in

  let squares = IV.tabulate (fun x -> x*x) in
  let _ = assert (IV.take_li 4 squares = [0; 1; 4; 9]) in

  let prefix = [|10; 20; 30|] in
  let join = IV.join prefix (IV.iterate succ 0) in
  let _ = assert (IV.take_li 6 join = [10; 20; 30; 0; 1; 2]) in
  let join0 = IV.join [||] (IV.iterate succ 0) in
  let _ = assert (IV.take_li 3 join0 = [0; 1; 2]) in

  let drop = IV.drop 3 (IV.iterate succ 0) in
  let _ = assert (IV.take_li 5 drop = [3; 4; 5; 6; 7]) in
  let drop0 = IV.drop 0 (IV.iterate succ 0) in
  let _ = assert (IV.take_li 3 drop0 = [0; 1; 2]) in

  let evens = IV.map 0 (fun x -> x * 2) (IV.iterate succ 0) in
  let _ = assert (IV.take_li 4 evens = [0; 2; 4; 6]) in

  let zeros = IV.mapi 0 (fun i x -> i - x) (IV.iterate succ 0) in
  let _ = assert (IV.take_li 4 zeros = [0; 0; 0; 0]) in

  let count = (IV.iterate succ 0) in
  let evens = (IV.map2 0 (+) count count) in
  let _ = assert (IV.take_li 4 evens = [0; 2; 4; 6]) in

  let fss a b = (a+b, string_of_int (a+b)) in
  let sscan = IV.sidescan "" fss 100 (IV.iterate succ 0) in
  let _ = assert (IV.take_li 4 sscan = ["100"; "101"; "103"; "106"]) in
  let scan = IV.scan 0 (+) 100 (IV.iterate succ 0) in
```

```
let _ = assert (IV.take_li 4 scan = [100; 100; 101; 103]) in
let scan' = IV.scan' 0 (+) 100 (IV.iterate succ 0) in
let _ = assert (IV.take_li 4 scan' = [100; 101; 103; 106]) in

let evens = IV.map 0 (fun x -> x * 2) (IV.iterate succ 0) in
let t = IV.tabulate (fun x -> x*x) in
let evens_at_t = IV.peekiv 0 t evens in
let _ = assert (IV.take_li 5 evens_at_t = [0; 2; 8; 18; 32]) in

let delay = IV.join (V.const 4 3) (IV.const 2) in
let basis = V.count_by 10. 1. 4 in
let vdelay = IV.vdelay delay basis (IV.count 0.) in
let _ = assert (IV.take_li 5 vdelay = [10.; 11.; 12.; 1.; 2.]) in
let vdelay0 = IV.vdelay (IV.const 0) basis (IV.count 0.) in
let _ = assert (IV.take_li 1 vdelay0 = [0.]) in
let delayz = IV.delayz 2 (IV.const 3.) in
let _ = assert (IV.take_li 4 delayz = [0.; 0.; 3.; 3.]) in

let add1 = IV.map 0. (fun x -> x +. 1.) in
```

```
  let vdelay_rec_1 = IV.vdelay_rec (IV.const 4) basis add1 in
  let _ = assert (IV.take_li 5 vdelay_rec_1 = [10.; 11.; 12.; 13.; 11.]) in
  let vdelay_rec_2 = IV.vdelay_rec (IV.const 1) [|0.|] add1 in
  let _ = assert (IV.take_li 3 vdelay_rec_2 = [0.; 1.; 2.]) in

  let impulse = IV.iterate (fun _ -> 0.) 1. in
  let filter1 = IV.filter1 ((0.5, 0.6), 0.7) impulse in
(*let _ = IV.print_floats 32 10 filter1 in  *)
  let filter2 = IV.filter2 ((0.9, 0.8, 0.7), (0.6, 0.5)) impulse in
(*let _ = IV.print_floats 32 10 filter2 in   *)

(*----
  let _ = assert (IV.take_li 4 (IV. ) = []) in
----*)

  (prerr_string "IV ok\n"; flush stderr; true)

let _ = assert (test ())               (* a.out.opt strips it all away *)
```

D.7 evvec.ml

```
open Time
open Event
open Vec
open Ivec
open Evivec

module EV = struct
  (* piecewise constant from breakpoint vector *)
  let pwc x0 evs =
    let nevs = V.length evs  in
    let tmax = E.t (V.last evs)  in
    let target = V.const x0 tmax  in    (* QQQ uninit *)
    let rec loop ievs xt0 t0 =
      if (ievs >= nevs) then ()
      else let (Event (t1, xt1)) = evs.(ievs)
           in (V.fill t0 t1 xt0 target;
               loop (ievs+1) xt1 t1)
    in (loop 0 x0 0;
        target)

  (* piecewise linear from first-derivative breakpoint vector *)
  let dpwl x0 devs = V.scanx (+.) x0 (pwc 0.0 devs)

  (* piecewise linear from breakpoint vector *)
  let pwl x0 evs =
    let diff (Event (t0, x0)) (Event (t1, x1)) =
      Event (t0, (x1 -. x0) /. (float (t1 - t0)))
    in dpwl x0 (V.map_slide2 diff (Event (0, x0)) evs)

  (* local helper to vfold, vmix *)
  let vhelper vop x =
    let last = V.last x  in
    let n = (E.t last) + (V.length (E.v last))  in
    let junktail = IV.const ((E.v x.(0)) @@ time_max)  in
    let x' = IV.join x junktail
    in IV.take n (vop x')

  (* mix grains by folding vertically. *)
  let vfold f e x = vhelper (EIV.vfold f e) x

  (* vfold specialized with f being (+.) *)
  let vmix      x = vhelper (EIV.vmix) x


(** utility stuff *)
  let close evs =
    if (V.isempty evs) then evs
    else V.join evs (V.from_scalar ((V.last evs) @+ 1))

  let from_list pairlist =
    V.from_list (List.map (fun (t, v) -> v @@ t) pairlist)

  let pwc_list x0 pairlist = pwc x0 (close (from_list pairlist))
  let pwl_list x0 pairlist = pwl x0 (close (from_list pairlist))
end;;

(****************************** testing *****************************)

let test () =
  let knees = EV.close (EV.from_list [(2, 1.0); (4, 0.0)])  in
  let pwctest = EV.pwc (-1.0) knees  in
  let pwltest = EV.pwl (-1.0) knees  in
  let mixme = V.tabulate (fun n -> Event (n*n, V.const n 4)) 4  in
  let mixed = EV.vfold (+) 0 mixme  in

  let _ = assert (pwctest = [|-1.0; -1.0; 1.0; 1.0; 0.0|])  in
  let _ = assert (pwltest = [|-1.0;  0.0; 1.0; 0.5; 0.0|])  in
  let _ = assert (mixed = [|0; 1; 1; 1; 3; 2; 2; 2; 0; 3; 3; 3; 3|])  in

  (prerr_string "EV ok\n"; flush stderr; true)

let _ = assert (test ())               (* a.out.opt strips it all away *)
```

D.8 evivec.ml

