Henkjan Honing
Institute for Logic, Language, and Computation (ILLC)
Faculty of Mathematics and Computer Science
University of Amsterdam
Spuistraat 134
NL–1012 VB Amsterdam, The Netherlands
Honing@FWI.uva.nl

The Vibrato Problem: Comparing Two Solutions

In discussing the formalization of musical knowledge, this article describes an important music-representation issue, the "vibrato problem." This problem characterizes the need for a knowledge representation that can reflect both discrete and continuous aspects of music at an abstract and controllable level. Two formalisms of *functions of time* that support this notion are compared: the approach used in the Canon family of computer music composition systems (Dannenberg, McAvinney, and Rubine 1986; Dannenberg 1989; Dannenberg, Fraley, and Velikonja 1991), and the Generalized Time Functions (GTF) Formalism of Desain and Honing (1992a, 1993). The comparison is based on a simplified version of Dannenberg's Arctic, Canon, and Fugue systems (referred to as ACF), obtained from the original programs using an extraction technique, and a simplified version of the GTF system that was made syntactically identical to ACF. In general, both approaches solve the vibrato problem, though in very different ways. The differences are explained in terms of abstraction, modularity, flexibility, transparency, and extensibility—important issues in the design of a representational system for music (Honing 1993b).

Aspects of Musical Knowledge

In music representation, a distinction can be made between discrete, symbolic representations (such as music notation) and continuous, numerical representations (as audio or control signals) (see, e.g., De Poli, Piccialli, and Roads 1991). In common practice Western music notation can represent symbolic constructs such as notes, rests, accents or meter, but it lacks ways of describing the continuous aspects of music (for example, the individual shaping of a note), other than using simple symbols or words such as tremolo or sforzato in the score. By contrast, an audio-signal representation allows a "complete" description of a piece of music, with all its continuous aspects. It includes, for example, the instrument's sound quality, the room acoustics, etc. This type of representation does not have symbolic characteristics, however; we cannot (at least not directly) derive from it the different streams or voices, the beginning of a note, or the metrical structure.

A similar distinction can be found in computer music systems, with discrete, note- and event-oriented MIDI systems at one end, and continuous, signal-oriented, Music V-like systems at the other. Sometimes one type of representation is more appropriate than the other, but a powerful representation system for music must integrate both aspects. To give an example, one might want to describe how certain parameters change continuously over time, with respect to specific parts or levels of the discrete structure. The representation system must incorporate specific knowledge on how these parameters change or behave under transformation of that structure (for instance, how a rhythmic fragment's particular kind of phrasing depends on its duration). We need to communicate information between the continuous and discrete aspects of a representation, passing information from the discrete components (for example, notes) to the continuous components (such as control functions), and vice versa. The "vibrato problem" (Desain and Honing 1992a) is a relatively simple

Computer Music Journal, 19:3, pp. 32–49, Fall 1995
© 1995 Massachusetts Institute of Technology.

representational problem, characterizing the kind of control that is needed.

The Vibrato Problem

In Figure 1a, a continuous (control) function is used for the pitch attribute of a discrete object—a note. The problem revolves around what should happen to the shape or form of the pitch contour of a vibrato when it is used for a longer note, or, equivalently, when the note is stretched. In the case of its interpretation as a simple sinusoidal vibrato, some extra vibrato cycles should be added to the pitch envelope (see the first frame in Figure 1b)—when interpreted as a sinusoidal glissando, the pitch contour should be elastically stretched (see the second frame in Figure 1b). However, all kinds of intermediate and more complex behaviors should be expressible as well (see third and fourth frames in Figure 1b). A similar kind of control is needed with respect to the start time of a discrete object (see Figure 1c): What should happen to the contour when it is used for an object at a different point in time, or, equivalently, when the note is shifted? Again, a large range of possible behaviors can be thought of, depending on the interpretation of the control function, i.e., the kind of musical knowledge embodied—attack-transients, independent or synchronized vibrati, or other functions of time (see Figure 1d).

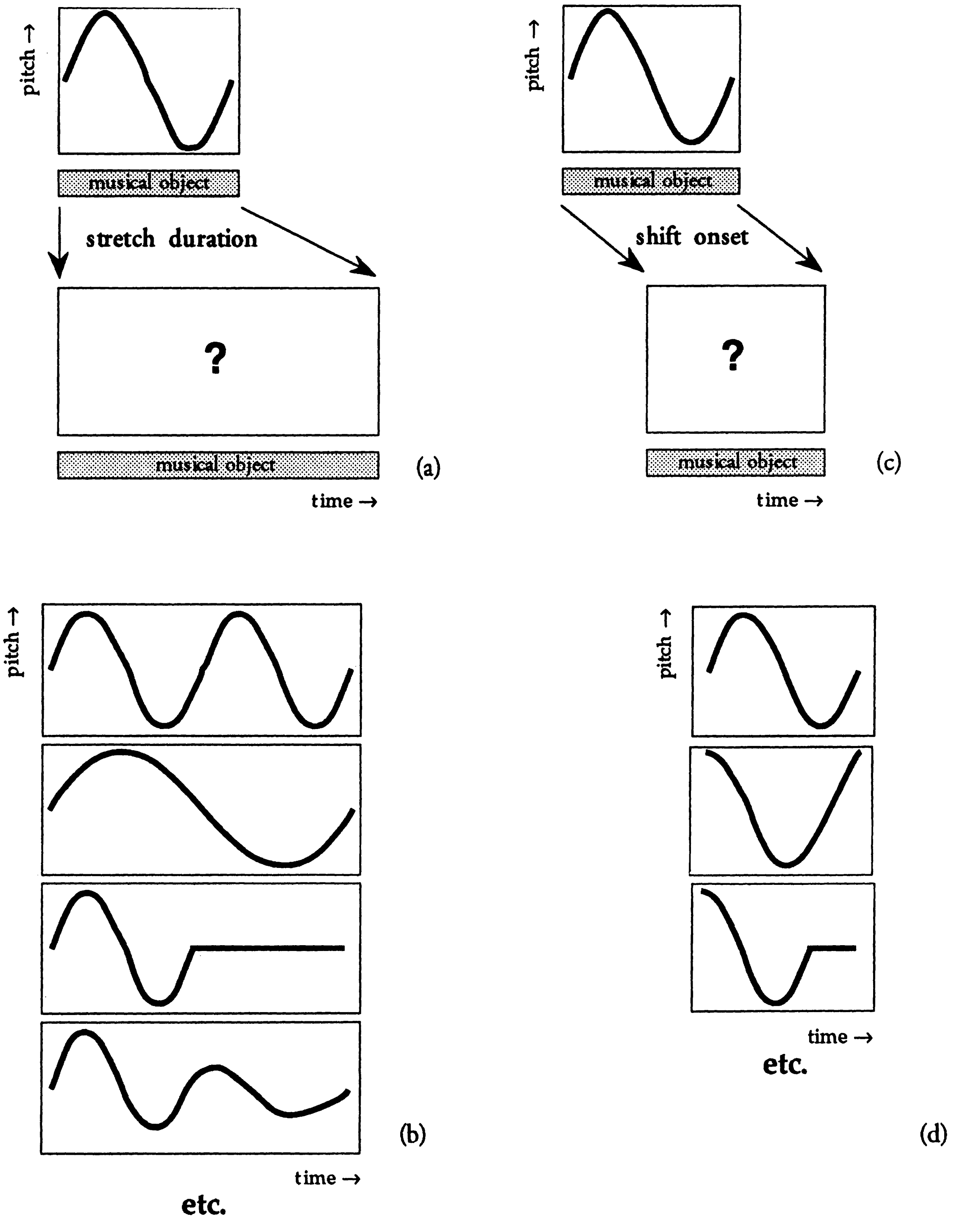
To get the desired isomorphism between the representation and the reality of musical sounds, a music representation language must support a property that we will call "context-sensitive polymorphism." "Polymorphism" for the fact that the result of an operation (like stretching) depends on its argument type (e.g., a vibrato time function behaves differently under a stretch transformation than a glissando time function), "context-sensitive" because an operation is also dependent on the lexical context in which it is used. As an example of the latter, interpret the situation in Figure 1c as two notes that occur in parallel, with one note starting a bit later than the other. The behavior of this musical object under transformation is now also dependent on whether a particular control function is linked to the object as a whole (i.e., to describe synchronized vibrati; see second frame in Figure 1d), or is associated with the individual notes (e.g., an independent vibrato; see first frame in Figure 1d). Specific language constructs are needed to made a distinction between these different behaviors.

