Temporal type constructors for computer music programming

Eli Brandt

June 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.

Chapter 1

Summary

My dissertation proposes 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.

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.

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`.

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`	(various types)

`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.

1.3 Applications written with Chronic

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

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. We do not have an example of a real composition scored in Chronic, but the examples give some confidence that it could be.

1.3.2 FOF synthesis

Chronic's style of programming applies beautifully to granular synthesis, such as the FOF synthesis algorithm (Rodet 1984). FOF is a technique for synthesizing a pitched signal with a spectral peak, a formant, as found in the human 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.

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.

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.

1.3.4 A frequency-domain pitch shifter

To pitch-shift a signal is to transpose its musical pitch without changing its length. The simplest kind of pitch shifter that doesn't fail utterly on polyphonic signals is 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. It is very roughly half as many lines, but then it draws on a richer library and has fewer comments. 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 constructed types. Then 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 applying different spectral manipulation within the the STFT analysis/synthesis framework.

Inadequate type systems discourage this kind of aggressive factoring into reusable subcomponents, because they can't 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 can't 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.

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 the dissertation gives reasons to go beyond these representations.

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 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 to note 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 an `['a] ivec_dat`. 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.

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`.

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 `ivecs` 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.

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.

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.

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.

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.

A small benchmark 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.

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, we build an `IVH` module on top pf `IV`, containing `IVH.peek`, and use it to write a non-monotonic `IV.peekiv_basis`.

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.

This is awkward. 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`.

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.

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.

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 the full document.

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.

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), 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.

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.

1.8 Conclusions

1.8.1 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. 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.

1.8.2 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.

1.8.3 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 were necessary to the development of this research.

1.8.4 Applications

1.8.4.1 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, which come from a lack of compile-time power. C++ templates also give us vector comprehensions (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.

1.8.4.2 Problem domains

What are some examples of algorithms involving non-trivial temporal types, where Chronic can be brought to bear?

- 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. 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.

1.8.5 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, the Gardner reverberator example 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 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.

1.8.6 Successes

The dissertation's chapter of example applications 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.

1.8.7 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.

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*.

Bibliography

Dannenberg, R. (2001, September). Personal communication.

Gardner, W. G. (1992, September). *The Virtual Acoustic Room.* Massachusetts Institute of Technology.

Järvi, J. and G. Powell (2000, November). The Lambda Library. Technical Report 378, Turku Centre for Computer Science.

Moorer, J. A. (1979). About this reverberation business. *Computer Music Journal 3*(2), 13–28.

Rodet, X. (1984). Time-domain formant wave-function synthesis. *Computer Music Journal 8*(3), 9–14.

Sprenger, S. (1999). Pitch scaling using the Fourier Transform. Online at `http://www.dspdimension.com/html/pscalestft.html`.

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.