Temporal semantics for a live coding language

Samuel Aaron Dominic Orchard Alan F. Blackwell
Computer Laboratory, University of Cambridge, UK
firstname.lastname@cl.cam.ac.uk

Abstract

Sonic Pi is a music live coding language that has been designed for educational use as a first programming language. However, it is not straightforward to achieve the necessary simplicity of a first language in a music live coding setting, for reasons largely related to the manipulation of time. The original version of Sonic Pi used a 'sleep' function for managing time, blocking computation for a specified time period. However, while this approach was conceptually simple, it resulted in badly timed music, especially when multiple musical threads were executing concurrently. This paper describes an alternative programming approach for timing (implemented in Sonic Pi v2.0) which maintains syntactic compatibility with v1.0, yet provides accurate timing via interaction between real time and a "virtual time". We provide a formal specification of the temporal behaviour of Sonic Pi, motivated in relation to other recent approaches to the semantics of time in live coding and general computation. We then define a monadic model of the Sonic Pi temporal semantics which is sound with respect to this specification, using Haskell as a metalanguage.

Categories and Subject Descriptors H.5.5 [*Information Interfaces and Presentation*]: Sound and Music Computing; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.3.2 [*Programming Languages*]: Applicative (functional) languages; J.5 [*Arts and Humanities*]: Music

Keywords time; music; live coding; temporal semantics; monads

1. Introduction

Timing is a critical component of music. Therefore any language for describing music must have a method for describing the precise timing of sounds, such as individual notes. Performing a piece of music correctly then amounts to a computation, evaluating the musical description to emit correct notes at correct times. This kind of timing contrasts with many notions in computing. For example, "real-time computing" approaches often focus on computing within a certain time limit (a deadline), thus high-performance is important. But in music, being early is just as bad as being late. A similar situation arises in mechanical coordination tasks, such as programming robotic limbs for walking. For these kinds of application, a robust programming model for timing is required. We argue that our Sonic Pi language provides a suitable, robust temporal model for music in the context of live programming and education.

Sonic Pi is a domain specific language for manipulating synthesisers through time [AB13]. It was designed for teaching core computing concepts to school students using creative programming, specifically live-coding music, as a means for engaging students. Sonic Pi is a mostly pure language, with first-class functions. Its impurity arises from timing and output effects (for producing sounds). The precise timing of effects, which do not occur too early or too late, is core to the programming approach of Sonic Pi. Primarily, this paper introduces the temporal programming model of Sonic Pi. We give a monadic description of its effects, showing that the impure parts of the language can be embedded in a pure language.

As well as the need for programming approaches to time, there is a well-recognised need for models of temporal behaviour coupled with reasoning systems for time. This has been explored particularly in logic, with modal logics such as the Real-Time Computation Tree Logic [EMSS91]. In the literature, Lee makes a powerful argument for the development of a semantics of time in computation, or as he describes it, a properly formalised class of "time system" that can be applied alongside type systems (which are already understood to be essential software engineering tools) [Lee09]. It is in this spirit that we develop two kinds of model for the temporal semantics of Sonic Pi: a *time system* and a denotational model.

The core contributions of this paper are three-fold:

- We present a new programming approach for precisely timing effects, which is implemented as part of the Sonic Pi language for music live coding. We explain how this programming approach has evolved to replace the previous version of the Sonic Pi language (Section 2), providing a syntactically identical language but with an improved approach to timing (Section 3).
- We formalise the temporal semantics of this approach, introducing a specification of the temporal behaviour of a core subset of Sonic Pi programs: a *time system*, which provides a static analysis of timing (Section 4). The style is axiomatic, and can be considered an abstract model of temporal behaviour.
- We give a monadic denotational semantics to a core subset language (Section 5) and prove it sound with respect to the time system, *i.e.*, the language is *time safe*. We later extend this model to include temporal warnings (Section 6). The denotational approach complements the abstract time system model.

We use the phrases *time system* and *time safety* to draw analogy with traditional notions of *type system* and *type safety*.

We begin with a discussion of the first programming language and live coding contexts (particularly for music), as these aspects motivate the language design. Readers who are keen to get to the language design may skip over this discussion to Section 2.

FARM '14, September 6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3039-8/14/09. . . $15.00.
http://dx.doi.org/10.1145/2633638.2633648

```
play 52
play 55
play 59
```

(a) Successive notes

```
play 52
sleep 1
play 55
sleep 1
play 59
```

(b) Notes separated by sleeps

Figure 1. Playing MIDI notes of an E minor chord in Sonic Pi v1.0

1.1 The first language and live coding contexts

A first programming language should be conceptually simple and syntactically uncluttered. However, achieving this simplicity in a music live coding language is not easy for reasons largely related to the representation of time. Representing musical time in a programming language is often problematic, firstly because natural ways of describing and organising musical events are multi-threaded (scores, chords, resonance, reverb), and secondly because so many standard computational formalisms treat execution time as a non-functional requirement [Lee09]. Time can be even more problematic in live coding, because the creation of the code, as a performance, is interleaved with the sound events resulting from its execution. Yet for users of Sonic Pi, the creative experience they have, like all experience, arises through time – as media theorist Mieke Bal says, "*time is where subjectivity is produced*" [Bal02].