```
open Time
open Event
open Ivec
open Vec
open Util

(*
type 'a evs_pop_rec =
    { evs: 'a event ivec;
      buf: 'a event vec;
      mutable ei: int }

let evs_pop_wrap evs =
  { evs = evs; buf = !evs#get_buf; ei = 0 }

let evs_pop pr =
  pr.ei <- pr.ei + 1;
  !(pr.evs)#compute pr.ei;
  pr.buf.(0)
*)

module EIV = struct
  (* piecewise constant ivec from breakpoint ivec *)
  class ['a] pwc_dat x0 evs_in = object inherit ['a] ivec_dat
    method get_name = "pwc"
    val mutable x0 = x0
    val evs = evs_in
    val evsbuf = !evs_in#get_buf
    val mutable ei = 0
    val mutable x1 = 0.                  (* these get filled in by init *)
    val mutable t1 = 0
    initializer
      buf <- V.const x0 maxblocklen;
      ei <- ei + 1;
      !evs_in#compute ei;
      let ev1 = evsbuf.(0)
      in (x1 <- E.v ev1;
          t1 <- E.t ev1)

    method dbg_recalc_hook = eprintf "pwc recalc\n"

   (* iterates over evs, w/ V.fill
    * for very frequent events, iterating over buf might be faster.
    *)
    method compute_hook new_upto len =
      let i0 = ref 0  in
      let i1 = ref (t1 - buf_start)  in
      while (!i0 < len) do
        if (!i1 > len)
        then (V.fill !i0 len x0 buf;
              i0 := len)
        else (V.fill !i0 !i1 x0 buf;
              i0 := !i1;
              x0 <- x1;
              ei <- ei + 1;
              !evs#compute ei;
              let ev1 = evsbuf.(0)
              in (x1 <- E.v ev1;
                  t1 <- E.t ev1;
                  i1 := t1 - buf_start)
             )
    done
end
let pwc (x0: 'a) evs = ref ((new pwc_dat x0 evs) :> 'a ivec_dat)

(* piecewise linear from first-derivative breakpoint ivec *)
let dpwl x0 devs =
  let dx = pwc 0.0 devs
  in IV.scan 0. (+.) x0 dx

(* piecewise linear from breakpoint vector *)
let pwl x0 evs =
  let diff (Event (t0, x0)) (Event (t1, x1)) =
    Event (t0, (x1 -. x0) /. (float (t1 - t0))) in
  let junk = Event (0, x0)
  in dpwl x0 (IV.map_slide2 junk diff (Event (0, x0)) evs)



type 'a life = { mutable dat: 'a vec; mutable len: int; mutable t: int }
let maxlive = 50

(* mixes grains by folding vertically. *)
class ['a] vfold_dat f ein grains_in = object inherit ['a] ivec_dat
  method get_name = "vfold"
  val f = f
  val e = ein
  val grains = grains_in
  val mutable grainsi = 0
  val grainsbuf = !grains_in#get_buf
  val mutable t1 = 0                  (* filled in by init *)
  val mutable v1 = [||]
  val mutable nlive = 0
  val mutable live = V.const { dat = [||]; len = 0; t = 0 } maxlive
  initializer
    buf <- V.const e maxblocklen;
    grainsi <- 1;
    !grains_in#compute grainsi;
    let g = grainsbuf.(0)
    in (v1 <- E.v g;
        t1 <- E.t g);
    for i = 0 to maxlive-1 do    (* double init is annoying *)
      live.(i) <- { dat = [||]; len = 0; t = 0 }
    done

  method dbg_recalc_hook = eprintf "vfold recalc\n"

 (* assumptions: substantial time between grain births;
  *              grains don't vary wildly in length;
  *              moderate (<20?) number alive at once.
  *)
  val mutable ax = ein
  val mutable i0 = 0
  val mutable ix = 0
  val mutable j = 0
  method compute_hook new_upto len =
    i0 <- 0;
    while (i0 < len) do
```

```
        let i1 = t1 - buf_start  in
(*      eprintf "i0 = %d, i1 = %d, len = %d, nlive = %d\n" i0 i1 len nlive;  *)

        ix <- i1;                          (* to start of next grain? *)
        if (ix > len) then ix <- len;      (* constrain by end of request *)
        for j = 0 to nlive-1 do            (* constrain by ends of grains *)
          let grain_upto_t = live.(j).t + live.(j).len  in
          let grain_upto_i = grain_upto_t - buf_start
          in if (grain_upto_i < ix) then
            (
(*           eprintf "   grain_upto_i = %d\n" grain_upto_i;  *)
             ix <- grain_upto_i)
        done;

(*      eprintf "ix = %d\n" ix;  *)

        for i = i0 to (ix-1) do            (* do the vfold *)
          ax <- e;
(*        eprintf "   vfold i = %d\n" i;  *)
          let t = i + buf_start in
          for j = 0 to nlive-1 do
            let pos = t - live.(j).t  in
(*          eprintf "      j = %d, t = %d, pos = %d\n" j live.(j).t pos;  *)
            flush stderr;
            ax <- f ax live.(j).dat.(pos)
(*          ; eprintf "         ax = %g\n" ax  *)
          done;
          buf.(i) <- ax
        done;

        i0 <- ix;                          (* advance i0 *)

        j <- 0;
        while (j < nlive) do               (* update and purge live *)
          let lj = live.(j)  in
(*        eprintf "   j = %d, t = %d\n" j lj.t;  *)
          if (ix + buf_start >= lj.t + lj.len) then    (* grain used up? *)
            (Array.blit live (j+1) live j (nlive-j-1);
(*           eprintf "   purging j = %d: t = %d, len = %d\n" j lj.t lj.len;  *)
             nlive <- nlive - 1)                          (* delete grain *)
          else j <- j+1
        done;
(*
        for j = 0 to nlive-1 do
          eprintf "      yo1 j = %d, t = %d\n" j live.(j).t
        done;
*)
        if (ix = i1) then                  (* new grain starts here *)
         (assert (nlive < maxlive);
          let li = live.(nlive) in            (* add it to the live set *)
           (live.(nlive) <- { dat = v1; len = V.length v1; t = buf_start+i1 };
(*          eprintf "      nlive <- %d\n" (1+nlive);  *)
            nlive <- 1 + nlive);

          grainsi <- grainsi + 1;
          !grains#compute grainsi;
          let e = grainsbuf.(0)
          in (t1 <- E.t e;                     (* examine next grain *)
              v1 <- E.v e)
         );
(*
        for j = 0 to nlive-1 do
          eprintf "      yo2 j = %d, t = %d\n" j live.(j).t
        done
*)
      done                             (* while i0 < len *)
  end
  let vfold f (e: 'a) x = ref ((new vfold_dat f e x) :> 'a ivec_dat)