Note that the vibrato problem is, in fact, a general issue in temporal knowledge representation, and is not restricted to music. In animation, for example, we could use similar representation formalisms. Think, for instance, of a scene in which a comic-strip character walks from point A to point B in a particular way. When one wants to use this specific behavior to have the character walk over a longer distance, should the character make more steps (cf. vibrato) or take larger steps, i.e. should it start running (cf. glissando)?

Dannenberg (1989) describes the "drum roll problem"—the discrete analogy of the vibrato problem—which in the case of stretching should be extended by adding more drum hits, instead of slowing down the rate of the drum roll. Several systems are based on this idea: the Arctic system (Dannenberg, McAvinney, and Rubine 1986), the Canon score language (Dannenberg 1989), the Fugue composition language (Dannenberg, Fraley, and Velikonja 1991), and Fugue's latest incarnation, Nyquist (Dannenberg 1993). Although these systems differ in several aspects, they all use a transformation system similar to the one proposed in Arctic. This shared mechanism of Arctic, Canon, and Fugue (and Nyquist) will be referred to as the ACF transformation system.

The core of the observations in this study are based on analyzing the behavior of simplified versions of ACF and GTF, extracted from the original code using programming language transformation techniques (e.g., Friedman, Wand, and Haynes 1992). This technique of extraction (Honing 1993a), making a small program from a larger system, is an attractive alternative to rational reconstruction (e.g., Richie and Hanna 1990). We will refer to such a simplified program as *micro-version* or *microworld*. It consists of a relatively complete

Figure 1. The vibrato problem. First, what should happen to a control-function contour when used for a discrete musical object with a different length? For example, a sine wave control function is associated with the pitch attribute of a note—vibrato (a); possible pitch contours for the stretched note, depending on the interpretation of the original contour, are shown in (b). Second, what should happen to the pitch-contour form when used for a discrete musical object at a different point in time (c)? Possible pitch contours for the shifted note are shown in (d). There is, in principle, an infinite number of solutions, depending on the type of musical knowledge embodied by the control function.



set of essential objects and mechanisms, and at the same time it is small and easy to comprehend.

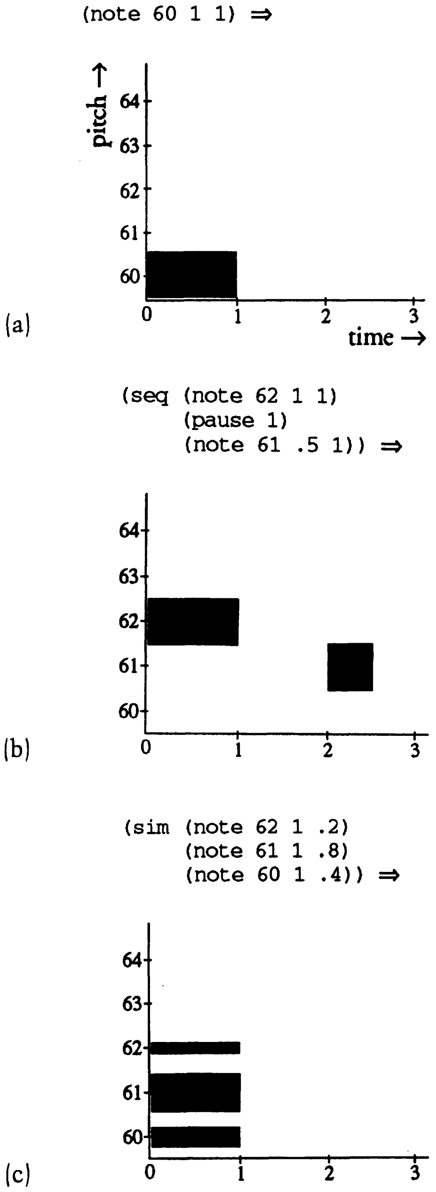
Shared Framework of ACF and GTF

First, we will describe the set of musical objects, time functions, and their transformation that is shared by the ACF and GTF micro-versions. The full Lisp source code of the micro-versions is available by Internet ftp from the *Computer Music Journal* archives; it can be found in the directory with the uniform resource locator (URL) ftp://www-mitpress.mit.edu/Computer-Music-Journal/Code/ACF_GTF. Both micro-versions use the Canon syntax (Dannenberg 1989). On some points the ACF systems differ among themselves, this will be noted where appropriate. The examples will be presented with their graphical output, using the micro-versions mentioned above.

In general, both the ACF and GTF systems provide a set of primitive musical objects (in ACF these are referred to as "behaviors"), and ways of combining them into more complex objects. Examples of basic musical objects are `note`—with parameters for duration, pitch, amplitude, and other attributes that depend on the synthesis method used, and `pause`—a rest with duration as its only parameter. These basic musical objects can be combined into compound musical objects using the time structuring constructs `seq` (for sequential ordering) and `sim` (for simultaneous or parallel ordering). Some examples are given in Figure 2, which shows simple pitch-time diagrams.

New musical objects can be defined using the standard procedural abstraction (function definition) of Lisp, for example, the following Lisp expression defines a function named `melody` that consists of three sequential notes. Figure 3 shows an example of its use in a simplified pitch-time diagram where the thickness of the "note" corresponds to its loudness.

```
;;; define a function
(defun melody ()
 ;;; that produces a sequence
 ;;; of three notes given as
 ;;; (note pitch duration amplitude)
 (seq (note 60 .5 1)
    (note 61 .5 1)
    (note 62 .5 1)))
```


```
(note 60 1 1) ⇒

(seq (note 62 1 1)
     (pause 1)
     (note 61 .5 1)) ⇒

(sim (note 62 1 .2)
     (note 61 1 .8)
     (note 60 1 .4)) ⇒
```

Figure 2. Examples of basic and compound musical objects in the ACF family of languages and GTF. Pitches are given as MIDI key numbers, duration as seconds, and amplitude on a 0 to 1 scale. A note with pitch 60, duration 1, and maximum amplitude (a); a sequence of a note, a rest, and another, shorter note (b); and three notes in parallel, each with different pitches and amplitudes (c). The height of each note bar is proportional to the corresponding note's amplitude.

Both ACF and GTF provide a set of control functions—functions of time—and ways of combining

Figure 3. Pitch vs. time diagram of a sequence that consists of a user-defined musical object (a three-note melody) played twice.

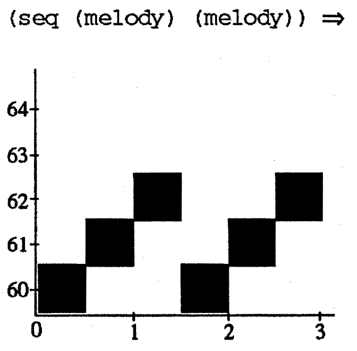
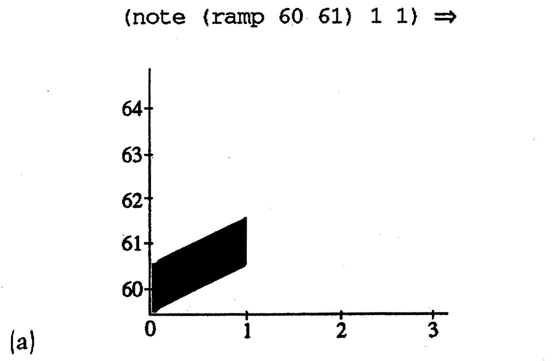
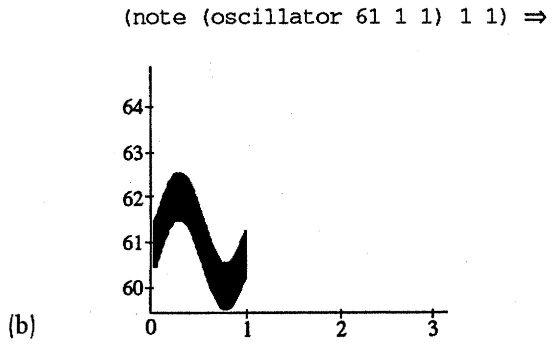


Figure 4. Two examples of a note that is parameterized with basic time functions given as its pitch values. An interpolating linear ramp with start and end values as parameters (a), and a sine wave oscillator with offset, modulation frequency, and amplitude as parameters (b).





them. We will use two basic time functions in this article: a linear interpolating `ramp` function, and an `oscillator` that generates a sine wave.

