

# Temporal type constructors for computer music programming

Eli Brandt

School of Computer Science, Carnegie Mellon University

<http://www.cs.cmu.edu/~eli/L/icmc00>

This paper introduces *temporal type constructors* to computer music programming, and shows how they make languages more expressive. Music programming involves time-structured data types such as audio, MIDI, control signals, and streams of spectral frames. Each computer music language supplies some fixed set of these. Temporal type constructors are instead a way for the programmer to invent these kinds of data, with the ability to manipulate them and their elements. Algorithms expressed in this way can be remarkably brief and clear; FOF (Rodet, 1984) is given as an example.

## 1. Introduction

If you are faced with the task of implementing a computer music algorithm of some complexity, for example FOF synthesis (Rodet, 1984), which programming language will you choose to do it in? Music-N languages like Csound (Vercoe, 1993) are too limited; it cannot reasonably be done. Low-level languages like C are too painful; it will be difficult to write, debug, and maintain. Higher-level vector languages like Matlab (MathWorks, 1997) make the work easier, but not enough easier.

Where is the complexity in implementing FOF? FOF is a technique for synthesizing a pitched signal with a spectral peak, a formant, as found in speech. It generates a stream of grains whose rate controls fundamental frequency, whose pitch controls formant frequency, and whose shape controls formant shape. The stream is irregularly timed, and a grain is a sequence of samples: FOF's heart is an infinite vector of timestamped vectors of samples, a data type with a complex time structure.

In a language which can express and manipulate that type, FOF can be written quite directly. Type constructors make this possible. Furthermore, they are useful far beyond the FOF example, in areas from musical DSP to algorithmic composition.

## 2. Type constructors

Type constructors make new types from existing ones. The idea is familiar from C:

$\alpha$  pointer      a pointer to an  $\alpha$ .

The " $\alpha$  pointer" constructor has one input,  $\alpha$ ; let  $\alpha$  be the primitive type `int` and it will return a type that we write `int pointer` and C writes `int*`: a pointer to an integer.

This paper proposes this set of type constructors:

$\alpha$  event      an  $\alpha$  with a timestamp.  
 $\alpha$  vec      a vector of  $\alpha$ .  
 $\alpha$  ivec      an infinite vector of  $\alpha$ .

These are *temporal* type constructors in that they introduce time structure. In this work time is an integer, from an implicit global sample clock.

An `int event` has a position in time. The  $\alpha$  `vec` and  $\alpha$  `ivec` constructors place their elements at successive times. (In fact they have an ambiguity which will not be fixed until Section 4: we will see soon that they can also be used with no temporal implications. But at this point it isn't necessary to introduce temporal and atemporal versions of each, and the whole issue can usually be ignored.)

The best way to understand these temporal type constructors is by example. They combine to build an extraordinary variety of musically useful types:

Sample ivec vec

Multi-channel audio represented as separate channels, each of which is one element of the `vec`. Remember, the type is read from the tail end: a vector of infinite vectors of samples. The `Sample` type is a placeholder for whatever an audio sample is, perhaps `float` or `int16`.

Sample vec ivec

Multi-channel audio represented as a stream of frames of samples. (The `ivec` in these two examples is temporal, but the `vec` is not: it is logically indexed not by time but by speaker or microphone position.)

float event ivec

A stream of timestamped floating-point values, which can be interpreted as a piecewise-constant signal. This resembles a MIDI controller stream.

Pitch\*Loudness event ivec

A stream of timestamped musical-keyboard events, resembling MIDI note-on messages. (The "\*" forms the *tuple* type where pitch and loudness are paired together.) The timestamps might, depending on the data's use, be non-decreasing.

(Pitch vec)\*(Loudness vec) event ivec

As above, but with pitch and loudness extended to curves for gestural control. (Here the `vec` is temporal and the `ivec` is not.)

Pitch vec event vec  
A chord sequence.

Sample vec event  
Audio which starts at a time before or after time zero.  
(An acausal FIR filter, perhaps?)

Complex vec ivec  
Short-time spectrum data. The `vec` has a frequency axis,  
the `ivec` a temporal one.

((Red\*Green\*Blue) vec vec ivec)\*(Sample ivec vec)  
Movie with multi-channel digital audio.

Sample vec event ivec  
A stream of timestamped audio grains, as used in FOF  
synthesis.

Those familiar with C++ template classes can think about writing an `event<T>` class containing a timestamp and an instance `T`, and a `vec<T>` with an array of `T`. The trick question is what goes in `ivec<T>`. With C++, we think about infinite streams, but we write about blocks and perform streaming through control flow.

An operation like “`ivec<int> B = foo(A);`” could only place in `B` a reference to `A` and a representation of `foo` to be applied as needed. An `ivec<>` ultimately contains a lazy implementation of the entire language.