  (* mixes grains by adding vertically. *)
  class vmix_dat grains_in = object inherit [float] ivec_dat
    method get_name = "vmix"
    val grains = grains_in
    val mutable grainsi = 0
    val grainsbuf = !grains_in#get_buf
    val mutable t1 = 0                  (* filled in by init *)
    val mutable v1 = [||]
    val mutable nlive = 0
    val mutable live = V.const { dat = [||]; len = 0; t = 0 } maxlive
    initializer
      buf <- V.const 0. maxblocklen;
      grainsi <- 1;
      !grains_in#compute grainsi;
      let g = grainsbuf.(0)
      in (v1 <- E.v g;
          t1 <- E.t g);
      for i = 0 to maxlive-1 do    (* double init is annoying *)
        live.(i) <- { dat = [||]; len = 0; t = 0 }
      done

    method dbg_recalc_hook = eprintf "vfold recalc\n"

   (* assumptions: substantial time between grain births;
    *              grains don't vary wildly in length;
    *              moderate (<20?) number alive at once.
    *)
    val mutable ax = 0.
    val mutable i0 = 0
    val mutable ix = 0
    val mutable j = 0
    method compute_hook new_upto len =
      i0 <- 0;
      while (i0 < len) do
        let i1 = t1 - buf_start  in

        ix <- i1;                          (* to start of next grain? *)
        if (ix > len) then ix <- len;      (* constrain by end of request *)
        for j = 0 to nlive-1 do            (* constrain by ends of grains *)
          let grain_upto_t = live.(j).t + live.(j).len  in
          let grain_upto_i = grain_upto_t - buf_start
          in if (grain_upto_i < ix) then
            ix <- grain_upto_i
        done;

        for i = i0 to (ix-1) do            (* do the vfold *)
          ax <- 0.;
          let t = i + buf_start in
          for j = 0 to nlive-1 do
            let pos = t - live.(j).t  in
            flush stderr;
            ax <- ax +. live.(j).dat.(pos)
          done;
```

```
        buf.(i) <- ax
      done;

      i0 <- ix;                          (* advance i0 *)

      j <- 0;
      while (j < nlive) do               (* update and purge live *)
        let lj = live.(j)  in
        if (ix + buf_start >= lj.t + lj.len) then    (* grain used up? *)
          (Array.blit live (j+1) live j (nlive-j-1);
           nlive <- nlive - 1)                          (* delete grain *)
        else j <- j+1
      done;

      if (ix = i1) then                  (* new grain starts here *)
       (assert (nlive < maxlive);
        let li = live.(nlive) in           (* add it to the live set *)
         (live.(nlive) <- { dat = v1; len = V.length v1; t = buf_start+i1 };
          nlive <- 1 + nlive);

        grainsi <- grainsi + 1;
        !grains#compute grainsi;
        let e = grainsbuf.(0)
        in (t1 <- E.t e;                     (* examine next grain *)
            v1 <- E.v e)
       );

    done                            (* while i0 < len *)
  end
  let vmix x = ref ((new vmix_dat x) :> float ivec_dat)


(*  slower, I should think
  (* piecewise constant from breakpoint vector *)
  let pwc x0 evs =
    let rec loop t0 xt0 evs =
      let (Event (t1, xt1), evs_tail) = IV.split1 evs
      in IV.join (V.const xt0 (t1-t0)) (loop t1 xt1 evs_tail)
    in loop 0 x0 evs
*)

(*
  class ['a] chain_dat f x0 evs_in = object inherit ['a] ivec_dat
    method get_name = "chain"
    val f = f
    val mutable xcurr = x0
    val mutable state = state0
    val evs = evs_pop_wrap evs_in
    val mutable v1 = ejunk               (* next ev -- filled in by init *)
    val mutable t1 = 0
    initializer
      buf <- V.const x0 maxblocklen;
      (x1, t1) <- E.split (evs_pop evs)

    method compute_hook new_upto len =

  end
  let chain f (x0: 'a) evs = ref ((new chain_dat f x0 evs) :> ('a chain_dat))
*)


(** utility stuff *)

  let from_list pairlist =
    let evlist = List.map (fun (t, v) -> v @@ t) pairlist  in
    let (_, lastval) = List.hd (List.rev pairlist)  in
    let terminator = Event (time_max, lastval)
    in IV.from_list evlist terminator

  let pwc_list x0 pairlist = pwc x0 (from_list pairlist)
  let pwl_list x0 pairlist = pwl x0 (from_list pairlist)
end


(****************************** testing *****************************)

let test () =
  let mixme = IV.tabulate (fun n -> Event (n*n, V.const n 4))  in
  let mixed = IV.take 10 (EIV.vfold (+) 0 mixme)  in
  let _ = assert (mixed = [|0; 1; 1; 1; 3; 2; 2; 2; 0; 3|])  in

  (prerr_string "EIV ok?\n"; flush stderr; true)

let _ = assert (test ())                (* a.out.opt strips it all away *)
```

D.9 chronlib.ml

```
(* Chronic library.
 *
 *)

open Util
open Time
open Event
open Vec
open Ivec
open Evvec


(******** scalar library ********)

module L = struct

        (** math **)

  let pi = 4.0 *. atan(1.0)
  let subunity = 1.0 -. (2.0 ** (-50.0))

  let sinc x =
    if x=0.0 then 1.0 else let pix = pi*.x in (sin pix)/.pix

        (** conversions between units **)

  let amp_to_db a = 20.0 *. (log10 a)
  let db_to_amp d = 10.0 ** (d /. 20.0)

  let middleC = 440. /. (2. ** (9./.12.))
  let middleC_octave = 5.                  (* MIDI octave convention *)

  let oct_to_hz oct =
    (2. ** (oct -. middleC_octave)) *. middleC
  let oct_to_ratio oct = 2. ** oct

  let cents_to_oct c = c /. 1200.
  let cents_to_ratio c = oct_to_ratio (cents_to_oct c)

  let notenum_to_hz note =                 (* 60 is middle C *)
    oct_to_hz ((float (note - 60)) /. 12. +. middleC_octave)

        (** windows **)

  let f_raised_cosine x =
    0.5 +. (0.5 *. cos(pi*.x))
  let f_hamming x =
    0.54 +. (0.46 *. cos(pi*.x))
  let f_blackman3 x =
    0.42 +. (0.5 *. cos(pi*.x)) +. (0.08 *. cos(2.0 *. pi*.x))
  let f_blackman4 x =
    0.35875 +. (0.48829 *. cos(pi*.x)) +. (0.14128 *. cos(2.0 *. pi*.x))
      +. (0.01168 *. cos(3.0 *. pi*.x))

        (** interpolation **)

  let interp1 t x =
    let a = frac t
    and i = int t
    in (1.0-.a) *. x.(i)  +.  a *. x.(i+1)