There are alternative ways of passing time functions to musical objects. One method is to pass a function directly as an attribute to, for instance, the pitch parameter of a note (see Figure 4). An alternative method is to make a musical object with simple default values and to obtain the desired result by transformation. In one context the first method might be more appropriate, in another context, the latter. The following examples show the equivalence between specification by means of transformation and by parameterization (their output is as shown in Figure 4a and 4b, respectively):

```
;;; specification by transformation
;;; of a ramp glissando
(trans (ramp 0 1) (note 60 1 1)) =
;;; specification by parameterization
(note (ramp 60 61) 1 1))

;;; specification by transformation
;;; of a vibrato using an oscillator
(trans (oscillator 0 1 1) (note 61 1 1)) =
;;; specification by parameterization
(note (oscillator 61 1 1) 1 1))
```

Note that, while specification by means of transformation is supported in both ACF and GTF, specification by means of parameterization is only available in Arctic and GTF.

Finally, both systems support different types of transformations. As an example of a time transformation, `stretch` will be used (see Figure 5a). This transformation scales the duration of a musical object (its second parameter) by a given factor (its first parameter). As examples of attribute transformations we will use one for pitch (named `trans`), and one for amplitude (named `loud`). These transformations take constants (see Figure 5b, 5c, and 5d) or time functions (see Figure 5e and 5f) as their first argument, and the object to be transformed as their second argument.

The ACF Transformation System

A central concept in the ACF systems is the notion of a transformation environment. This environment, or context, is implemented as a number of global variables that are dynamically bound and serve as implicit parameters to every "behavior" (i.e., musical object). Behaviors, transformations, and time functions can, in principle, inspect, ignore, or modify these variables. They are procedures that know how to change (or "behave") in response to, for example, a stretching or transposi-

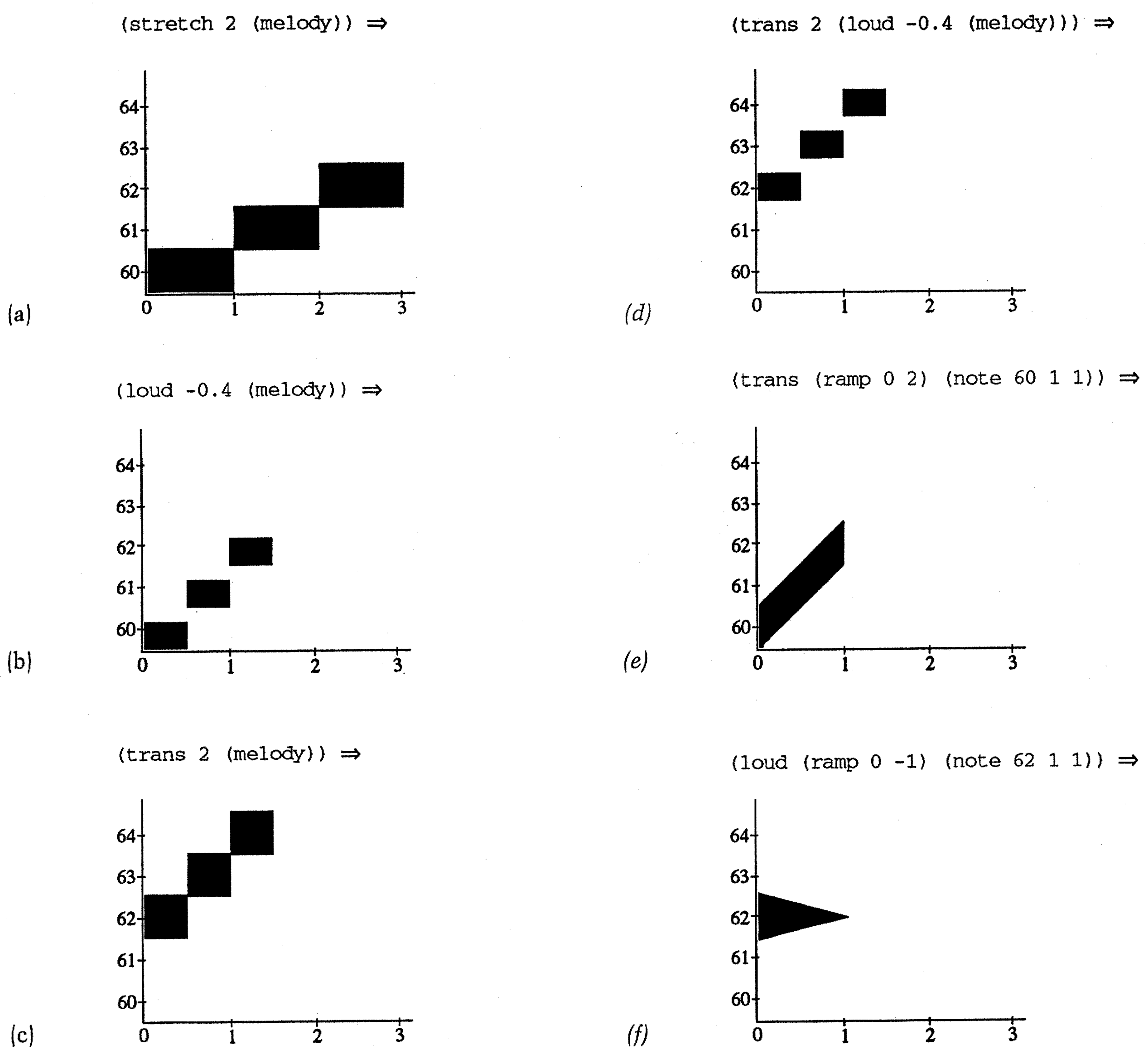


Figure 5. Examples of transformations on musical objects: a time transformation—`stretch` *(a); a constant amplitude transformation—*`loud` *(b); and a constant pitch transformation—*`trans` *(c). A few of the many possible nestings of these two transformations are shown: a transposed quiet melody (d) and a time-varying pitch transformation (e); and a time-varying amplitude transformation (f).*

tion transformation, and produce continuous signals (e.g., graphical output or MIDI) as a side effect. The ability of behaviors to adapt themselves—in their own specific way—to changes of the values of these environment variables is the basis of the ACF solution to the vibrato problem; for example, a vibrato behavior will behave differently in an environment modified by a `stretch` transformation than will a glissando behavior.

While dynamic binding is a popular programming technique, it often makes a proper understanding of the resulting execution very difficult. To get a precise insight in how ACF makes use of this technique, we first concentrate on the special variables in the environment that have to do with time, and take a simple `note` behavior as an example. We will use diagrams to illustrate the specific communication of these implicit parameters

Figure 6. The variable binding and scope-of-reference diagram for the expression (note p d)*. The symbol ← is used for assignment, + for addition, and × for multiplication; italics are used for functions, bold type for formal parameters, bold names above a frame indicate an operator or transformation, and curved arrows emphasize references. Although* note*, in reality, has more than two parameters, in these diagrams it is sufficient to look at pitch* p *and duration* d *only.*

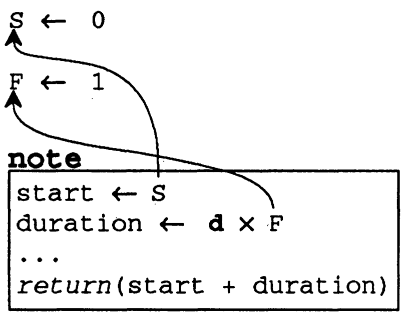


Figure 7. Binding and scope diagram for the expression (seq (note p_0 d_0) (note p_1 d_1))*, a sequence of two notes. See Figure 6 caption for symbol explanations.*

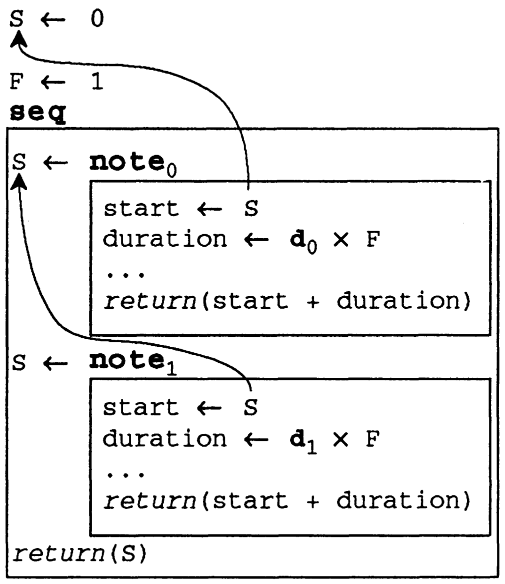


Figure 7

Figure 8. Binding and scope diagram for the expression (sim (note p_0 d_0) (note p_1 d_1))*, two notes in parallel. See Figure 6 caption for symbol explanations.*

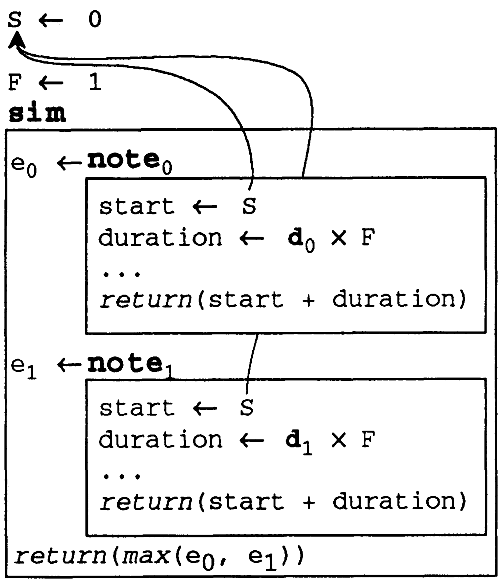


Figure 8

and the dynamic binding scheme used. In the figures that illustrate variable binding, the symbol ← is used for assignment, + for addition, and × for multiplication. Italics are used for functions, bold type for formal parameters, bold names above a frame for an operator or transformation, and curved arrows emphasize references.

