Implementing Temporal Type Constructors for Music Programming

Eli Brandt
School of Computer Science, Carnegie Mellon University
email: eli@cs.cmu.edu

Abstract

Applied to musical DSP programming, temporal type constructors enable high-level expression of sample-level algorithms. However, realizing this promise is not trivial. This work explains why, and describes a practical realization. This realization presents a functional interface onto a back end of interlinked objects. Its speed is a factor of two to ten off that of C++.

1 Introduction

An ideal language is both general and expressive: all algorithms can be represented, and represented cleanly. Languages used for computer music tend to fail at one goal or the other: for example, C expresses a synthesis "patch" less cleanly than Nyquist (Dannenberg 1997), but an LPC processor for Nyquist can only be written in C. Temporal type constructors help a language succeed at both. We describe how a domain-specific language with temporal type constructors, called *Chronic*, is embedded within the functional language O'Caml (Leroy 2001). Chronic is quite general; operations need not preserve length, nor even be causal.

A type constructor builds 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, we'll see, can cross-fertilize to build a great variety.

2 Temporal type constructors

This section is a brief review of work described in an earlier paper (Brandt 2000).

A *temporal* type constructor is one that introduces a relation to a one-dimensional axis, which we call time. In truth, it does not have to be time, but time is the most common kind of axis in computer music.

I have proposed using three temporal type constructors: event, vec, and ivec. An "α event" is an α at a single specified time, where α can be any type. An "α vec" is some number of αs in sequence, and an "α ivec" is an infinite number. This work uses machine integers for the type Time.

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.

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 A cornucopia of types

It is fascinating to see the richness of temporal types arising from these three type constructors. See the earlier paper for more examples, including ones beyond music.

Multi-channel audio. If Sample is the type of an audio sample, then Sample ivec vec is a vec of ivecs of Samples; the length of the vector is the number of channels. The transposed Sample vec ivec would be convenient for certain manipulations.

A non-MIDI score. (Pitch vec $\times$ Pressure vec) event ivec is a score suitable for instruments that are not like keyboards.

These note events can represent continuous control gestures. Each one encodes its own duration (as the length of the vecs), and each one has a timestamp. These times might, depending on the way the score is generated and used, be non-decreasing.

Short-time spectrum data. Complex vec ivec represents data from a short-time Fourier transform.

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

Grains of sound. Sample vec event ivec is a stream of time-stamped segments of audio, as used in granular synthesis.

This is an example of the 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 (Rodet 1984) and formant-corrected pitch shifting, involve arranging small grains of sound.

2.2 Models of time

There are two general approaches to temporal data. One, made possible by 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, but one lacks the inverse operation, drop:

$$\begin{aligned} \mathsf{delay}\ 2\ [1\ 4\ 9\ 16\ \ldots] &= [0\ 0\ 1\ 4\ 9\ \ldots] \\ \mathsf{drop}\ 2\ [0\ 0\ 1\ 4\ 9\ 16\ \ldots] &= [1\ 4\ 9\ \ldots] \end{aligned}$$