As noted by Rorhruber [BMNR14], there have been many publications discussing alternative approaches to the representation of time in live coding, choosing between explicit or implicit representation of time and between description of time with reference to internal or external state. These many interesting strategies include McLean's formalism of cyclic time in the Tidal language [McL13], and Sorensen's temporal recursion in Impromptu/Extempore [SG10]. In this paper, we present a formalism that is designed to achieve production-quality sound (via the SuperCollider synthesis server) while allowing inexperienced programmers in an educational setting (typically 11-12 year-old children) to express the temporal structure in terms that have an intuitive correspondence to the experience and production of music.

In music, it is clear that we must be able to speak about the precise location of events in time, and hence that any music programming language must of necessity provide some kind of time semantics, even if these are only informal. In the case of live coding languages, an additional consideration is that the time at which the program is edited may coincide or overlap with the time at which it is executed. This overlap between execution and creation time is of broader value in software engineering, as noted for example by McDirmid [ME14], whose Glitch system allows the user to adjust the notional execution time relative to a point in the source code editing environment. Tools of this kind can also benefit from a formal semantics in which to define the relationship between changes or navigation within the code, and changes or navigation within the cause-effect sequence of execution time.

2. Problems with timing in Sonic Pi v1.0

Sonic Pi was designed to teach a large number of computing concepts covered in the new UK computing curriculum introduced in September, 2014. Examples of these concepts are conditionals, iteration, variables, functions, algorithms and data structures. We also extend beyond these to provide educators with an opportunity to introduce concepts which we believe will play an increasingly important role in future programming contexts such as multi-threading and hot-swapping of code.

One of the core UK computing curriculum concepts that Sonic Pi immediately focusses on is the sequential ordering of effects in imperative programs, such as playing successive notes see Figure 1(a) (which is considered here to be a Sonic Pi v1.0 program).

```
loop do
  play 30                  #A
  sample :drum_heavy_kick  #B
  sleep 0.5                #C
end
```

Figure 2. A continuously repeating bass and drum hit.

```
in_thread
  loop do
    play 30
    sleep 0.5
  end
end
in_thread
  loop do
    sample :drum_heavy_kick
    sleep 1
  end
end
```

Figure 3. Two concurrent threads playing in synchronisation.

Sonic Pi v1.0 takes advantage of the fast clockspeeds of modern processors in assuming that the sequence of instructions of Figure 1(a) are likely to be executed so quickly in succession that they will be perceived as a chord *i.e.*, all the note being played simultaneously, rather than as successive notes in an *arpeggio* form. In order to further separate the instructions in time such that their separation may be perceived it is necessary to insert explicit timed delays. This can be achieved by "sleeping" the current thread for a number of seconds, see Figure 1(b). This notion of sleep is similar to that of the standard POSIX sleep operation that suspends execution for the specified time [IG13].

These temporal semantics worked well in a computing education context for demonstrating effect execution order, but they do not translate well to music contexts due to a mismatch with user expectations; they do not allow correct timing of musical notes. This mismatch was emphasised when Sonic Pi gained the ability to play drum samples. Consider the example in Figure 2. Here we are attempting to regularly play MIDI note 30 at the same time as the sample `:drum_heavy_kick` with half a second between each onset. Unfortunately the execution will not produce the desired behaviour and the rhythm will drift in time due to the addition of the execution time itself to the sleep time. For example, after line A in Figure 2 has completed execution, the clock time will have moved on by the amount of time it took to execute the line. Similarly for line B. Line C introduces two extra sources of time, the sleep time and the time spent waiting for the scheduler to pick up and continue executing the thread. Therefore, instead of each iteration taking precisely 0.5s, the actual time is the summation of the computation time of A, the computation time of B, 0.5 and the scheduler pick-up time. Depending on load and processor speed, these extra times can produce dramatically noticeable results. This is profoundly apparent when the user requests two threads to work in synchronisation such as in Figure 3. The threads may start out in synchronisation, but because the extra computation time will differ across the threads, they will drift at varying rates and move out of synchronisation.

Sonic Pi's timing issues are further compounded by the fact that calls to `play` and `sample` are asynchronous messages, and there is an additional time cost for these messages to be sent and interpreted by the external synth process. This then adds additional varying time (jitter) to each sound trigger.