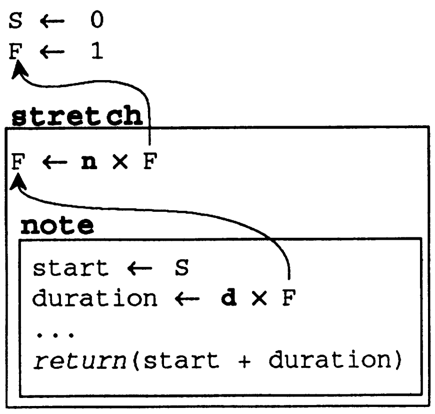
There are two implicit parameters in the environment that have to do with time. One holds the current start time (called time in the ACF systems, but here referred to as start or S, to distinguish it from actual time or "now"), the other a duration stretch factor (called dur in Arctic and Canon, and stretch in Fugue; we will use stretch or F, to avoid confusion with duration). The global environment is initially set with S (start time) as 0 and F (stretch factor) as 1 (see Figure 6). A note procedure, evaluated in this environment, derives its start time and its stretched duration (i.e., product of the note's formal parameter d, for duration, and F) from these implicit parameters found in the environment in which the note is evaluated. The body of note (indicated with an ellipsis in Figure 6) may refer to these time parameters. Note that behaviors return their end time (or logical stop time, as it is referred to in the ACF systems) for use by the time structuring behaviors seq and sim.

In Figure 7, an example of the seq behavior is shown. It modifies the environment and as a result (using dynamic binding), influences the behavior of note. The returned end time, after evaluating

Figure 9. Binding and scope diagram for the expression (stretch n (note p d)), *a note made* n *times as long. See Figure 6 caption for symbol explanations.*

Figure 10. Binding and scope diagram for the expression (trans n (note p d)), *a note with a pitch transposed by a constant* n. *See Figure 6 caption for symbol explanations.*

Figure 11. Binding and scope diagram for the expression (trans (oscillator of a) (note p d)), *a note with a pitch transposed by a sine wave time function constructor with parameters offset, modulation frequency and amplitude.*



the first note, is used to set the value of S. This new value is then used when evaluating the next note, resulting in the notes being ordered (i.e., played or drawn) one after the other.

A sim behavior, conversely, will evaluate all its arguments with the same start time and return the maximum end time (see Figure 8).

A time transformation in this diagrammatic notation is shown in Figure 9. The stretch transformation alters the duration stretch factor of the enclosing environment by multiplying it with a factor. As a result, the note's duration will be *n* times as long for a stretch factor of *n*.

The next example is an attribute transformation (Figure 10). The trans transformation is used for transposing the pitch of behaviors (if they have such an attribute). The special variable transpose is therefore introduced in the environment as illustrated in Figure 10. The note behavior adds the value of transpose to its own explicit pitch (the formal parameter p). For every other transformable attribute (e.g., loudness, channel, or articulation factor), such a special attribute variable is added to the environment.

Finally, in Figure 11 a time-varying transformation is shown in the same diagrammatic way for comparison. In this example, an oscillator function is an argument to the trans transformation. Instead of adding a constant value to the value of transpose, a new expression is built from the result of evaluating the oscillator constructor and the value of transpose in the en-

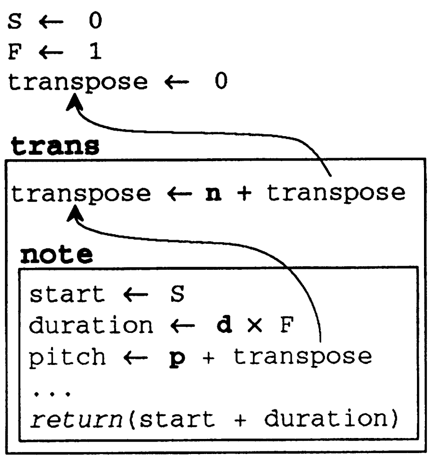


Figure 10

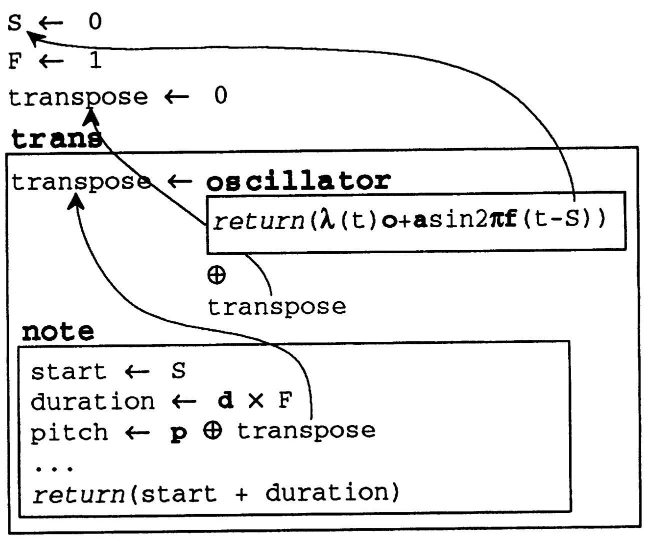


Figure 11

closing environment (here it is 0, but could be a time function as well). The note procedure body (i.e., the ellipsis in Figure 11) can refer to this "composed" pitch value. Note that oscillator is actually a time function constructor, that is, it returns a time function. Lambda expressions are used to refer to these anonymous time functions. They are of the form $\lambda(x_1, ..., x_n)e$, where $x_1, ..., x_n$ are parameter names and e some expression.

Implementation

The technique used to make nesting of operations on different attributes possible, and to communicate the appropriate values of the environment variables to the behaviors, is dynamic binding. Time functions, behaviors, and transformations can refer to free, but invisible (at the user-level) environment parameters. It simplifies procedure-call by using implicit parameters that communicate information to the behaviors (`start`, `stretch`, `transpose`, etc.), and, therefore, mainly cleans up the syntax (i.e., syntax abstraction). (Note that the transformations in ACF are coded as Lisp macros, not as functions.) However, since these environment parameters play a central role in the behavior of the language, the user must be aware of its workings when using or extending the language, so there is no real abstraction from these implementation details (see Abelson and Sussman 1985).

Furthermore, a particular kind of *delayed evaluation* is used. Symbolic expressions, describing functions of time, are combined into new expressions that are not yet evaluated. Only at run-time (for example, when the picture is generated) will these expressions be evaluated, and return a fully transformed function of time. This mechanism and the functions of time are made explicit in the ACF microworld using a time function combinator (shown as ⊕ in the figures).

Equation 1 shows an example of a time function constructor (`oscillator`) that returns an anonymous function of time $\lambda(t)$. Its behavior is described by an expression that has access to time t, the formal parameters of the `oscillator` constructor, i.e., o (`offset`), f (`frequency`), and a (`amplitude`), and to S (`start time`) that is bound to its value in the enclosing environment (cf. Figure 11).

$$oscillator(o, f, a) = \lambda(t)o + a \sin 2\pi f(t - S) \qquad (1)$$

The evaluation of, for example, (`oscillator 62 1 0.5`), will produce a closure that consists of a function of time $\lambda(t)$ that has bindings to its three formal parameters (o, f, and a) and to the current (i.e., define time) value of S (S is not a formal parameter).