In Csound (Vercoe 1993), 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—negated is a float ivec—and is explicit about the looping:

```
negated = IV.map (~-) 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. The programmer cannot choose the temporal structure of the program to fit that of the algorithm.

3 Chronic's interface

Chronic consists of a core and a standard library. Each of its core modules deals with one kind of temporal type, while the standard library provides computer-music primitives. Chronic is embedded within O'Caml, so code written with Chronic is O'Caml code.

In the core library, V has the usual vector functions (map, fold, scan, filter, index, take subvectors), IV has their infinite counterparts, and E assembles and disassembles events. EV and EIV deal specifically with α event vec and α event ivec; they have functions like "piecewise constant" (convert an α event ivec to an α ivec through zero-order hold) and "vertical fold" (fold time-aligned samples in an α vec event ivec).

The standard library, L, provides oscillators, envelope generators, filters, and other common tools. Many of them have finite and infinite variants, accessed as L.V and L.IV.

Earlier work (Brandt 2000) gives code for FOF synthesis in a direct predecessor to Chronic, and explains how temporal type constructors let the code's structure directly express our idea of the algorithm: create grains, place each one in time (forming a float vec event ivec), and mix them all down.

4 Chronic's implementation

Dealing with events and vectors is elementary in principle, though at times (especially in EV and EIV) it becomes rather involved. This paper will focus on ivecs, and on how the use of a *phase distinction* allows a clean functional front end to a back end of imperatively-updated objects.

4.1 It's not trivial

One's first thought is that representing ivecs is trivial in a lazy functional language: an ivec is an infinite list. This has the correct semantics, but its asymptotic space use is wrong. A program to negate each sample should run indefinitely in $O(1)$ space, whereas this representation takes $O(n)$ space (and eventually runs out of memory). An ivec ought to discard past data that will not be needed again.

The simplest way to fix this is to give an ivec two parts: an infinite list representing the future, and a "now" value telling when the future starts from. This gives the correct asymptotic behavior, but the list manipulation is still costly. Note also that fan-out of a value to multiple readers causes multiple computation of the value.

To take advantage of array processing's greater speed, we might make an ivec a list of arrays. This allows block-based computation with variable-length blocks, but the length of each block is controlled by the creator. This control must be given to the receiver—ivec manipulation may warp time unpredictably, and in the end, the digital-to-analog converter needs to be able to pull a constant data rate downstream.

To give this control to the receiver, an ivec becomes a closure. Specifically, it becomes convenient to make it an O'Caml object.

4.2 The phase distinction

The phase distinction is the idea that ivecs are handled in two phases, *building* and *computation*. The user-visible API glosses over the distinction, but it is worth noting because the two phases have quite disparate implementations. In this way, an API in a functional style is layered over an implementation in a different style, which would be awkward to work with directly—namely, a dataflow DAG of interlinked objects making heavy use of imperative update.

A small example will give the general idea:

```
  (* build a dataflow graph *)
let threes = IV.const 3
```

```
let sixes = IV.map (fun x -> x*2) threes
  (* ask it to compute *)
let _ = IV.print_int 10 4 sixes
```

IV.const returns an int ivec; IV.map takes this int ivec and returns another. The building phase is now complete. What's been built, as we'll see, is a dataflow graph of two nodes, with sixes holding a reference to threes.

IV.print_int causes computation. It asks sixes to compute successive blocks of samples (of lengths 4, 4, and 2). Each time, sixes asks threes to compute a corresponding block, and operates on 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 only becomes tolerable when it is amortized over a block of samples.

4.3 The computation phase

An α ivec is implemented as a reference to an object. This object is an α ivec_dat, or some subclassed object that through subtype coercion can be viewed as an α ivec_dat. An α ivec_dat has three publically-visible methods:

```
get_buf: unit -> 'a array
compute: time -> unit
seek:    time -> unit
```

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, to support fan-out to multiple readers, who must all advance time in lockstep. Subclasses do not override compute. They override the pure-abstract compute_hook, which compute calls as needed after doing temporal bookkeeping.

The get_buf method returns the output buffer where compute places its results. A global maxblocklen determines the size of the output buffer.

The seek method advances time without generating output, so it is permitted to step more than maxblocklen ahead of the current time. It may also, for some ivec_dat subclasses which override it, be cheaper than compute.

4.4 Subclassing ivec_dat

I will simply leave out certain technical details that would take too much explanation to render non-confusing; the gist here is accurate, but the code will not compile as is.

IV.map depends on a class called map_dat. This class inherits from ivec_dat and provides a compute_hook, which compute will then call. 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.

```
class ['a, 'b] map_dat (f: 'a->'b) (xin: 'a ivec) =
object inherit ['b] ivec_dat
  val f = f
  val x = xin
  val xbuf = !xin#get_buf

  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
```

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

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

The building phase. Associated with map_dat is a builder function, map:

```
let map (f: 'a -> 'b) x =
  ref ((new map_dat 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—an 'a ivec. Every builder function, through similar subtype coercion, returns this same type.