        (** filter coefficients **)

  let filter_helper fc =
    let omega = twopi *. fc  in
    let sin_omega = sin omega  in
    let cos_omega = cos omega
    in (omega, sin_omega, cos_omega)

  let filter_helper_q fc q =
    let (omega, sin_omega, cos_omega) = filter_helper fc  in
    let alpha = sin_omega /. (2. *. q)
    in (omega, sin_omega, cos_omega, alpha)

  let filter_helper_bw fc octaves =
    let (o, s, c) = filter_helper fc  in
    let alpha = s *. sinh(log(2.)/.2. *. o/.s *. octaves)
    in (o, s, c, alpha)

  let filter2_normalize ((b0, b1, b2), (a0, a1, a2)) =
    let a0r = 1. /. a0  in
    ((b0 *. a0r, b1 *. a0r, b2 *. a0r),
              (a1 *. a0r, a2 *. a0r))

  let lowpass2 fc q =
    let (o, s, c, a) = filter_helper_q fc q  in
    let b1 = 1. -. c  in
    let b0 = b1 *. 0.5
    in filter2_normalize
      ((b0, b1, b0), (1. +. a, -.2. *. c, 1. -. a))

  let para_eq fc dbgain octaves =
    let (o, s, c, a) = filter_helper_bw fc octaves  in
    let biga = 10. ** (dbgain /. 40.)  in
    let ata = a *. biga  in
    let ada = a /. biga  in
    let b1 = -.2. *. c
    in filter2_normalize
      ((1. +. ata, b1, 1. -. ata), (1. +. ada, b1, 1. -. ada))

end  (* L *)


(******** ivec library ********)

module LIV = struct

  (* convert vec to ivec by repeating its last element *)
  let extend x =
    IV.join x (IV.const (V.last x))

  (* convert (ivec -> ivec) function to (vec -> vec) *)
  let finitize f =
    fun x -> IV.take (V.length x) (f (extend x))

        (** musical units **)

  let oct_to_hz oct = IV.map L.oct_to_hz oct
```

```
      (** phasors **)

  let phasor_wrap freq ph0 =
    let t = ref 0  in
    let ph = ref ph0  in
    let freqbuf = !freq#get_buf  in
    let f () =
      while (!ph < 1.0) do
        !freq#compute !t;
        ph := freqbuf.(0) +. !ph;
        incr t
      done;
      ph := !ph -. 1.;
      (float !t) -. !ph /. freqbuf.(0)    (* should be prior freq? *)

    in IV.iterate_niladic 0. f
   (* tabulate would break on seek, and iterate would make first elt some x0 *)

      (** oscillators **)

  (* 0 <= f < 0.5
     0 <= ph0 < 1 *)
  let osc_sine f ph0 =
    let twopif   = 2.0*.pi*.f
    and twopiph0 = 2.0*.pi*.ph0
    in let twoc = 2.0*.cos(twopif)
    and z1 = sin (twopiph0 -. twopif)
    and z2 = sin (twopiph0 -. 2.*.twopif)
    in IV.filter2_ic ((1., 0., 0.), (-.twoc, 1.)) (z1, z2) (IV.const 0.)

  let osc_cosine f = osc_sine f 0.25

  let noise_white () =
    IV.tabulate (fun _ -> (Random.float 2.0) -. 1.0)

  let impulse () =
    IV.iterate (fun _ -> 0.) 1.

      (** mixing **)

  (* mix: float ivec vec -> float ivec
   * add the ivecs together.
   *)
  let mix signals = V.foldl0 (IV.map2 0. (+.)) signals

  (* gain_mix: (float * (float ivec)) vec -> float ivec
   * multiply each signal by the corresponding gain, and sum.
   *)
  let gain_mix gainsigs =
    let (gains, signals) = V.unzip gainsigs
    in  V.foldl0 (IV.map2 0. (+.)) (V.map2 IV.scale gains signals)

      (** envelopes **)

  let relax k target x = (target -. k *. (target -. x))

(*** need EIV.chain

    (* attack and release are measured in number of samples to 99.3% *)
    let env_ar (attack: float) (release: float) (gate: bool event ivec) x0 =
      let kfrac = 0.9933  in
      let k_atk = kfrac ** (1. /. attack)  in    (* k_atk ^ attack = kfrac *)
      let k_rel = kfrac ** (1. /. release)  in
      let fchain xcurr gate_event =
        let f = if (E.v gate_event)
          then (relax k_atk 1.)
          else (relax k_rel 0.)  in
        IV.iterate f xcurr
      in EIV.chain fchain x0 gate

    (* stage times are measured in number of samples to 99.3% *)
    let env_adsr attack decay sustain release (gate: bool event ivec) x0 =
      let kfrac = 0.9933  in
      let k_atk = kfrac ** (1. /. attack)  in    (* k_atk ^ attack = kfrac *)
      let k_dec = kfrac ** (1. /. decay)  in
      let k_rel = kfrac ** (1. /. release)  in
      let fchain xcurr gate_event =
        if (E.v gate_event)
        then let att_dec xcurr attackP_ev =
          let f = if (E.v attackP_ev)
            then (relax k_atk 1.)
            else (relax k_dec sustain)
          in IV.iterate f xcurr
        in EV.chain att_dec xcurr [| true @@ 0; false @@ (int attack) |]
        else IV.iterate (relax k_rel 0.) xcurr
      in EIV.chain fchain x0 gate

    (* stages are ((target, k) @@ time). *)
    let env_multistage_k stage_evs x0 =
      let fchain xcurr stage_ev =
        let (target, k) = E.v stage_event  in
        IV.iterate (relax k target) xcurr
      in EIV.chain fchain x0 stage_evs

    let env_adsr' attack decay sustain release (gate: bool event ivec) x0 =
      let iattack = int attack  in
      let kfrac = 0.9933  in
      let atk = (1.,      kfrac ** (1. /. attack))  in   (* k^attack = kfrac *)
      let dec = (sustain, kfrac ** (1. /. decay))   in
      let rel = (0.,      kfrac ** (1. /. release)) in
      let f_slide2 e1 e2 =
        let t1 = E.t e1  in
        if (E.v e1)                        (* if attack *)
        then let dur = E.diff e1 e2 in
             if (dur < iattack)            (* if release cuts off attack *)
             then [| atk @@ t1 |]          (* then just attack *)
             else [| atk @@ t1; dec @@ (t1 + iattack) |]   (* else decay too *)
        else [| rel @@ t1 |]
      in let stages = IV.ravel (IV.map_slide2 f_slide2 gate)
      in env_multistage_k stages x0
***)

      (** filters **)

  let filter1 = IV.filter1             (* expose these here *)
  let filter2 = IV.filter2

  (* exponential relaxation filter (first-order lowpass)
   * t is time to relax halfway.
   * x0 is initial value of x.
   *)
  let lag t x0 x =
```

```
  let k = 0.5 ** (1. /. t)
  in IV.filter1v ((IV.const (1.-.k), IV.const 0.), IV.const (-.k)) 0. x0 x

(* time-varying exponential relaxation filter (first-order lowpass)
 * t is time to relax halfway.
 * x0 is initial value of x.
 *)
let lagv t x0 x =
  let minus_k = IV.map 0. (fun t -> -. (0.5 ** (1. /. t))) t  in
  let b0 = IV.map 0. (fun mk -> 1. +. mk) minus_k
  in IV.filter1v ((b0, IV.const 0.), minus_k) 0. x0 x

(* second-order IIR lowpass *)
let lowpass2 fc q =
  filter2 (L.lowpass2 fc q)

(* parametric band EQ *)
let para_eq fc dbgain octaves =
  filter2 (L.para_eq fc dbgain octaves)