The temporal issues described in this section are summarised in Figure 4, which describes the timing behaviour of Sonic Pi v1.0 code triggering three successive chords. Each of the Δ times in the far left column represents the real computation time of each statement. Notice how they are all unique. The precise duration is related to aspects such as the amount of processing required for

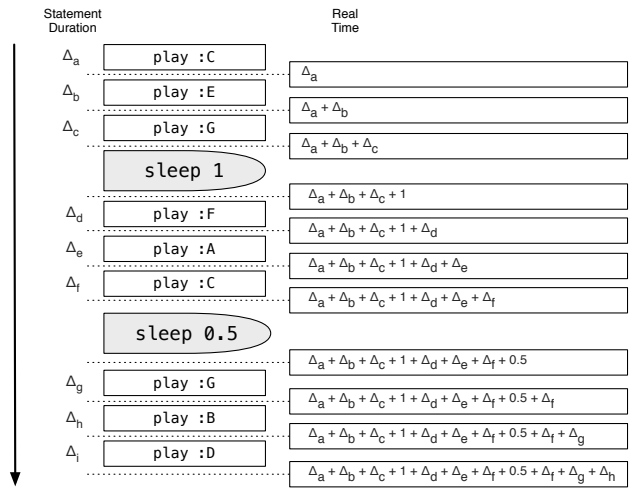

Figure 4. The timing behaviour in Sonic Pi v1.0

the computation, the current load of the system and the processor speed. The duration of Δs is therefore nondeterministic and will not be consistent across runs of the same program. As Figure 4 illustrates, the actual run time of the program is a summation of all these Δ times in addition to the requested sleep durations:

$$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f + 0.5 + \Delta_g + \Delta_h + \Delta_i$$

This results in both drift and jitter of the timing of the sounds produced by the program.

2.1 Temporal expectations

When users create programs in Sonic Pi, the ease with which they can produce the musical effects they intend is dependent on their expectations of the code's behaviour. As described by Honing [Hon93] (see Section 7), music systems may represent temporal structure either explicitly (describing time intervals and relations) or implicitly (in an ordered sequence of notes having different durations). Musical scores provide an implicit time representation, while most music programming systems rely on explicit time representation. Unfortunately, in the case of general purpose programming languages, the typical implementation of the sleep operator supports an explicit representation of rhythm that is guaranteed to be accurate only in the ordering of the notes, not in the elapsed time between them. In teaching programming, the usual focus is on correctness of this explicitly specified behaviour. When teaching programming through the medium of music, as in Sonic Pi, the musical expectations that are usually associated with implicit representation of rhythm mean that non-expert musicians are likely to hear that something is wrong, while not being able to express precisely what the problem is.

Less expert musicians might be able to identify more explicit problems (such as extra beats), but find it harder to say precisely what the problem is when that problem is related to their implicit expectations. Even if the user can perceive the timing mistakes, the language provides no means to fix them. One of the goals of Sonic Pi is to to create a system that is useful to experienced musicians (with clear musical goals) and acceptable to inexperienced musicians that may not be able to clearly articulate what they want to achieve, but know when it is wrong.

It is therefore important to maintain the conceptual simplicity of the original approach, while providing an improved time semantics that satisfies not only explicit expectations of the musical listener, but also these implicit expectations.

```
play :C ; play :E ; play :G
sleep 1
play :F ; play :A ; play :C
sleep 0.5
play :G ; play :B ; play :D
```

Figure 5. Playing three chords (C major, F major, G major) in Sonic Pi v2.0, with the second two played closer together by $0.5s$.

3. Reinventing sleep

Sonic Pi v2.0 introduces a new implementation and semantics of the sleep command which maintains syntactic and conceptual compatibility with the previous implementation yet modifies the temporal semantics to match the implicit rhythmical expectations previously described. The semantics is no longer similar to that of the POSIX sleep command. The underlying programming model of Sonic Pi v2.0 provides a way to separate the ordering of effects from the timing of effects. Figure 5 shows the program that was used in Figure 4, but we now treat it as Sonic Pi v2.0 program. This example program (playing three chords in sequence) combines simple notions of parallel, timed, and ordered effects.

The first three statements play the notes of a C major chord in parallel. A `sleep` statement then provides a "temporal barrier" which blocks the computation from continuing until 1 second has elapsed since the *start* of the program (not since the end of playing the notes). Once one second has elapsed, the next three statements are executed, which play an F major chord. The next `sleep` means that the final chord is not played until 1.5 seconds has elapsed since the start of the program. Figure 6 illustrates the timing. Thus, `sleep` t communicates that, after it has been evaluated, at least t seconds has elapsed since the last `sleep`. This provides a minimum time. In between calls to `sleep`, any other statements can (with some limits) be considered task parallel. The semantics of `sleep` is similar to the interaction of time and the (multiply) overloaded `=>` operator in the live coding language ChucK [WC03].

These semantics are achieved by introducing a notion of *virtual time* as a thread-local variable which is only advanced by the new `sleep` operation. Each thread has access to both real time and virtual time, with the virtual time used to schedule external effects. In order to keep the execution of the program in synchronisation with the explicit timing requirements of the program, `sleep` takes into account the execution time since the last `sleep` and reduces the requested sleep time appropriately. Therefore if the user issues a `sleep 1` statement, and the current execution time since the last `sleep` statement is 0.1 seconds, the implementation only sleeps the current thread for 0.9s. This ensures that no drifting occurs.