The GTF Formalism

The approach that was taken in Desain and Honing (1992a, 1993) is that of a mixed representation—describing those aspects that are best represented numerically by continuous control functions, and those aspects that are best represented symbolically by discrete objects. Together, these discrete musical objects and continuous control functions can form alternating layers of discrete and continuous information. For example, a phrase can be associated with a continuous amplitude function, while consisting of notes associated with their own envelope functions, which are in turn divided into small sections, each with its specific amplitude behavior. The lowest layer could even be extended all the way down to the level of discrete sound samples.

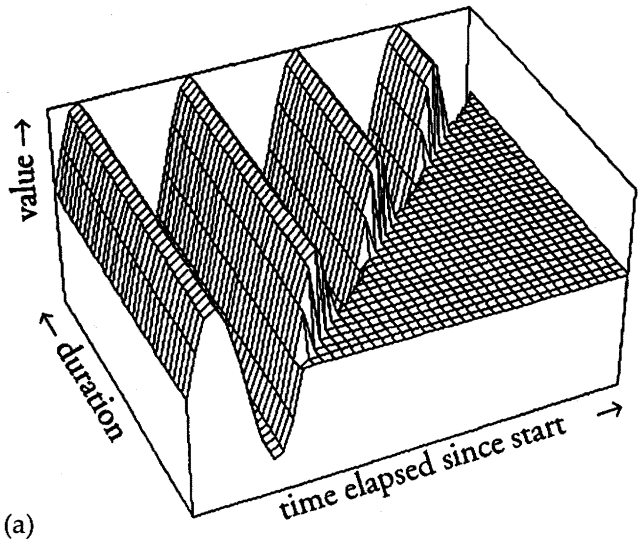
With respect to the continuous aspects (the vibrato problem), control functions of multiple arguments were proposed—so called "time functions of multiple times" or generalized time functions (GTF). These are functions of the actual time, start time and duration (or variations thereof) that can be linked to a specific attribute of a musical object.

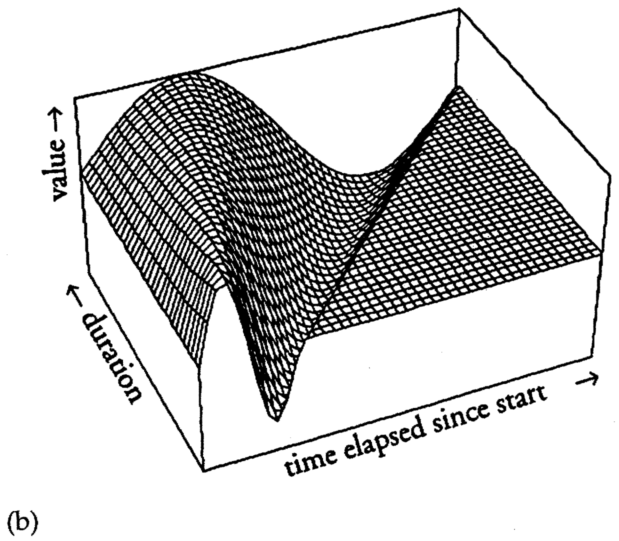
If we ignore for the moment the dependence of time functions on absolute start time, they can be plotted as three-dimensional surfaces; they show a control value for every point in time, given a certain time interval (see Figure 12). Similar plots could be made that show a surface dependent on start time. A specific surface describes the behavior under a specific time transformation (e.g., stretching the discrete object it is linked to). This surface is shown for a simple sinusoidal vibrato (Figure 12a) and a sinusoidal glissando (Figure 12b). In these pictures, the flat triangle-shaped surface of a constant value should be considered undefined. An extension of the GTF micro-version explicitly deals with defining reasonable extrapolations of these functions outside the time interval of the object they are used for, but this is beyond the scope of this article.

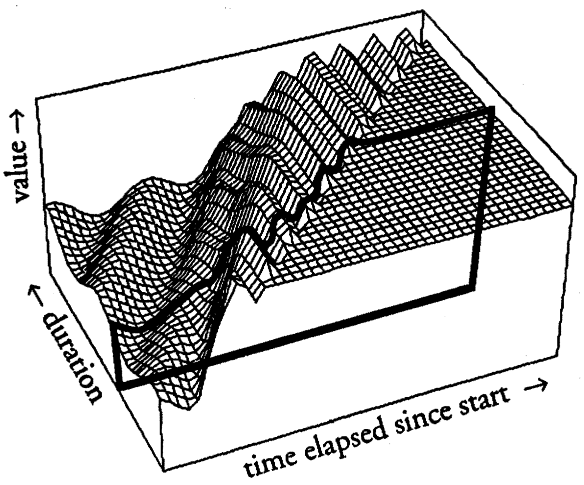
A vertical slice through such a surface describes the characteristic behavior for a certain time inter-

Figure 12. Two surfaces showing the values for generalized time functions as a function of time and duration (start time is ignored in this depiction). In the case of a sinusoidal vibrato, we add more periods of the vibrato function for longer durations (a), whereas for a sinusoidal glissando, the function stretches along with the duration parameter (b).

Figure 13. A more complex generalized time function as a function of time and duration (start time is ignored in the depiction). The appropriate time function to be used for an object of a certain duration is a vertical slice out of the surface.







val—the specific time function for a musical object of a certain duration, as shown in Figure 13.

Furthermore, there are standard ways of combining basic GTFs into more complex control functions, using a set of combinators (compose, concatenate, multiply, add, etc.), or by supplying GTFs as arguments to other GTFs while the components retain their characteristic behaviors. Discrete musical objects (like `note` and `pause`) also have standard ways of being combined into new ones (e.g., using the time structuring functions `S` and `P`—similar to `seq` and `sim` in ACF). To integrate these continuous and discrete aspects, the system provides facilities that support different kinds of communication between continuous control functions and discrete musical objects. For example, control functions can be passed to attributes of musical objects either by parameterization (one passes it directly to an attribute of, e.g., a note) or by transformation (where the musical objects have default values for their attributes and the desired result is obtained by transformation of the object). Several other paths of communication are supported as well, for instance, passing control functions "laterally" between musical objects (i.e., to have access to the control functions of the preceding or succeeding musical objects in a sequence, e.g., to represent transitions between notes) or a "bottom-up" type of communication where some outer control function is dependent on the behavior of one or more embedded control functions (e.g., when defining an overall amplitude time function that behaves like a compressor). However, we will not discuss these types of communication here (see Desain and Honing 1993 for more details).

Implementation

Musical object generators (like `note`, `seq`, or `sim`) are functions of start time, stretch factor, and an environment. The latter supports a purely functional notion of environment (Henderson 1980), and is mainly used to define attribute transformations in the microworld (see `attribute-transform` in the GTF micro-version). Other usage is beyond the scope of this article. Musical object generators can be freely transformed by means of function composition, without actually being calculated, using delayed evaluation. These functions are then only applied to a given start time, stretch factor, and environment, and return data structure describing the musical object that, in turn, can be used as input to a play or draw system. This data structure could take many forms, as long as it contains the start time and duration of the object and it is possible to associate GTFs with attributes of such objects. In the GTF micro-version, an ad hoc unstructured event-list representation is used for simplicity—the full system uses a more elegant set of hierarchical musical objects.

Generalized time functions are functions of three arguments, start, duration and actual time (i.e., $\lambda(s, d, t)$). Equation 2 shows an example of an `oscillator` time function constructor that returns such a function (Note that in the case of oscillator, the duration parameter d is ignored.)

$$oscillator(o, f, a) = \lambda(s, d, t) o + a \sin 2\pi f (t - s) \quad (2)$$

The interpreter system that, for example, generates pictures or prints text, will communicate the start time (s) from the object with whose attribute the GTF is associated, and sample the resulting time function (i.e., a slice out of the specific GTF space; cf. Figure 13) according to the needs of the output medium.

The micro-version of the GTF system contains only the objects and mechanisms central to the current discussion. The naming and order of arguments of the top-level functions is adapted such that the user-level syntax is identical to that in the ACF micro-version.

Comparison

As we saw above, in the ACF systems, a time function is a function of time that has access to variables representing duration, start, and stretch factor. In the GTF formalism, a time function is a function of multiple arguments—start, duration, and actual time. Both formalisms acknowledge that, next to absolute time, both start time and duration are needed to describe appropriate time-varying behavior under time transformation—for example, to be able to distinguish between a glissando and a vibrato. There are, however, several fundamental differences between the two formalisms that are not easily identified at first sight. To explore them, the syntax of the GTF was made identical to ACF. With this identical syntax, we can "port" expressions from GTF to ACF and vice versa, and compare the (graphical) output—when identical expressions result in the same graphical output, we know that the systems have the same semantics.