For this reason `ivec` cannot be implemented as a simple library in any host language that is not lazy itself. *Acute*, a prototype system embedded directly in the functional language O’Caml (Leroy 2000), lacks `ivec`.

### 3. Implementing FOF synthesis

Figure 1 illustrates how a pure functional language with these type constructors expresses FOF. The code shown is logically identical to the working FOF code in *Acute*, but its syntax is imaginary, as O’Caml’s actual syntax is not worth trying to explain.

This imaginary syntax uses local variables like this:

```
{  
  x = 6;  
  x_squared = x*x;  
  x_squared + x  
} + 6
```

A curly-brace block is an expression, whose value is given by its final line. Here it evaluates to 42, and this can be used in a larger expression as shown.

Another common idiom is the application of an anonymous function:

```
V.map (fun x -> 10*x) [1 4 9]  
= [10 40 90]
```

The expression in parentheses is a function of one variable; `map` takes such a function and returns the result of applying it to each element of a vector.

What’s welcome about the code in Figure 1 is that it expresses our idea of the FOF algorithm. It builds grains

one by one and places them in time. 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. When using temporal types, programmers are able to step outside of time, to move time into the data.

Grain start times are generated by `phasor_wrap_times` from the *Acute* utility library, which is like an oscillator but returns 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”.

This unflattened would be of type `float vec event ivec`, except that *Acute* only permits it to be `float vec event vec`: FOF doesn’t run indefinitely. O’Caml infers types, but this one type is included explicitly.

Finally FOF needs a flattened version of unflattened, with the overlapping grains mixed to a single audio stream.

All of the temporal bookkeeping, the worry about overlapping grains, that the code has evaded, is reusably encapsulated in `ifold` from the *EV* library. This “vertically folds” a `vec event vec` down to a `vec`:

```
EV.vfold (+) 0.0 [[1 2 3 4]@2 [50 60 70]@4]  
= [0 0 1 2 53 64 70]
```

The mixed-down stream is FOF’s output.

On first reading, skip the code for *octaviation*.

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].

The code in Figure 1 uses functions from several libraries, whose uses should be deducible from context.

- `V`, the *Acute* vector library: `take`, `map1`, `map3`, `iterate`.
- `EV`, the *Acute* event/vector library: `pwl`, `ifold`.
- `U`, utilities: `phasor_wrap_times`, `db_to_amp`, `osc_sine`, `osc_sine_v`.

One might compare this code with the implementation of FOF found in *Csound*. In terms of lines of code, the version using temporal type constructors is one fifth the size, or one third if it includes the utility functions. More importantly, its expression of the algorithm is relatively direct and comprehensible.

Figure 1: FOF in Acute

```

function fof (t_max,      // times are in samples; frequencies are normalized to the sampling frequency.
  ffund, phase0, octavi, // fundamental, initial phase, "octaviation": float vec.
  fform, bw, db,         // formant frequency, bandwidth, and level: float vec.
  rise, fall, dur,       // rise time, fall time, and total length: float.
  risefall_table,       // first half is rise, second half fall: float vec.
  lfmode) =              // boolean: track fform during a grain?
{
  (times: float vec) = U.phasor_wrap_times(t_max, ffund, phase0); // grain start times

  // to octaviate down one octave, fade out every second grain; etc.
  function octmul(oct, i) = { // from octaviation and grain number, find multiplier.
    (octf, octi) = modfi(oct); // fractional and integer parts
    pow2 = 1 << octi; // 2octi = 1 0...0 (binary)
    // the following expression is the value of the braces-block, and thus of the function.
    if 0 < (i & (pow2 - 1)) then 0.0 // not a multiple of 2octi
    else if 0 = (i & pow2) then 1.0 // of the form (2n)*(2octi)
    else 1.0-octf // otherwise partially fade this grain.
  }

  (unflattened: float vec event vec) = // the stream of grains.
  V.mapi (fn (i, t) -> { // anonymous function, mapped over times, takes i and times[i]
    int_t = int(t);
    octm = octmul(octavi[int_t], i); // get current value.
    fform0 = fform[int_t]; db = db[int_t]; bw = bw[int_t];
    grain = if octm=0.0 then [] else { // [] is zero-length vector
      ampl = octm * U.db_to_amp(db[int_t]);
      phase0 = frac(t) * fform0; // position the sinusoid precisely.
      sine = if not lfmode then U.osc_sine(dur, fform0, phase0)
        else U.osc_sine_v(dur, V.take(fform, int_t, dur), phase0);
      k_env = exp(-pi*bw); // decay multiplier. slightly magic.
      env = V.iterate(dur, (fn x -> k_env * x), 1.0); // exp decay
      smooth_idx = // 0 linear to 0.5, then flat, then 0.5 to 1
        EV.pwl(dur, 0.0, [0.5 @ rise, 0.5 @ (dur-fall), 1.0 @ dur]);
      smooth = U.tablei(risetab, smooth_idx); // table lookup to get rise, flat top, fall
      V.map3((fn (x, y, z) -> ampl*x*y*z), sine, env, smooth) // multiply these to give grain.
    }
    grain @ int_t // the anonymous fn's result is a timestamped grain.
  }) times; // (finally, the second arg to mapi.)
  EV.vfold (+) 0.0 unflattened; // mix the grains to get fof's value.
}