Figure 6 demonstrates the timing of the Figure 5 program in Sonic Pi v2.0, which contrasts with the timing diagram in Figure 4. The overall elapsed time for the program is now:

$$(v2.0) \qquad 1.5 + \Delta_g + \Delta_h + \Delta_i$$

which contrasts with the Sonic Pi v1.0 timing for the same program:

$$(v1.0) \quad 1.5 + \Delta_a + \Delta_b + \Delta_c + \Delta_d + \Delta_e + \Delta_f + \Delta_g + \Delta_h + \Delta_i$$

This shows that we have eliminated drift in Sonic Pi v2.0 since the only overhead is now just the overhead of the `play` statements following the last `sleep`. For Sonic Pi v1.0, each block of `play` statements adds overhead, leading to timing drift over the course of a program. In Section 4 we will make precise the behaviour of the new sleep operation.

In order to deal with relative execution times within a sleep barrier, *e.g.*, the `play :C ; play :E ; play :G` operations in Figure 5, and also to accommodate the cost of scheduling output effects (to the synthesiser serve), a constant `scheduleAheadTime`

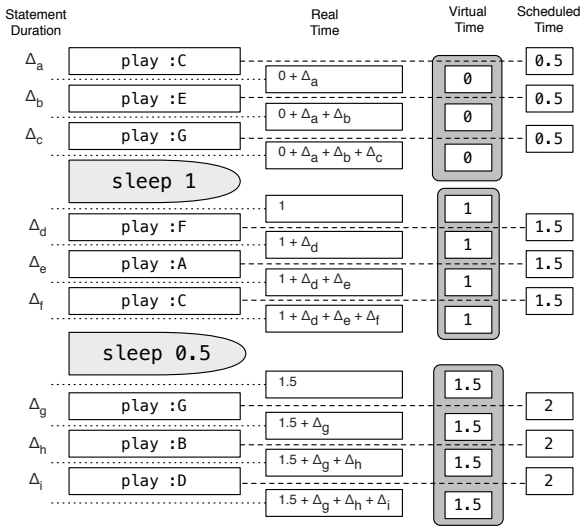

Figure 6. Timing behaviour of Sonic Pi v2.0 including virtual and scheduled time with a `scheduleAheadTime` of 0.5.

value is added to the current virtual time for all asynchronously scheduled effects. Provided that the addition of the jitter time and the execution time between calls to `sleep` never exceeds this value, the temporal expectations of the system are met.

It is possible that a computation preceding a `sleep` can overrun; that is, run longer than the sleep time. Thus, the programming model is not suitable for realtime systems requiring hard deadlines but `sleep` instead provides a *soft deadline* (in the terminology of Hansson and Jonsson [HJ94]). However, if a given thread falls behind, the user receives explicit timing warnings (described further in Section 6). Finally, if the thread falls further behind by a user-specifiable amount of time then Sonic Pi will stop that thread by throwing a time exception. This therefore not only provides essential information to users about the temporal behaviour of the program but also serves as a safety mechanism against a common class of errors such as placing an isolated call to `play` within a loop with no calls `sleep`. In such cases, the thread will no longer permanently sit in a tight loop consuming all resources, but will self-terminate allowing any other threads to continue executing normally.

3.1 Examples

Figure 7 shows four similar programs which each have different internal behaviours for `sleep`, illustrating its semantics. We use the function `kernelSleep`, which is not a standard part of the Sonic Pi language, as a placeholder to represent a computation lasting a particular length of time (as specified by the parameter to `kernelSleep`). The first three example programs take 3s to execute and the last takes 4s to execute, with the behaviours:

(a) 3s – sleeps for 1s then sleeps for 2s (two sleeps performed);

(b) 3s – performs a computation lasting 1s, ignores the first `sleep` since its minimum duration has been reached, and then sleeps for 2s (one sleep performed);

(c) 3s – performs a computation lasting 2s, which means that the first `sleep` is ignored, and the second `sleep` waits for only 1s to reach its minimum duration (half a sleep performed);

(d) 4s – performs a computation lasting 2s, thus the first `sleep` is ignored, then performs a computation lasting 2s, thus the second `sleep` is ignored (no sleeps performed).

```
sleep 1
sleep 2
```
(a) Two sleeps

```
kernelSleep 1
sleep 1
sleep 2
```
(b) One sleep

```
kernelSleep 2
sleep 1
sleep 2
```
(c) Half a sleep

```
kernelSleep 2
sleep 1
kernelSleep 2
sleep 2
```
(d) No sleeps

Figure 7. Example programs with different `sleep` behaviours

4. A *time system* for Sonic Pi

From our experiences, we've found that the programming model of Sonic Pi, particularly its temporal model, is easy to understand by even complete beginners, including children. By a few simple examples it is easy to demonstrate the temporal semantics, using sounds as output, without having to appeal to any meta-theory; Sonic Pi attains the goal of being a good first language.

In this section, we approach the programming model of Sonic Pi from a more theoretical angle, in order to develop a specification of our programming model that can be reused for other applications and languages outside of the Sonic Pi context. From our model we prove a number of core properties of Sonic Pi as well. It is in no way necessary for programmers of Sonic Pi to understand this theory, but the contribution here is useful for future language design and implementation research.