4.5 Putting the pieces together

Now we have a better understanding of the example:

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

IV.map is the builder function that creates a new map_dat, and gives it its f and its x. Its x is threes, a reference to a const_dat. During computation, the call sequence is:

```
IV.print_int 10 4     (* blocksize 4, up to 10 *)
   !sixes#compute 4
      !sixes#compute_hook 4 4
         !threes#compute 4
            !threes#compute_hook 4 4
         VL.map buf 4 f !threes#get_buf
   !sixes#compute 8
      !sixes#compute_hook 8 4
         !threes#compute 8
            !threes#compute_hook 8 4
         VL.map buf 8 f !threes#get_buf
   ...
```

4.6 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. For example, a 1-cycle:

```
let rec nats = IV.map succ (IV.delay 1 [|0|] nats)
(* [|0|] is the basis, so nats is 0, 1, 2, ... *)
```

This is not legal, as values of type 'a ivec cannot 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 not as an argument of type ($\alpha \to \alpha$) but as one of type (α ivec $\to \alpha$ ivec). The function for recursive constant-length delay, delay_rec, has arguments (delay_length: time), (basis: α vec), and (f: α ivec $\to \alpha$ ivec), and has output (y: α ivec). It satisfies $y = f\ (\mathsf{delay}\, y)$. With it we write

```
let nats = IV.delay_rec 1 [|0|] (IV.map succ)
```

This $y = f_1\ (\mathsf{delay}\, y)$ may not seem sufficient. For example, a repeating echo y of an input x takes the form $y = f_2\ x\ (\mathsf{delay}\, y)$. But through partial application, $(f_2\ x)$ constitutes f_1.

4.7 Missing pieces

Chronic is not ready for production use. A crucial missing piece is that the O'Caml compiler does not presently specialize in order to inline: map (+) involves a function call on each element. This takes far longer than the addition itself. If Chronic is to work with this compiler, it needs to perform specialization itself. I suggest using the ocamlp4 preprocessor to generate code from vector comprehensions.

A notion of control-rate signals is another high-priority enhancement. Furthermore, the builder functions could dispatch based on rate tags, for transparent overloading.

Another interesting possibility is entirely reimplementing Chronic in terms of C++ templates (Veldhuizen 1995b) (Veldhuizen 1995a).

5 Measurements

Table 1 shows the results of two timing benchmarks. The first benchmark times the operation of mixing two double-precision signals, implemented in Chronic and in C++. Both use a block size of 256 and compute 2^{24} samples.

The second benchmark times a Chronic implementation of FOF synthesis against the C implementation in Csound. The same control signals are used for each.

The case called "Chronic′" is with the specialization described in Section 4.7—performed by hand, to see what the effect of comprehensions would be.

O'Caml is compiled with ocamlopt -unsafe -noassert, version 3.01. C++ is compiled with g++ -O -funroll-loops, version 2.95.2. The Csound used is version 3.50.

	mix	FOF
C++	*1.0*	—
Csound	—	*1.0*
Chronic	21.2	12.0
Chronic′	2.3	9.9

Table 1: relative times to run benchmarks.

We see that the speed of Chronic without comprehensions is dismal. Chronic′, however, performs quite well on the first benchmark.

Both perform unspectacularly on the second benchmark. Profiling shows that 20% of their time is spent on garbage collection (because elements are created and destroyed at every FOF burst), but more time is lost to minutiae such as float-to-int conversion. Better understanding might allow better performance.

6 Conclusion

Chronic's present execution speed is suitable sometimes for general production use, and sometimes for research and prototyping. Earlier work showed on paper how temporal type constructors make for clearer and more direct programs, and this work shows how these programs can be executed.

References

Brandt, E. (2000). Temporal type constructors for computer music programming. In *Proc. International Computer Music Conference*, pp. 328–331. International Computer Music Association.

Dannenberg, R. B. (1997, Fall). Machine Tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal 21*(3), 50–60.

Leroy, X. (2001). Objective Caml. `http://caml.inria.fr/ocaml/`.

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

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.