(* XXX buggy!  several dB of peaking in the passband.  precision? *)
(* Butterworth lowpass filters of orders 4, 6, 8 *)
let buttergen qvec fc x =
  let f x q = lowpass2 fc q x
  in V.foldl f x qvec
let butter4 fc x =
  buttergen [| 0.60492333; 1.33722126 |] fc x
let butter6 fc x =
  buttergen [| 0.58338080; 0.75932572; 1.95302407 |] fc x
let butter8 fc x =
  buttergen [| 0.57622191; 0.66045510; 0.94276399; 2.57900101 |] fc x

      (** delay and reverb **)

(* first-order allpass with gain _g_ and delay length _len_ *)
(* we rearrange the block diagram to fit it into delay_rec's model *)
let allpass1 len g x =
  let gg = 1. -. (g *. g)  in
  let f x y0 = IV.map2 0. (fun x y0 -> gg*.x +. g*.y0) x y0  in
  let y0 = IV.delayz_rec2 len f x  in
  IV.map2 0. (-.) y0 (IV.scale g x)

(* delay is followed by fnest *)
let allpass1_nest len (fnest: float ivec -> float ivec) g x =
  let gg = 1. -. (g *. g)  in
  let f x y0 = IV.map2 0. (fun x y0 -> gg*.x +. g*.y0) x y0  in
  let y0 = IV.delayz_rec2 len (fun x z -> fnest (f x z)) x  in
  IV.map2 0. (-.) y0 (IV.scale g x)

let filtered_comb delay g1 g2 x =
  let filter = IV.filter1 ((1., 0.), -.g1)  in
  let feedback x y = IV.map2 0. (+.) x (IV.scale g2 (filter y))
  in IV.delayz_rec2 delay feedback x

let moorer_reverb fs gain x =
  let delay = [| 50.;  56.;  61.;  68.;  72.;  78.  |]  in       (* msec *)
  let g1 =    [| 0.46; 0.48; 0.50; 0.52; 0.53; 0.53 |]  in
  let fcomb delay g1 = filtered_comb
      (truncate (delay *. 0.001 *. fs)) g1 (gain *. (1. -. g1)) x  in
  let (combs: float ivec vec) = V.map2 fcomb delay g1  in
  let sumcombs = V.foldl0 (IV.map2 0. (+.)) combs
    in allpass1 (truncate (0.0061 *. fs)) 0.7 sumcombs

        (** I/O **)

  (* little-endian binary output *)
  let output_le_short h x =
    output_byte h (x land 0xff);
    output_byte h (x lsr 8)

  (* n.b. we're using ocaml's 31-bit ints here *)
  (* output_binary_int is big-endian on (little-endian) x86 hw. *)
  let output_le_int h x =
    output_le_short h (x land 0xffff);
    output_le_short h (x lsr 16)

  (* write mono 16-bit WAV.
   * x is clipped to [-1, 1].
   * (this writes a valid WAV file, but reversing it will not read
   * all valid WAV files, not by a long shot.)
   *)
  let write_wav name (fs: float) nframes (x: float ivec) =
    let fs = truncate fs  in           (* accept float just for convenience *)
    let blocklen = 256  in
    let h = open_out_bin name  in

    let s = String.create (2*blocklen)  in
    let f len xbuf =
      for i = 0 to (len-1) do
        let xi = xbuf.(i) in
        let clipped = if (xi < -1.) then -1. else
                      if (xi > 1.) then 1. else xi in
        let intx = 0xffff land (int_of_float (floor (clipped*.32767. +. 0.5)))
        in s.[2*i]     <- char_of_int (intx land 0xff);
           s.[2*i + 1] <- char_of_int (intx lsr 8)
      done;
      output h s 0 (2*len)

    in (output_string h "RIFF";
        output_le_int   h (4+24+8 + 2*nframes);  (* riff+fmt+data bytes *)
        output_string   h "WAVE";
        output_string   h "fmt ";         (* fmt chunk *)
        output_le_int   h 16;             (* remaining bytes of fmt *)
        output_le_short h 1;              (* magic? *)
        output_le_short h 1;              (* one channel *)
        output_le_int   h fs;             (* frames per second *)
        output_le_int   h (fs*2);         (* bytes per second *)
        output_le_short h 2;              (* bytes per frame *)
        output_le_short h 16;             (* bits per sample *)
        output_string   h "data";
        output_le_int   h (nframes*2);    (* bytes of data *)

        IV.block_apply_upto nframes blocklen f x;

        close_out h
       )

end  (* LIV *)


(******** vector library ********)
```

```
module LV = struct

        (** math **)

  let dot x y =
    V.foldl (+.) 0.0 (V.map2 ( *.) x y)

        (** conversions between units **)

  let oct_to_hz oct = V.map L.oct_to_hz oct

        (** window functions **)

  let window f n =
    let nf = float n
    in let x = V.count_by (-1.0) (2.0 /. (nf -. 1.0)) n
    in V.map f x

  let cyclic_window f n =
    let nf = float n
    in let x = V.count_by (-1.0) (2.0 /. nf) n
    in V.map f x

  let raised_cosine = window L.f_raised_cosine
  let cyclic_raised_cosine = cyclic_window L.f_raised_cosine
  let hamming   = window L.f_hamming        (* -40 dB *)
  let blackman3 = window L.f_blackman3      (* -58 dB *)
  let blackman4 = window L.f_blackman4      (* -92 dB *)

        (** interpolation **)

  (* zero-order (floor) resampling. *)
  let resample0 freq phase0 x =
    let t = V.count_by_upto phase0 freq (float ((V.length x) - 1))
    in V.map (fun t -> x.(int t)) t

  (* first-order resampling. *)
  let resample_linear freq phase0 x =
    let t = V.count_by_upto phase0 freq (float ((V.length x) - 1))
    in V.map (fun t -> L.interp1 t x) t

  (* zero-order (floor) resampling, controlled by _freq_, to _len_. *)
  let resample0_to len freq phase0 x =
    let xlen = V.length x  in
    let t = V.count_by phase0 freq len  in
    let f t =
      let i = int t
      in if (i >= xlen) then 0. else x.(i)
    in V.map f t

        (** phasors **)

  (* requires 0 <= f; f < 0.5 recommended *)
  let phasor n f ph0 =
    let step ph = mod_float (ph +. f) 1.0
    in V.iterate step ph0 n

  (* requires 0 <= f.(i); f.(i) < 0.5 recommended *)
  let phasor_v f ph0 =
    let step ph f = (mod_float (ph +. f) 1.0, ph)
    in V.sidescan step ph0 f

  (* requires 0 <= f < 1 *)
  let phasor_wrap tmax f ph0 =          (* XXX this code is crap *)
    let i = ref 0
    and ph = ref ph0
    and tv = ref [||]
    in (
    while (!i < tmax) do
      let fi = f.(!i) in
      (
       if (!ph >= 1.0) then (
         fdec 1.0 ph;
         tv := V.join !tv (V.from_scalar ((float !i) -. (!ph /. fi)))  (*?!*)
        ) else ();
       incr i;
       finc fi ph;
      );
    done;
    !tv
   )