Firstly, we define an abstract specification of virtual time and actual elapsed time in a simple core subset of Sonic Pi (Section 4.1). This gives an abstract, axiomatic model of the semantics which we call a *time system*. This model is made more concrete by providing a denotational-style, monadic semantics in the next section (Section 5), introducing the *temporal monad*. We prove the monadic model sound with respect to the initial axiomatic specification, up to small permutations in time delay (Section 5.3).

Terminology and notation We refer to sequences of statements as *programs*. Throughout, P, Q range over programs, and s, t range over times (usually in seconds).

A core fragment of Sonic Pi In the rest of this paper, we model a core subset of the Sonic Pi v2.0 language with the following grammar, where P are programs, S statements, and E expressions:

$$P ::= P; S \mid \emptyset$$
$$S ::= E \mid v = E$$
$$E ::= \texttt{sleep}\ \mathbb{R}_{\geqslant 0} \mid A^i \mid v$$

where A^i represents operations (actions) in Sonic Pi other than `sleep`, *e.g.*, some A^j is the `play` operation. We use this to abstract over operations in the language which do not modify virtual time.

By the above definition, programs P are a "snoc"-list (*i.e.*, elements are "consed" onto the end, not front as is standard for inductively-defined lists) where $\emptyset$ is the empty list. Equivalently, sequential composition of statements is syntactically left-associated. This structure aids later proofs since it allows inductive reasoning on a statement of a program and its preceding program, which is key to accurately modelling `sleep`.

Statements S may be expressions on their own, or may have (pure) bindings to variables. Throughout we consider the first case of S as a degenerate case of the second where the variable is irrelevant *i.e.*, $\_ = E$ where $\_$ denotes a wildcard variable.

We'll add the previously seen `kernelSleep` operation to the core subset here, which blocks the current computation for the time specified by its parameter, *i.e.*, it has the semantics of POSIX *sleep*. This operation is not available in the actual language, but it is useful for examples and contrasting with `sleep`.

This core subset is a zero-order language, in the sense that we do not include the definition or calling of user-defined functions; nor

do we incorporate the threading constructs provided by Sonic Pi. Extending the model here to include these is however straightforward, but we stick with a simple language for the sake of succinctly introducing and reasoning about the core temporal behaviour.

4.1 Virtual time and real time

As described previously, the programming model of Sonic Pi distinguishes between the actual time elapsed since the start of a program P which we write here as $[P]_{\mathsf{t}}$ and the virtual time which is advanced by `sleep` statements which we write as $[P]_{\mathsf{v}}$. Both these abstract functions return time values, thus, $[-]_{\mathsf{v}}, [-]_{\mathsf{t}} \in \mathbb{R}_{\geqslant 0}$, *i.e.*, both return positive, real-number values.

In this section, we give specifications to $[-]_v$ and $[-]_t$ providing an axiomatic model of Sonic Pi's temporal behaviour.

Virtual time $[-]_{\mathsf{v}}$ can be easily defined over all programs, statements, and expressions, since the `sleep` operation is the only expression changing virtual time:

Definition 1. Virtual time is specified for statements of Sonic Pi programs by the following cases:

$$[P; v = E]_{\mathsf{v}} = [P]_{\mathsf{v}} + [E]_{\mathsf{v}} \qquad [\texttt{sleep } t]_{\mathsf{v}} = t \qquad [v]_{\mathsf{v}} = 0$$
$$[\emptyset]_{\mathsf{v}} = 0 \qquad [A^i]_{\mathsf{v}} = 0$$

We therefore overload $[-]_{\mathsf{v}}$ to programs, statements, and expressions. Anything other than `sleep` or sequential composition has the virtual time is 0. Note that the equations on the left define $[-]_{\mathsf{v}}$ for programs (with statements covered by the single case for $P; v = E$), and on the right for expressions.

Equality on time Providing exact deadlines in real-time systems is difficult due to non-determinism combined with execution overheads. We do not ignore this problem in the programming model of Sonic Pi and the discussion here. We define the relation $\approx$ on actual times, where:

$$\forall s, t. \quad s \approx t \;\equiv\; |(s - t)| \leqslant \epsilon \tag{1}$$

for some value of ϵ which is the maximum negligible time value with respect to the application at hand. For example, if $\epsilon = 0.1$ then $3 \approx 3.05 \approx 2.92$.

In the case of Sonic Pi, we mitigate any ϵ-time differences by scheduling calls to the synthesise server using the current virtual time (see the diagram of Figure 6). Later in the denotational model (Section 5), we'll demonstrate sources of temporal variations ϵ, which are limited to a very small part of the model. Crucially, these ϵ time differences do not accumulate– the `sleep` operation provides a barrier which prevents this.