Referential Transparency

First, let us look at an example of a compound musical object as shown in Figure 14a. It consists of two notes in a sequence separated by a rest. Both notes have an `oscillator` time function associated with their pitch attribute, a duration of 1.0 sec and 1.5 sec, respectively, and constant amplitude. The `pause` has a duration of 0.5 sec. The expression has identical output in ACF and GTF.

Suppose we want to abstract from this particular expression. We can do this by making a procedure (using Lisp function definition) that takes any time function and communicates it to the pitch parameter of the notes. This would give us the following expression.

```
;;; abstract from the pitch parameter
(defun a-musical-object (pitch)
  (seq (note pitch 1 1)
      (pause .5)
      (note pitch 1.5 1)))
```

When we look at the output of this function applied to the same time function that was used in Figure 14a, we see that its semantics are different in ACF and GTF, as is illustrated in Figure 14b. In ACF, the sine wave extends over the rest, whereas in GTF the sine wave starts at phase 0 at the beginning of each note. The same thing happens in the closely related expression shown in Figure 14c that uses a `let` binding (the `let` construct being useful "syntactic sugar" for function application). This specific difference in semantics between ACF and GTF can be explained by having a closer look at the two definitions of an `oscillator` time function in the two formalisms (the equations for which are repeated below).

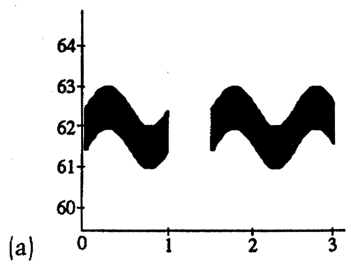
ACF: $oscillator(o, f, a) = \lambda(t) o + a \sin 2\pi f(t - S)$ (3)

GTF: $oscillator(o, f, a) = \lambda(s, d, t) o + a \sin 2\pi f(t - s)$ (4)

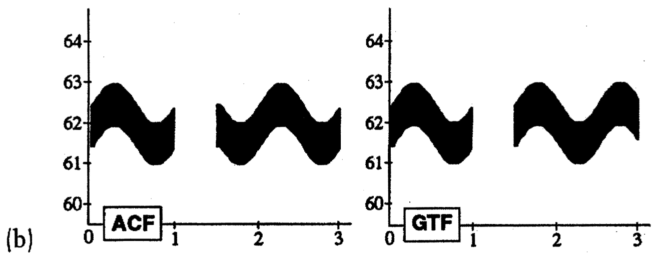
This seemingly small difference in implementation has an important effect on the workings of the systems. In the GTF definition of `oscillator` there are no free variables—the result is dependent on the function's formal parameters only—a purely *functional* style with no use of global data. This means that time functions can be bound or combined independent of the context in which they are actually used. In contrast, the ACF definition of `oscillator` has a reference to the free variable S (in fact, it can refer to any of the environment variables). Since this variable, in this case start-time S, can change depending on the context, the expression can likewise yield different results in different contexts. This is an *imperative* style, using state and assignment. In ACF, time functions must be defined in the context where they are actually used; one cannot, for example, abstract from them in one context but use them in another. In a functional language, one would expect the expression shown in Figure 14a to have the same semantics (i.e., graphical output) as those shown in Figures 14b and 14c, since it is a property of such a language that a name can only be associated with a value once. This property is called *referential transparency*. It is considered a severe loss when this property does not hold (Stoy 1977); for example, we can no longer be certain that $f(x) - f(x)$ is zero. Thus, reasoning about such programs becomes much harder because the whole mechanism of reasoning (the lambda calculus) is lost. In some cases it is important to drop this property, for example, in a non-deterministic programming style, but to give it up so early, at the fundamentals of what could become a basis for a representa-

Figure 14. Musical objects parameterized with time functions as expressions, and their graphical output as generated by the micro-versions of ACF and GTF, respectively (where they differ). A sequence of two notes separated by a rest and its identical output in both ACF and GTF is shown in (a); abstraction from the pitch parameter and its different output in ACF and GTF in (b); using local binding for the expression in (a) and its output in ACF and GTF is shown in (c).

```
(seq (note (oscillator 62 1 .5) 1 1)
     (pause .5)
     (note (oscillator 62 1 .5) 1.5 1)) ⇒
```



```
(a-musical-object (oscillator 62 1 .5)) ⇒
```



```
(let ((vibrato (oscillator 62 1 .5)))
  (seq (note vibrato 1 1)
       (pause .5)
       (note vibrato 1.5 1))) ⇒
```

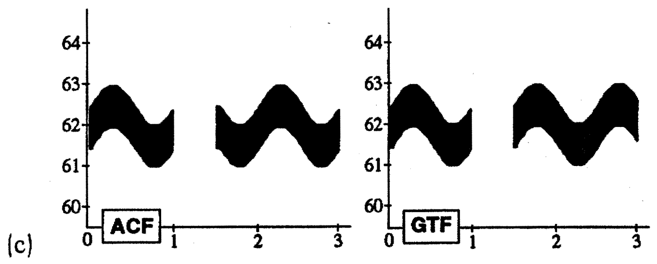
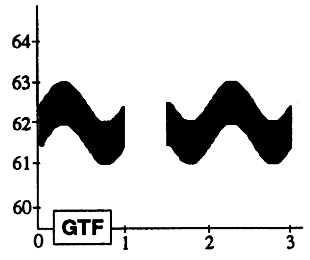


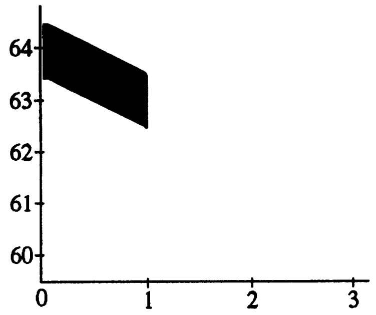
Figure 15. Using a GTF-specific language construct to attach a time function to the whole sequence (instead of to each individual note) to obtain the same result as the expression in Figure 14c given by ACF.

Figure 16. Transforming the constant pitch of a note of duration 1 with a linear interpolating ramp, resulting in a glissando that starts at 64 and ends at 63, which produces identical output in ACF and GTF.

```
; GTF-specific
(with-attached-gtfs ((vibrato (oscillator 62 1 .5)))
  (seq (note vibrato 1 1)
       (pause .5)
       (note vibrato 1.5 1))) ⇒
```



```
(trans (ramp 1 0) (note 63 1 1)) ⇒
```



tional system for music, seems to be a mistake. (Note that referential transparency is not a property of Lisp itself, since it combines functional with imperative language constructs.)

Since the ACF systems lack three central aspects of a functional language—a name is only once associated with a value (referential transparency), functions can be treated as values (first-class objects), and there are no side effects—we would not consider Arctic, Canon, Fugue, and Nyquist to be functional languages.

Attaching Time Functions to Musical Objects

Independent of these representation language design issues, we sometimes desire the behavior exhibited by ACF, in the sense that both time functions should refer to the same start time, as if they were linked to the whole object, instead of the individual notes. To do this properly, without conflicting with the referentially transparent `let` in Lisp, we need to introduce a new construct that is syntactically different. The macro `with-attached-gtfs` is an example of a construct providing such alternative semantics (see Figure 15 for an example) and the micro-version source file for the definition. It turns the expression in its body into a musical object generator, attaches the time functions mentioned to the start time and duration of the whole object (instead of using the start times and durations of the individual components, as is the default case), and communicates these "redirected" time functions to the places where they were mentioned in the expression.

With the `attach-gtf` construct, linking a time function's start and duration parameters to musical objects or values is generalized, i.e., time functions can be linked to any musical object, independent time point, or time interval. While the latter two situations are supported in ACF, time functions cannot be linked to musical objects. This is a second major difference between the two formalisms (More examples based on this difference will be given below in the section on Flexibility).