        (** tables **)

  (* interpolated table lookup according to a phase vector *)
  let tablei tab ph =
    let ntab = L.subunity *. (float ((V.length tab) - 1))
    in V.map (fun ph -> L.interp1 (ntab *. ph) tab) ph

(*** is this supposed to be faster?
    (* phase input is [0, 1] *)
    let tablei' tab ph =
      let ntab = subunity *. (float ((V.length tab) - 1))  in
      let out = V.copy ph  in
      let n = V.length ph  in
      for i = 0 to n-1 do
        out.(i) <- interp1 (ntab *. ph.(i)) tab
      done;
      out
***)

        (** oscs **)

  let table_of_sine_harmonics amps n =
    let nh = V.length amps  in
    let k = twopi /. (float n)  in
    let f t =
      let base_phase = (float t) *. k  in
      let harmonic i = amps.(i-1) *. (sin ((float i) *. base_phase))  in
      let harmonics = V.map harmonic (V.int_count 1 nh)
      in V.foldl (+.) 0. harmonics
    in V.tabulate f n

  let table_of_sine n =
    table_of_sine_harmonics [|1.|] n

  (* interpolating table oscillator *)
  (* 0 <= f < 0.5, 0 <= ph0 < 1 *)
  (* XXX is this glitch-free? *)
  let osci tab n f ph0 =
    tablei tab (phasor n f ph0)
```

```
(* variable-frequency interpolating table oscillator *)
(* 0 <= f.(i) < 0.5; 0 <= ph0 < 1 *)
let osci_v tab f ph0 =
  tablei tab (phasor_v f ph0)

(* 0 <= f < 0.5
   0 <= ph0 < 1 *)
let osc_sine n f ph0 =
  let twopif   = 2.0*.pi*.f
  and twopiph0 = 2.0*.pi*.ph0
  in let twoc = 2.0*.cos(twopif)
  and x0 = sin twopiph0
  and x1 = sin (twopiph0 -. twopif)
  and x = V.const 0.0 n            (* QQQ uninit *)
  in let rec loop i x0 x1 =
    x.(i) <- x0;
    if (i < n-1) then (loop (i+1) (twoc*.x0 -. x1) x0) else ()
  in (loop 0 x0 x1;
      x)

(* 0 <= f < 0.5 *)
let osc_sine_slow n f ph0 =
  let t = V.count_by ph0 f n
  in V.map (fun t -> sin (2.0*.pi*.t)) t

(* 0 <= f < 0.5 *)
let osc_additive n freq_amp_list =
  let bank = V.map (fun (f,a) -> V.scale a (osc_sine n f 0.)) freq_amp_list
  in V.foldl (fun sum x -> V.map2 (+.) sum x) (V.const 0. n) bank

let noise_white n =
  V.tabulate (fun _ -> (Random.float 2.0) -. 1.0) n

      (** envelopes **)

let env_lin_ar dur attack release =
  let ia = int attack and ir = int release
  in V.drop_end 1 (EV.pwl_list 0. [(ia, 1.); (dur - ir, 1.); (dur, 0.)])

let declick x =
  let ramplen = 30.
  in V.map2 ( *.) x (env_lin_ar (V.length x) ramplen ramplen)

      (** filters **)

let filter1 coeffs =
  LIV.finitize (IV.filter1 coeffs)

let filter2 coeffs =
  LIV.finitize (IV.filter2 coeffs)

let lowpass2 fc q =
  filter2 (L.lowpass2 fc q)

let para_eq fc dbgain octaves =
    filter2 (L.para_eq fc dbgain octaves)

        (** I/O **)

  (* write mono 16-bit WAV.
   * x is clipped to [-1, 1].
   *)
  let write_wav name (fs: float) (x: float vec) =
    LIV.write_wav name fs (V.length x) (LIV.extend x)

end  (* LV *)



(****************************** testing *****************************)

let test () =
  let env_lin_ar = LV.env_lin_ar 8 2. 4.  in

  let _ = assert (env_lin_ar = [| 0.; 0.5; 1.; 1.; 1.; 0.75; 0.5; 0.25 |])  in

  (prerr_string "L ok?\n"; flush stderr; true)

let _ = assert (test ())                (* a.out.opt strips it all away *)


(* LIV.allpass1(_nest), LV.lowpass2, LV.para_eq check out okay *)
(*---
let impulse = IV.iterate (fun _ -> 0.) 1.
let ap1 = LIV.allpass1 2 0.5 impulse
let _ = eprintf "--------\n"
let ap2 = LIV.allpass1_nest 3 (LIV.allpass1 2 0.5) 0.5 impulse

let n = 9
(* let _ = IV.print_floats n 4 ap2 *)

(* let ap2 = LIV.allpass1_nest 3 (LIV.allpass1 2 0.5) 0.5 impulse
 *
 * 0   -0.5   (0.75 fed into nest)
 * 1
 * 2
 * 3          -0.375    (-0.1875 fed into nest)
 * 4
 * 5          0.5625               (0.28125 fed into nest)
 * 6                    0.09375
 * 7          0.28125
 * 8                    -0.140625  -0.140625
 *)

let vimp = V.iterate (fun _ -> 0.) 1. 64
let lowpass = LV.lowpass2 0.125 2. vimp
let peak = LV.para_eq 0.125 6. 0.5 vimp
---*)
```

D.10 vecl.ml

```
(* polymorphic vector, w/ length bound on operations.  for use by IV.
 * all(?) operations (for which it makes sense) can be done in place, out == x.
 *
 *)

open Globals
open Util
open Vec

(*
type 'a vec = 'a array
*)

module VL = struct

(** create *)

  let tabulate_offset_i out len f ofs =
    for i = 0 to (len-1) do  out.(i) <- f (i+ofs)  done

  let tabulate out len f =
    for i = 0 to (len-1) do  out.(i) <- f i        done

  let iterate_and_spill out len f x0 =            (* for use by IVR *)
    let xi = ref x0 in
    (for i = 0 to (len-1) do
      (out.(i) <- !xi;
       xi := f !xi)
     done;
     !xi)

  let iterate_niladic out len f =
    for i = 0 to len-1 do
      out.(i) <- f ()
    done

(** map *)

  let map out len f x =
    for i = 0 to len-1 do
      out.(i) <- f x.(i) done

  let mapi_offset_i out len f iofs x =
    for i = 0 to len-1 do
      out.(i) <- f (i+iofs) x.(i) done

  let map2 out len f x y =
    for i = 0 to len-1 do
      out.(i) <- f x.(i) y.(i) done

  let add2 out len x y =
    for i = 0 to len-1 do
      out.(i) <- x.(i) +. y.(i) done

  let map3 out len f x y z =
    for i = 0 to len-1 do
      out.(i) <- f x.(i) y.(i) z.(i) done

  let map4i_offset_i out len f iofs
      x1 x2 x3 x4 =
    for i = 0 to len-1 do
      out.(i) <- f (i+iofs) x1.(i) x2.(i) x3.(i) x4.(i) done

  let map_slide2 out len f oldx x =
    let ox = ref oldx in
    for i = 0 to len-1 do
      out.(i) <- f !ox x.(i);
      ox := x.(i)
    done;
    !ox