Axioms of actual time The virtual time and actual time of a single sleep statement are roughly the same, *i.e.*, $[\texttt{sleep } t]_{\mathsf{v}} \approx [\texttt{sleep } t]_{\mathsf{t}}$ and thus $[\texttt{sleep } t]_{\mathsf{t}} \approx t$ (by the specification in Definition 1). This holds only when `sleep` is used in isolation, that is, when it is the only statement in a program. As shown by the examples of Section 3.1, the use of `sleep` t in a program does not mean that a program necessarily waits for t seconds– depending on the context, it may wait for anywhere between 0 and t seconds.

Definition 2. The actual elapsed time $[-]_{\mathsf{t}}$ can be (partially) specified at the level of programs by the following equations:

$$[\emptyset]_{\mathsf{t}} \;\approx\; 0$$
$$[P; \texttt{sleep } t]_{\mathsf{t}} \;\approx\; ([P]_{\mathsf{v}} + t) \;\mathit{max}\; [P]_{\mathsf{t}}$$
$$[P; v = A^i]_{\mathsf{t}} \;\approx\; [P]_{\mathsf{t}} + [A^i]_{\mathsf{t}}$$

In the case of $A^i = \texttt{kernelSleep}$, then $[\texttt{kernelSleep } t]_{\mathsf{t}} = t$.

Example 1. The following two small example programs illustrate this definition, both of which have actual time 2 but arising from different calls to `sleep` and `kernelSleep`.

– $[\texttt{kernelSleep 2; sleep 1}]_{\mathsf{t}} \approx 2$

where $P = \texttt{kernelSleep 2}$, $[P]_{\mathsf{v}} = 0$, $t = 1$, and $[P]_{\mathsf{t}} = 2$, thus $([P]_{\mathsf{v}} + t) < [P]_{\mathsf{t}}$

– $[\texttt{kernelSleep 1; sleep 2}]_{\mathsf{t}} \approx 2$

where $P = \texttt{kernelSleep 1}$, $[P]_{\mathsf{v}} = 0$, $t = 2$, and $[P]_{\mathsf{t}} = 1$, thus $([P]_{\mathsf{v}} + t) > [P]_{\mathsf{t}}$

Definition 2 illuminates the semantics of `sleep`, showing the interaction between actual $[-]_{\mathsf{t}}$ and virtual time $[-]_{\mathsf{v}}$ in the case for `sleep`. In this case, the definition of $[-]_{\mathsf{t}}$ is not a straightforward recursive decomposition on programs, statements, and expressions as in the definition of $[-]_{\mathsf{v}}$. Instead, the actual time of a `sleep` depends on its *context*, which is the pre-composed (preceding) program P and its actual time $[P]_{\mathsf{t}}$. This is why we have structured the core subset language here in terms of "snoc"-list since the temporal semantics of an individual statement can depend on the program that has *come before* it (the tail of the "snoc"-list). Thus, the syntactic structure here facilitates the modelling of `sleep` and subsequent proofs on program properties (coming up next).

The specifications on $[-]_{\mathsf{v}}$ and $[-]_{\mathsf{t}}$ provide the following lemma about the temporal semantics of Sonic Pi programs:

Lemma 1. *For any program P then $[P]_{\mathsf{t}} \geqslant [P]_{\mathsf{v}}$.*

That is, the actual running time of a program is always at least the virtual time; a Sonic Pi program never "under runs" its virtual time specification.

Proof. By induction on the structure of programs.

- $P = \emptyset$. Trivial since $[\emptyset]_{\mathsf{v}} = 0$ by Definition 1.
- $P = (P'; v = E)$, split on E
  - $E = \texttt{sleep } t$
    (a) by Definition 1, $[P'; \texttt{sleep } t]_{\mathsf{v}} = [P']_{\mathsf{v}} + t$.
    (b) by Definition 2, $[P'; \texttt{sleep } t]_{\mathsf{t}} = ([P']_{\mathsf{v}} + t) \;\mathit{max}\; [P']_{\mathsf{t}}$.
    (c) by (b) $(([P']_{\mathsf{v}} + t) \;\mathit{max}\; [P']_{\mathsf{t}}) \geqslant [P']_{\mathsf{v}} + t$
    $\therefore$ by (a) and (c) then $[P'; \texttt{sleep } t]_{\mathsf{t}} \geqslant [P' \texttt{sleep } t]_{\mathsf{v}}$
  - otherwise $E = A^i$
    (a) by Definition 1, $[P'; v = A^i]_{\mathsf{v}} = [P']_{\mathsf{v}}$
    (b) by Definition 2, $[P'; v = A^i]_{\mathsf{t}} = [P']_{\mathsf{t}} + [A^i]_{\mathsf{t}}$
    (c) by inductive hypothesis $[P']_{\mathsf{t}} \geqslant [P']_{\mathsf{v}}$.
    (d) since $[-]_{\mathsf{t}} \in \mathbb{R}_{\geqslant 0}$, by monotonicity and (c) $[P']_{\mathsf{t}} + [A^1]_{\mathsf{t}} \geqslant [P']_{\mathsf{v}}$.
    $\therefore$ by (a), (b), (d) then $[P'; v = A^i]_{\mathsf{t}} \geqslant [P'; v = A^i]_{\mathsf{v}}$. □

Note that this proof only makes use of basic properties on relations and the specifications of $[-]_{\mathsf{t}}$ and $[-]_{\mathsf{v}}$ given here. This will be useful later: we can prove soundness of our denotational model with respect to the two definitions and get the above lemma for free following from this proof.