Modularity

Before we continue the comparison, as an exercise to the reader, try to decide whether the following two transformations should have a similar or different result in ACF and GTF: first, a transposition of a note by a declining glissando of a semitone that is then made twice as long; and second, a transposition of a note with the same declining glissando that was first made twice as long. This means, in Lisp, is

```
(stretch 2
     (trans (ramp 1 0)
            (note 63 1 1)))
```

the same as

```
(trans (ramp 1 0)
       (stretch 2
              (note 63 1 1)))?
```

Figure 17. Transforming the constant pitch of a note of duration 2 with a linear interpolating ramp yields a different result in ACF than in GTF.

```
(trans (ramp 1 0) (note 63 2 1)) ⇒
```

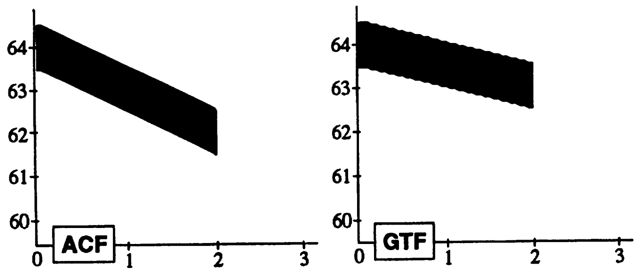
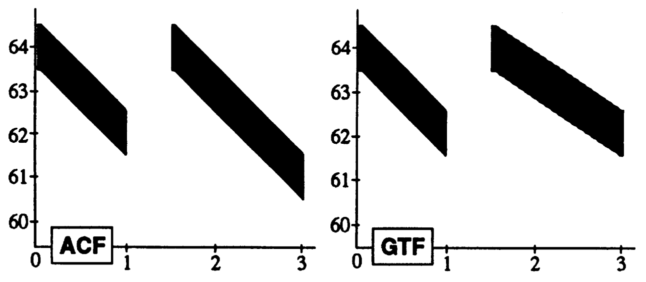


Figure 18. A sequence of two notes with different duration, separated by a rest, whereby each pitch attribute is associated with the same ramp time function. This produces a different output in ACF than in GTF.

```
(seq (note (ramp 64 62) 1 1)
     (pause .5)
     (note (ramp 64 62) 1.5 1)) ⇒
```



The answer will be given below. We will first look at a simpler sub-example, shown in Figure 16. In this case, the pitch of the note is transposed with a descending linear ramp, adding values to the note's constant pitch. Both ACF and GTF produce the same output.

However, when the note is made twice as long, by giving it duration 2, in ACF the shape of the ramp does not change, while in GTF it stretches along with the note's duration, i.e., the pitch of the note still starts at 64 and ends at 63 (see Figure 17).

This is not a bug, but a fundamental language-design decision. The difference in behavior is caused by what, again, seems to be a small difference in the two time-function definitions. In ACF, at define time a time function $\lambda(t)$ is "instantiated." It has access to the formal parameters of ramp and the implicit parameters of the transformation environment (S and F; see Equation 5 below). In GTF (Equation 6), ramp evaluates to a function of start time, duration, and time (i.e., $\lambda(s, d, t)$). This definition is independent of the transformations acting on the objects it might be linked to. Note that stretch factor F is not mentioned in Equation 6, while it is in Equation 5.

$$\text{ACF: } ramp(from, to, d) = \lambda(t)\, from + \frac{t-S}{Fd}(to - from) \quad (5)$$

$$\text{GTF: } ramp(from, to) = \lambda(s,d,t)\, from + \frac{t-s}{d}(to - from) \quad (6)$$

Furthermore, in ACF ramp has an extra parameter named duration (d) that must be explicitly communicated to the time function, while in GTF, in the default case, the time function is given the duration of the object that it is used for. So, to obtain the same output in ACF as in GTF for this example, ramp must be explicitly informed about the duration of the object it is used for (duration is underlined):

```
;;; ACF-specific
(trans (ramp 1 0 2) (note 63 2 1))
```

All functions of time in the ACF systems have this optional duration parameter (with the default being 1.0 sec). However, Arctic (Dannenberg, McAvinney, and Rubine 1986) elegantly works around this problem by introducing normalized durations—all time functions and behaviors must be explicitly stretched to obtain the desired duration. Another, more elaborate example is shown in Figure 18.

Here as well, to obtain the same output in ACF as in GTF, the durations of the individual notes have to be explicitly communicated to every function of time (duration is underlined):

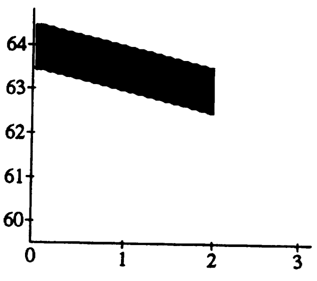
```
;;; ACF-specific
(seq (note (ramp 64 62 1) 1 1)
     (pause .5)
     (note (ramp 64 62 1.5) 1.5 1))
```

Finally, to come back to the question stated in the beginning of this section with regard to the effect of the order of applying transformations, Figure 19 shows that ACF and GTF give different results—for reasons just described.

The difference of having time functions that can be attached to musical objects (like in GTF), and time functions that are independent entities and are also sensitive to time transformations (as in

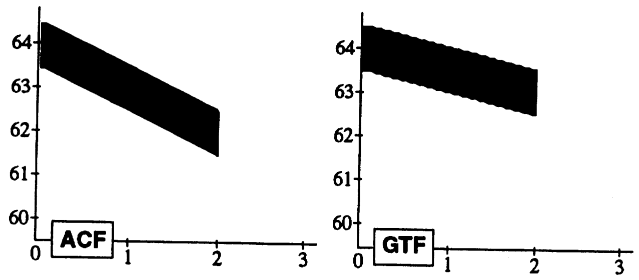
Figure 19. How order affects applying a `trans` *and* `stretch` *transformation to a note. Stretching a transposed note gives the same result in ACF and GTF (a); but transposing a stretched note gives a different result in ACF and GTF (b).*

```
(stretch 2 (trans (ramp 1 0) (note 63 1 1))) ⇒
```



(a)

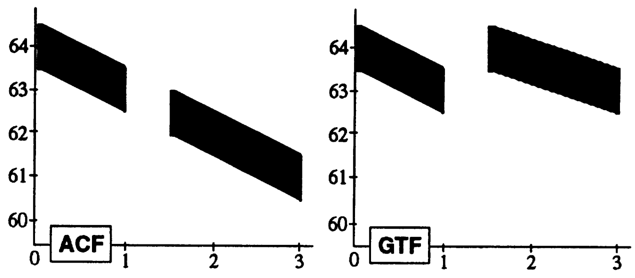
```
(trans (ramp 1 0) (stretch 2 (note 63 1 1))) ⇒
```



(b)

Figure 20. Sequence of two notes with a time function locally bound to the variable glissando, and its differing output in ACF and GTF.

```
(let ((glissando (ramp 64 63)))
  (seq (note glissando 1 1)
       (pause .5)
       (note glissando 1.5 1))) ⇒
```



ACF), indicates an important difference in modularity between the two formalisms. In GTF, time functions and transformations are orthogonal; the definition of one can be changed or extended without influencing the workings of the other. In ACF, time functions and transformations interact (for example, time functions are communicated a stretch factor—a time-transformation parameter). The issue of orthogonality will become crucial when the language is extended with, for example, time-varying time transformations (i.e., tempo or event-shift transformations using timing functions), because all behaviors must be modified to be able to work with these extensions.

Flexibility

Next, consider the output of the glissando example in Figure 20. For the same reasons as described for the example in Figure 14c, ACF and GTF give different results.

However, the point should be made here that despite the characteristics of a specific language, one sometimes wants to express one and sometimes the other behavior, i.e., time functions that are dependent or independent of musical objects. In GTF, the semantics of the ACF example can be obtained by defining a linear ramp that is independent of the duration of the object it is attached to—an `independent-ramp` (see Figure 21a).

But the `independent-ramp` in GTF is not the same as `ramp` in ACF. It still uses the start time of the object to which it is applied. While the time function constructor has a fixed decline/incline, it always starts at the same value at the object's start time (see Figure 21b). This is another behavior that might be preferable in some musical situations.

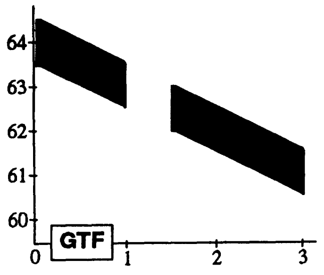
Yet another alternative is shown in Figure 21c. A `ramp` is linked here to the whole object, stretching along with its duration, such that it always starts at 64 and ends at 63.

The general point is that the issue is not deciding on correct semantics, but, instead, indicating how much flexibility we need to express a multitude of musically viable situations. Furthermore, the examples above make use of very simple time functions—without mechanisms to compose new time functions from existing ones, they will remain trivial examples. It is essential that we can abstract from them, building more musically realistic functions out of simpler ones that are well understood. As an example, assume we want to

Figure 21. Three GTF-specific examples of alternative ways to link a time function to musical objects. Attaching a linear ramp with its own independent duration (its third argument) to the whole sequence (identical to the output in ACF for the expression shown in Figure 20 for ACF) (a); parameterizing the individual notes with a ramp independent of the duration of the object it is used for (it starts at 64 for every note, but then has a fixed decline) (b); and attaching a ramp to the whole sequence, resulting in a glissando over the entire object starting at 64 and ending at 63 (c).