(** scan *)

  let sidescan_with_spill out len f a0 b =
    let a = ref a0 in
    let tabf i =
      let (a', ci) = f !a b.(i) in (a := a'; ci)
    in (tabulate out len tabf;
        !a)

  let scan_with_spill out len f a0 x =
    let a = ref a0 in
    (for i = 0 to len-1 do
      (out.(i) <- !a;
       a := f !a x.(i))
     done;
     !a
    )
end
```

D.11 util.ml

```
open Printf

        (*** refs ***)

let finc dx rx =  rx := !rx +. dx
let fdec dx rx =  rx := !rx -. dx

        (*** integer ***)

(* 0 <= result < m *)
let mod_floor x m =
  let r = x mod m
  in if (r < 0) then (r + m) else r

        (*** floating point ***)

let pi = 4. *. (atan 1.)
let twopi = 2. *. pi

let int = truncate
let float = float_of_int

let sq x = x *. x

let fuzz = (ldexp 1. (-32))
let fuzzeq x y =
  (abs_float (x -. y)) < fuzz

(* x mod m, into [0, m) interval.  (whereas mod_float can be negative) *)
let mod_float_nonneg x m =
  let raw = mod_float x m
  in if (raw < 0.) then (raw +. m) else raw

(* integer part is rounded down *)
let modf_floor x =
  let f = floor x in (f, x-.f)

let frac x = x -. (floor x)

let modfi x =
  let (fracpart, intpart) = modf x
  in (fracpart, truncate intpart)

(* 0 <= result < m *)
let mod_float_floor x m =
  let r = mod_float x m
  in if (r < 0.0) then (r +. m) else r

(* integers get decremented *)
let floor_intdown x =
  let fx = floor x
  in if (x = fx) then (x -. 1.0) else x

        (*** statistics ***)

let sum x = Array.fold_left (+.) 0. x

let mean x = (sum x) /. (float (Array.length x))

(* ought to make this estimate pop. stddev instead *)
let mean_and_stddev x =
  let xmean = mean x  in
  let var = sum (Array.map (fun x -> sq (x -. xmean)) x)  in
  let sample_stddev = sqrt (var /. (float (Array.length x)))
  in (xmean, sample_stddev)

        (*** string ***)

(* first index from _from_ for which _f_ is true, or Not_found *)
let string_find_first_from f from s =
  let n = String.length s  in
  let i = ref from  in
  while (!i < n && (not (f s.[!i]))) do
    incr i
  done;
  if (!i >= n) then (raise Not_found) else (!i)

(* finds first char of _s_ in the string _chars_, or Not_found *)
let index_in_from chars from s =
  string_find_first_from (fun c -> String.contains chars c) from s

(* finds first char of _s_ not in the string _chars_, or Not_found *)
let index_not_in_from chars from s =
  string_find_first_from (fun c -> not (String.contains chars c)) from s

        (*** i/o ***)

(* Printf.eprintf doesn't flush stderr?  morons, orangutans, radiators.  *)
let eprintf stuff =
  let gunk = Printf.eprintf stuff
  in (flush stderr; gunk)

let print_nl () = (print_string "\n"; print_newline)
(* print_newline does not go so far as to print a newline.  flushes? *)

let output_binary_float out f =         (* dear god, forgive me *)
  let i64_255 = Int64.of_int 255  in
  let low_byte i = Int64.to_int (Int64.logand i i64_255)  in
  let bits = ref (Int64.bits_of_float f)  in
  for i = 0 to 7 do
    output_byte out (low_byte !bits);
    bits := Int64.shift_right_logical !bits 8
  done

(* read as a list of strings *)
let getlines_from_channel ch =
  let rec grotty_helper accum =
    try grotty_helper ((input_line ch) :: accum)
    with End_of_file -> accum
  in List.rev (grotty_helper [])

let getlines_from filename =
  getlines_from_channel (open_in filename)

let scanf_floats () =
  getlines_from_channel stdin
```

```
        (*** fnal ***)

let id x = x

(* lifts a 2-op curried fn to work on options,
 * propagating None values.
 *)
let opt_lift2 f =
  fun x y ->
    (match (x, y) with
      (Some x', Some y') -> Some (f x' y')
    | _ -> None
    )

        (*** tuples ***)

let pair_of x = (x, x)
let triple_of x = (x, x, x)

let map_pair f (a, b) = (f a, f b)
let map_triple f (a, b, c) = (f a, f b, f c)

let one_and_one a b = (a, b)
let two_and_one (a, b) c = (a, b, c)
let one_and_two a (b, c) = (a, b, c)
let one_and_one_and_one a b c = (a, b, c)

        (*** deep copy ***)

let deepcopy x =
  let xpickle = Marshal.to_string x [Marshal.Closures]
  in Marshal.from_string xpickle 0

        (*** timing ***)

(*---
let utime f =
  let start = (Unix.times ()).Unix.tms_utime  in      (* look Ma no parens! *)
  let _ = f ()  in
  let stop =  (Unix.times ()).Unix.tms_utime
  in stop -. start

let utime_mean n f =
  let x = Array.map utime (Array.make n f)
  in mean_and_stddev x
---*)

        (*** testing ***)

let test (tag, correct_value, questionable_value) =
  if (questionable_value <> correct_value)
  then (eprintf "%s\n" ("test failed: " ^ tag); false)
  else true

        (*** debugging ***)

let myassert tag flag =
  if flag then ()
  else (eprintf "assertion failed: %s\n" tag;
        raise Exit)


(****************************** testing *****************************)

let test () =
  let _ = assert (string_find_first_from (fun c -> c = 'x') 2 "x0xb0x" = 2) in
  let _ = assert (string_find_first_from (fun c -> c = 'x') 3 "x0xb0x" = 5) in
  let _ = assert (index_not_in_from "ab" 0 "abracadabra" = 2)  in
  let _ = assert (index_not_in_from "ab" 2 "abracadabra" = 2)  in