The abstract specification of the temporal behaviour here gives us a model to reason about time in Sonic Pi programs.

Example 2. Consider subprograms A, B, C where $[A]_{\mathsf{v}} = [B]_{\mathsf{v}} = [C]_{\mathsf{v}} = 0$ which are interposed with two sleep statements of duration s_1 and s_2:

$$P = A;\ \texttt{sleep } s_1;\ B;\ \texttt{sleep } s_2;\ C$$

Then by the above definitions, we see that if $[A]_{\mathsf{t}} \leqslant s_1$ and $[B]_{\mathsf{t}} \leqslant s_2$ then $[P]_{\mathsf{t}} = s_1 + s_2 + [C]_{\mathsf{t}}$.

We now move on to a denotational model, which provides a semantics for the core subset of the language described here. We'll prove this sound semantics with respect to the axiomatic model of this section, linking the two levels of model.

5. A denotational model of Sonic Pi's temporal semantics

In the following, we use Haskell as a meta language for a denotational model since it provides a convenient syntax for working with monads. This approach also provides an executable semantics that is useful for experimentation and integrating into other approaches. The source code is available at `https://github.com/dorchard/temporal-monad`.

We prove our model sound with respect to the time system approach of the previous section (Section 5.3) and briefly consider alternate simplified models using applicative functors or monoids (Section 5.5). In Section 6, we extend the model with "temporal warnings" describing overrun errors that can occur at runtime.

5.1 The *Temporal* monad

We define an interpretation $[\![-]\!]$ for programs, statements, and expressions into values of a parametric data structure, named *Temporal*, which encapsulates the effects of the Sonic Pi programs. Computations encapsulated by $Temporal$ are functions of the form:

$$(\textit{start time}, \textit{current time}) \\ \to (\textit{old vtime} \to \textit{kernel-access}\ (\textit{result}, \textit{new vtime}))$$

that is, mapping a pair of two times: the start time and current time of the computation (which are used to compute the time elapsed since the program start), to a stateful computation on virtual times (mapping from an old virtual time to a new virtual time) which may access the kernel to get the actual clock time, and produces a result along with the new virtual time. Concretely, $Temporal$ is defined:

$$\begin{array}{l}\textbf{data}\ \mathit{Temporal}\ a = \\ \quad T\ ((\mathit{Time}, \mathit{Time}) \to (\mathit{VTime} \to \mathit{IO}\ (a, \mathit{VTime})))\end{array}$$

where IO is part of the Haskell implementation and encapsulates access to the actual clock time from operating system.

$Temporal$ has a monad structure, defined by the following instance of the $Monad$ class:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ \mathit{Temporal}\ \textbf{where} \\ \quad \mathit{return}\ a \quad = T\ (\lambda\_ \to \lambda vT \to \mathit{return}\ (a, vT)) \\ \\ \quad (T\ p) \gg\!= q = T\ (\lambda(\mathit{startT}, \mathit{nowT}) \to \lambda vT \to \\ \qquad\qquad \textbf{do}\ (x, vT') \quad \leftarrow p\ (\mathit{startT}, \mathit{nowT})\ vT \\ \qquad\qquad\quad \textbf{let}\ (T\ q') = q\ x \\ \qquad\qquad\quad \mathit{thenT} \quad \leftarrow \mathit{getCurrentTime} \\ \qquad\qquad\quad q'\ (\mathit{startT}, \mathit{thenT})\ vT')\end{array}$$

To ease understanding, we recall the types of *return* and $(\gg\!=)$ and give some intuition of their behaviour for *Temporal*:

- $\mathit{return} :: a \to \mathit{Temporal}\ a$ lifts a pure value into a trivially effectful computation by ignoring the time parameters and providing the usual pure state behaviour of returning the parameter state vT unchanged along with the result. The nested use of return, on the right, comes from the IO monad (*i.e.*, $\mathit{return} :: a \to \mathit{IO}\ a$).
- $(\gg\!=) :: \mathit{Temporal}\ a \to (a \to \mathit{Temporal}\ b) \to \mathit{Temporal}\ b$ composes two computations together. The result of composing two temporal computations, with start time *startT*, current time *nowT*, and virtual time *vT*, is the result of evaluating first the left-hand side at time *nowT* and then right-hand side at the new current time *thenT*. The expression $\mathit{getCurrentTime} :: \mathit{IO}\ \mathit{Time}$ retrieves the time from the operating system.