```

## 4. Extensions

Chronic, the successor to Acute, will include the infinite vectors that Acute lacks. Attractive further extensions to the type system include finite and infinite *signals*, which would include a sampling rate, and *matrices*, or a way of expressing a type  $\alpha$  vec vec where each  $\alpha$  vec is of the same length. Differentiating signals from vectors would fix the ambiguity mentioned in Section 2.

Support for general feedback delay structures is an important feature of Chronic, but is not relevant here.

## 5. Implementation

The implementation of a language as described is work in progress and a topic for another paper, but the problems need to be mentioned here. They are the dark side of the language's power, and they force constraints on it.<sup>†</sup> An *ivec* could represent microphone input, but what if I index

---

<sup>†</sup> (Absolute power is absolutely unimplementable.)

into it to get a sample from the arbitrarily distant past? This must be disallowed.

A less pressing problem is that serializing complex out-of-order data access is tricky, and inefficient if done wrong. I believe, but have not yet demonstrated, that it can be done right, and that the entire language can run very roughly on a par with C, perhaps half as fast.

## 6. Related work

Many languages include particular instances of temporal types: audio, or streams of MIDI events. This paper emphasizes abstracting the type constructors so that not only those particular types but combinatorially many others can be formed.

Remarkably few languages have used type constructors similar to `vec` and `event`. Fran, a language for animated graphics, takes a related but distinct approach. The original version (Elliot and Hudak, 1997) has `Behavior` and `Event` constructors, which parallel `ivec` and `event`. A `Behavior`, however, is a function of continuous time rather than discrete. `Behavior` is more general than `ivec`, more permissive; it carries less information about the data.

This matters computationally: a general `Behavior` cannot usefully be operated on blockwise or in any other way than lazily by individual time; no length of it can fit in a delay line. It matters conceptually: a `Behavior` has no inherent notion of “next element”, so digitally filtering a `Behavior` makes little sense, and a float `Event Behavior` is not at all like the float `event ivec` in Section 2. Recent versions of Fran (Elliot, 1998) have elaborated `Event` so that it is more like an `event ivec` in itself.

Fran's use of continuous time is elegant, and is well suited to animation. In part this is because the frame rate is over three orders of magnitude lower than for audio: discrete time's efficiency is less needed, and its quantization error is more problematic. Daniels (1999) has worked on the semantics of the continuous time model, and Elliot (1998) describes several approaches towards its implementation.

## 7. Conclusions

This work is in pursuit of a high-level computer music language that makes low-level ones unnecessary. This means that you can write new DSP in it, rather than dropping down to a low-level language to write a black-box unit generator. It means that you can define and manipulate a new data type—LPC frames, wavelets, short-time bispectra—without dropping down to write a whole family of black boxes to operate on it.

Type constructors make this possible because constructed types are *transparent*: the type-elements they were constructed from are accessible. In contrast, many languages provide audio as an opaque primitive type, but

have no concept of a sample. Transparency lets high-level structure coexist with low-level manipulation, expressiveness with power.

Computer music programmers think about temporal structure. With temporal type constructors they can write about it.

## 8. Acknowledgements

Thanks to those who commented on drafts of this paper: Jim Wright, Daniel Oppenheim, Robert Fuhrer, Roger Dannenberg, and Steve Abrams, who brought up the point that some vectors are temporal and some are not.

## 9. Bibliography

- Daniels, Anthony (1999).  
A semantics for functions and behaviors.  
PhD thesis, University of Nottingham, Dec. 1999.
- Elliott, Conal, and Paul Hudak (1997).  
“Functional reactive animation”.  
Proc. ICFP97, p. 263.
- Elliot, Conal (1998).  
“Functional Implementations of Continuous Modeled Animation”.  
Proc. PLILP/ALP '98. Extended version:  
<ftp://ftp.research.microsoft.com/pub/tr/tr-98-25.pdf>
- Leroy, Xavier (2000).  
The Objective Caml system, release 3.00.  
Institut National de Recherche en Informatique et en Automatique. <http://caml.inria.fr/ocaml/htmlman>
- MathWorks (1997).  
Matlab language reference manual. Version 5.  
The MathWorks, Inc.
- Rodet, Xavier (1984).  
“Time-domain formant wave-function synthesis”.  
*Computer Music Journal*, 8(3): 9–14.
- Vercoe, Barry (1993).  
Csound — a manual for the audio processing system and supporting programs with tutorials.  
MIT Media Lab.