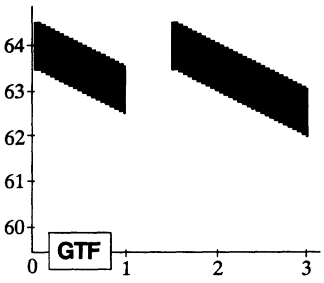
Figure 22. Output of the user-defined function `example` *that links a composite time function, constructed from a ramp starting at 64 and ending at 63 over the duration of the whole sequence, and an oscillator attached to each individual note (a). It displays the correct behavior when stretched as a whole: the glissando is compressed (but still starts at 64 and ends at 63), while the vibrato component drops some periods, depending on the new durations of the individual notes (b).*

```
; GTF-specific
(with-attached-gtfs
   ((glissando (independent-ramp 64 63 1)))
  (seq (note glissando 1 1)
       (pause .5)
       (note glissando 1.5 1))) ⇒
```



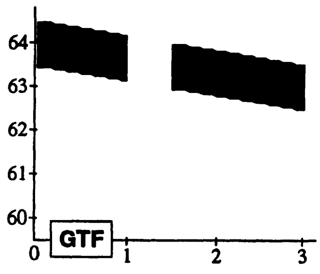
(a)

```
; GTF-specific
(let ((glissando (independent-ramp 64 63 1)))
  (seq (note glissando 1 1)
       (pause .5)
       (note glissando 1.5 1))) ⇒
```



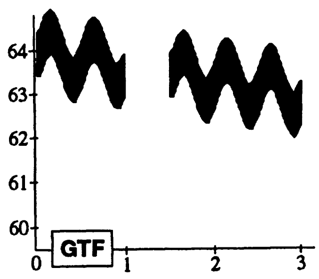
(b)

```
; GTF-specific
(with-attached-gtfs ((glissando (ramp 64 63)))
  (seq (note glissando 1 1)
       (pause .5)
       (note glissando 1.5 1))) ⇒
```



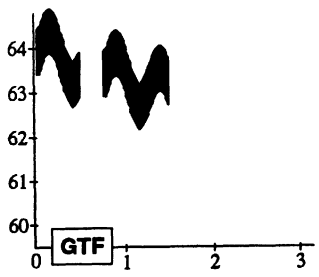
(c)

```
(example) ⇒
```



(a)

```
(stretch .5 (example)) ⇒
```



(b)

define a time function that embodies glissandi with a little vibrato, a simplistic first step in the direction of expressing the musical knowledge used in singing. This is shown in Figure 22. We can compose a glissando with a vibrato by adding the results of a `ramp` that is linked to the whole musical object, and an `oscillator` time function that is linked to the individual components of the musical object (all this without having to refer to the internal structure of `a-musical-object`).

```
;;; GTF-specific
(defun example ()
  (with-attached-gtfs ((glissando (ramp 64 63)))
    (let* ((vibrato (oscillator 0 2 .5))
           (pitch (time-fun-+ glissando vibrato)))
    (a-musical-object pitch))))
```

Programs Versus Data

In ACF, all behaviors and transformations are programs, and the output is generated as a side-effect. Functions of time are also, in a sense, behaviors that can inspect the transformation environment (In Arctic there is indeed no distinction between time functions and behaviors; all behaviors are, in fact, functions of time). This implies that if one wants to add a MIDI play function or a graphical extension, all behaviors must be modified—a tedious job in a large-scale system (see Dannenberg, Fraley, and Velikonja 1991). In GTF, all objects of the language (musical objects, transformations, and time functions) are first-class objects, as they deliver data structures and can be bound and passed as arguments. They can therefore be inspected by other programs or serve as input to other systems (for example, graphics- or sound-generation systems).

This data-versus-programs distinction also has an important influence on the expressiveness of the representation itself. For instance, in the case of a language with musical objects as procedures, there is no access to these objects after definition. This forces all communication from, for example, time functions to musical objects and vice versa, to be realized at define time. Representation problems that can be characterized as based upon "bottom-up" or "lateral" communication, dependent on the accessibility of musical objects after definition, cannot be represented in such languages (see the "compressor problem" and "transition problem" in Desain and Honing 1993).

Conclusion

In this study, two formalisms for describing functions of time were compared using micro-version programs as a means to gain insight in their workings. Although both systems provide a solution to the vibrato problem—in that they acknowledge the need for more time information besides actual time—several important semantic differences were indicated. These differences were shown to be intrinsic to the design of the two systems and in the way they support notions such as abstraction, flexibility, and extensibility.

This article is restricted to the vibrato problem, which reflects just a minor aspect of a representational system for music. Transformations and musical objects—their construction, structuring, and use—are not discussed. Other, more pragmatic issues, including efficiency and real-time possibilities, are also left untouched. The aim, though, is to achieve a true understanding of what seems to be irrelevant differences between two relatively simple formalisms. This understanding is essential, for instance, in choosing a formalism as a fundamental building block of a more elaborate representation system for music. Finally, the vibrato problem is a key example of the kind of expressive power that we need for the next generation of synthesizers that allow high-level musical control; for example, synthesis methods based on physical models (Smith 1992), or revitalized additive synthesis (Serra and Smith 1990).

Acknowledgments

The author wishes to express his special thanks to Roger Dannenberg for his open and collaborative attitude, providing full access to his systems; the article benefited greatly from comments he made on earlier versions. Peter Desain is to be thanked for helping out at crucial stages of this research and for greatly improving its presentation. Also, thanks to Huub van Thienen for his detailed comments on an earlier draft. None of them, of course, necessarily subscribes to any of my conclusions. Remko Scha and the Computational Linguistics Department of the University of Amsterdam are thanked for providing the environment in which this research could evolve.

Part of this work benefited from a travel grant by Netherlands Organization for Scientific Research (NWO) while visiting the Center for Computer Research in Music and Acoustics, Stanford University, on kind invitation of Chris Chafe and John Chowning. The author's research has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences (KNAW).

References

Abelson, H., and G. Sussman. 1985. *Structure and Interpretation of Computer Programs.* Cambridge, Massachusetts: MIT Press.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1).

Dannenberg, R. B. 1993. "The Implementation of Nyquist, A Sound Synthesis Language." *Proceedings of the 1993 International Computer Music Conference.* San Francisco: International Computer Music Association.

Dannenberg, R. B., C. L. Fraley, and P. Velikonja. 1991. "Fugue: A Functional Language for Sound Synthesis." *IEEE Computer* 24(7).

Dannenberg, R. B., P. McAvinney, and D. Rubine. 1986. "Arctic: A Functional Language for Real-Time Systems." *Computer Music Journal* 10(4).

De Poli, G., A. Piccialli, and C. Roads, eds. 1991. *Representations of Musical Signals.* Cambridge, Massachusetts: MIT Press.

Desain, P., and H. Honing. 1992a. "Time Functions Function Best as Functions of Multiple Times." *Computer Music Journal* 16(2). Reprinted in P. Desain and H. Honing 1992b.

Desain, P., and H. Honing. 1992b. *Music, Mind and Machine: Studies in Computer Music, Music Cognition and Artificial Intelligence.* Amsterdam, The Netherlands: Thesis Publishers.

Desain, P., and H. Honing. 1993. "On Continuous Musical Control of Discrete Musical Objects." *Proceedings of the 1993 International Computer Music Conference.* San Francisco: International Computer Music Association.

Friedman, D. P., M. Wand, and C. T. Haynes. 1992. *Essentials of Programming Languages.* Cambridge, Massachusetts: MIT Press.

Henderson, P. 1980. *Functional Programming. Application and Implementation.* London: Prentice-Hall.

Honing, H. 1993a. "A Microworld Approach to the Formalization of Musical Knowledge." *Computers and the Humanities* 27.

Honing, H. 1993b. "Issues in the Representation of Time and Structure in Music." In I. Cross and I. Deliège, eds. "Music and the Cognitive Sciences." *Contemporary Music Review.* London: Harwood Press. Pre-printed in P. Desain and H. Honing 1992b.

Richie, G. D., and F. K. Hanna. 1990. "AM: A Case Study in AI Methodology." In D. Partridge and Y. Wilks, eds. *The Foundations of Artificial Intelligence. A Source Book.* Cambridge, UK: Cambridge University Press.

Serra, X., and J. O. Smith. 1990. "Spectral Modeling Synthesis: A Sound Analysis System Based on a Deterministic plus Stochastic Decomposition." *Computer Music Journal* 14(4).

Smith, J. O. 1992. "Physical Modeling Using Digital Waveguides." *Computer Music Journal* 16(4).

Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Cambridge, Massachusetts: MIT Press.