  (prerr_string "some of util ok?\n"; flush stderr; true)

let _ = assert (test ())                (* a.out.opt strips it all away *)
```

Bibliography

André, C. (1996, July). Representation and analysis of reactive behaviors. In *Proc. CESA'96*, Lille, France.

Ashcroft and Wadge (1977, July). Lucid, a nonprocedural language with iteration. *CACM 20*(7), 519.

Berry, G. and G. Gonthier (1992). The Esterel synchronous programming language. *Science Of Computer Programming 19*(2), 87–152.

Blelloch, G. E. (1995). NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, CMU.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). *Pattern-Oriented Software Architecture*. Wiley and Sons Ltd.

Cook, P. R. and G. P. Scavone (1995). The Synthesis ToolKit (STK). In *Proc. International Computer Music Conference*. International Computer Music Association.

Cycling '74 (2001). MSP product description. Via WWW.
`http://www.cycling74.com/products/msp.html`.

Dannenberg, McAvinney, and Rubine (1986, Winter). Arctic: a functional approach to real-time control. *Computer Music Journal 10*(4), 67–78.

Dannenberg, R. (2001, September). Personal communication.

Dannenberg, R. and E. Brandt (1996, August). A flexible real-time software synthesis system. In *Proc. International Computer Music Conference*, pp. 270–273. International Computer Music Association.

Dannenberg, R. B. (1997a, Fall). Abstract time warping of compound events and signals. *Computer Music Journal 21*(3).

Dannenberg, R. B. (1997b, Fall). The implementation of Nyquist, a sound synthesis language. *Computer Music Journal 21*(3), 71–82.

Dannenberg, R. B. (1997c, Fall). Machine Tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal 21*(3), 50–60.

Dannenberg, R. B., P. Desain, and H. Honing (1997). *Musical Signal Processing*, Chapter 8. Swets and Zeitlinger.

Dolson, M. (1987). The phase vocoder: A tutorial. *Computer Music Journal 10*(4), 14–27.

Elliott, C. (1998). Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*.

Elliott, C. and P. Hudak (1997). Functional reactive animation. In *Proceedings of ICFP '97*, pp. 263.

Freericks, M. (1996). *Design and Implementation of a Partial Evaluation-Based Compiler for an Asynchronous Realtime Programming Language*. Ph. D. thesis, Bielefeld University.

Gardner, W. G. (1992, September). *The Virtual Acoustic Room*. Massachusetts Institute of Technology.

Gauthier, T., P. LeGuernic, and L. Besnard (1987). Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd Conf. on Functional Programming Languages and Computer Architecture, LNCS 274*. Springer Verlag.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer.

Halbwachs, N., P. Caspi, and D. Pilaud (1991, September). The synchronous dataflow programming language Lustre. In *Another Look at Real Time Programming*, Proc. IEEE, Special Issue.

Harel, D. (1987). Statecharts: a visual approach to complex systems. *Science of Computer Programming 8*(3), 231–274.

Harel, D. and A. Pneuli (1985). *Logic and Models of Concurrent Systems*, Chapter On the development of reactive systems, pp. 477–498. Springer Verlag.

Hilfinger, P. (1985). A high-level language and silicon compiler for digital signal processing. In *Proceedings of the Custom Integrated Circuits Conference*, pp. 213–216. IEEE Computer Society Press.

Hudak, P., T. Makucevich, S. Gadde, and B. Whong (1996). Haskore music notation—an algebra of music. *Journal of Functional Programming 6*(3).

Intel (2002). Intel signal processing library. Via WWW.
`http://developer.intel.com/software/products/perflib/spl/`.

Järvi, J. and G. Powell (2000, November). The Lambda Library. Technical Report 378, Turku Centre for Computer Science.

Jay, C. B. and P. A. Steckler (1998). The functional imperative: shape! In C. Hankin (Ed.), *Proc. 7th European Symposium on Programming*, Volume 1381 of *Lecture Notes in Computer Science*, pp. 139–153. Springer Verlag.

Kopec, G. E. (1985, August). The signal representation language SRL. *IEEE Trans. Acoust. Speech Signal Proc. 33*(4), 921–931.

Lansky, P. (1990). *CMix Release Notes and Manuals*. Department of Music, Princeton University.

Lansky, P. and K. Steiglitz (1995). Ein: A signal processing scratchpad. *Computer Music Journal 19*(3).

LeGuernic, P., A. Benveniste, P. Bournai, and T. Gautier (1985). Signal: a data-flow oriented language for signal processing. Technical Report RR 378, INRIA.

Leroy, X. (2001). Objective Caml. Online at `http://caml.inria.fr/ocaml/`.

Letz, S., Y. Orlarey, and D. Fober (1995). Real time functional languages. In *Proc. ICMC95*, pp. 549–552. International Computer Music Association.

Maraninchi, F. (1991). The Argos language: graphical representation of automata and description of reactive systems. In *Proc. International Conference on Visual Languages*, Kobe, Japan.

Mathews, M. V. (1969). *The Technology of Computer Music*. MIT Press.

McCartney, J. (1996). SuperCollider: a new real time synthesis language. In *Proc. International Computer Music Conference*. International Computer Music Association.

McGraw, J. R., S. Skedzielewski, S. Allan, R. Oldehoeft, J. R. W. Glauert, C. Kirkham, W. Boyce, and R. Thomas (1985). Sisal: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory.

Moorer, J. A. (1979). About this reverberation business. *Computer Music Journal 3*(2), 13–28.

Mullin, L. M. R. (1991). *Arrays, Functional Languages and Parallel Systems*, Chapter The Ψ-function: a basis for FFP with arrays, pp. 185–201. Kluwer.

Nico, P., J. Feo, and R. K. Yates (1997, June). Implicitly parallel real-time signal processing. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, Las Vegas.

Oppenheim, A. V. and R. W. Schafer (1975). *Digital Signal Processing*. Prentice-Hall.

Orlarey, Y. and S. Letz (2002, February). Personal communication.

Puckette, M. (1988). The patcher. In *Proc. International Computer Music Conference*, pp. 420–429. International Computer Music Association.

Puckette, M. (1991). FTS: a real-time monitor for multiprocessor music synthesis. *Computer Music Journal 15*(3), 58–67.

Puckette, M. (1996). Pure Data. In *Proc. International Computer Music Conference*, San Francisco, pp. 269–272. International Computer Music Association.

Rodet, X. (1984). Time-domain formant wave-function synthesis. *Computer Music Journal 8*(3), 9–14.

Scheirer, E. D. (1998). SAOL: The MPEG-4 structured audio orchestra language. In *Proc. International Computer Music Conference*, pp. 432–438. International Computer Music Association.

Scholz, S.-B. (1996a). On programming scientific applications in SAC — a functional language extended by a subsystem for high-level array operations. In *Proc. IFL'96, LNCS 1268*, pp. 85–104. Springer-Verlag.

Scholz, S.-B. (1996b). *Single Assignment C — Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. Ph. D. thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel.

Scholz, S.-B. (1998). On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proc. APL'98*, pp. 40–45. ACM.

Sprenger, S. (1999). Pitch scaling using the Fourier Transform. Online at `http://www.dspdimension.com/html/pscalestft.html`.

Steiglitz, K. (1996). *A DSP Primer.* Addison-Wesley.

The MathWorks, Inc. (1997). *Matlab language reference manual* (Version 5 ed.). The MathWorks, Inc.

Veldhuizen, T. (1995a, June). Expression templates. *C++ Report 7*(5), 26–31.

Veldhuizen, T. (1995b, May). Using C++ template metaprograms. *C++ Report 7*(4), 36–43.

Vercoe, B. (1993). *Csound — a manual for the audio processing system and supporting programs with tutorials.* Media Lab, MIT.

Vercoe, B. L., W. G. Gardner, and E. D. Scheirer (1998). Structured audio: Creation, transmission, and rendering of parametric sound representations. *Proc. IEEE 86*(5), 922–940.

Xenakis, I. (1971). *Formalized Music.* Indiana University Press.