Thus, the current time is retrieved with each use of $\gg\!=$, rather than using $\mathit{getCurrentTime}$ directly in any operation that requires the time. This choice was made in order to collect the temporal features of the model together in the monad.

$$\begin{array}{l}\mathit{time} :: \mathit{Temporal}\ \mathit{Time} \\ \mathit{time} = T\ (\lambda(\_, \mathit{nowT}) \to \lambda vT \to \mathit{return}\ (\mathit{nowT}, vT)) \\ \\ \mathit{start} :: \mathit{Temporal}\ \mathit{Time} \\ \mathit{start} = T\ (\lambda(\mathit{startT}, \_) \to \lambda vT \to \mathit{return}\ (\mathit{startT}, vT)) \\ \\ \mathit{getVirtualTime} :: \mathit{Temporal}\ \mathit{VTime} \\ \mathit{getVirtualTime} = T\ (\lambda(\_, \_) \to \lambda vT \to \mathit{return}\ (vT, vT)) \\ \\ \mathit{setVirtualTime} :: \mathit{VTime} \to \mathit{Temporal}\ () \\ \mathit{setVirtualTime}\ vT = T\ (\lambda\_ \to \lambda\_ \to \mathit{return}\ ((), vT)) \\ \\ \mathit{kernelSleep} :: \mathit{RealFrac}\ a \Rightarrow a \to \mathit{Temporal}\ () \\ \mathit{kernelSleep}\ t = T\ (\lambda(\_, \_) \to \lambda vT \to \\ \qquad\qquad \textbf{do}\ \mathit{threadDelay}\ (\mathit{round}\ (t * 1000000)) \\ \qquad\qquad\quad \mathit{return}\ ((), vT))\end{array}$$

Figure 8. Simple *Temporal* computations, used by the model

To model program evaluation, the $runTime$ operation executes a temporal computation inside of the *IO* monad, providing the start time from the operating system and virtual time 0:

$$\begin{array}{l}\mathit{runTime} :: \mathit{Temporal}\ a \to \mathit{IO}\ a \\ \mathit{runTime}\ (T\ c) = \textbf{do}\ \mathit{startT} \leftarrow \mathit{getCurrentTime} \\ \qquad\qquad\qquad\qquad\quad (x, \_) \quad \leftarrow c\ (\mathit{startT}, \mathit{startT})\ 0 \\ \qquad\qquad\qquad\qquad\quad \mathit{return}\ x\end{array}$$

Example 3. To illustrate the evaluation of temporal computations and the ordering and interleaving of calls to the operating system for the current time, consider the program:

$$\mathit{runTime}\ (\textbf{do}\ \{f; g; h;\})$$

(where $f = (T\ f'), g = (T\ g'), h = (T\ h')$) which desugars to the following *IO* computation, after some simplification:

$$\begin{array}{l}\textbf{do}\ \mathit{startT} \quad \leftarrow \mathit{getCurrentTime} \\ \quad (\_, vT') \leftarrow f'\ (\mathit{startT}, \mathit{startT})\ 0 \\ \quad \mathit{thenT} \quad \leftarrow \mathit{getCurrentTime} \\ \quad (\_, vT'') \leftarrow g'\ (\mathit{startT}, \mathit{thenT})\ vT' \\ \quad \mathit{thenT}' \quad \leftarrow \mathit{getCurrentTime} \\ \quad (y, \_) \quad \leftarrow h'\ (\mathit{startT}, \mathit{thenT}')\ vT'') \\ \quad \mathit{return}\ y\end{array}$$

This illustrates the repeated calls to $\mathit{getCurrentTime}$, the constant start time parameter, and the threading of virtual time state throughout the computation.

Figure 8 shows effectful operations of the *Temporal* monad, used in the next part of the model to access the current time, the start time, get and set the virtual time, and cause a kernel sleep.

5.2 Interpreting Sonic Pi statements

The interpretation $[\![-]\!]$ is overloaded on programs, statements, and expressions, thus the type of the interpretation depends on the syntactic category. Each interpretation produces a computation in the $Temporal$ monad. For open syntax (*i.e.*, with free variables), we model a variable environment mapping variables to values by the Env type, which is threaded through the interpretation. For expressions, we model the value domain via the $Value$ data type, for which we elide the details here.

The interpretation reassociates the left-associated program syntax (where the last statement is at the head of the snoc-list representation) to a right-associated semantics using a continuation-passing approach, *e.g.*, for a three statement program:

$$[\![((\emptyset; S_1); S_2); S_3]\!] = [\![S_1]\!] \mathbin{\hat{\circ}} ([\![S_2]\!] \mathbin{\hat{\circ}} ([\![S_3]\!] \mathbin{\hat{\circ}} [\![\emptyset]\!]))$$

where $\hat{\circ}$ represents (forwards, left-to-right) sequential, monadic composition of denotations in the *Temporal* monad.

The interpretation of statement sequences is defined:

$$\begin{aligned}[\![P]\!] &:: (Env \to Temporal\ ()) \to Temporal\ ()\\ [\![\emptyset]\!]\ k &= k\ emptyEnv\\ [\![P; S]\!]\ k &= [\![P]\!]\ (\lambda env \to ([\![S]\!]\ env) \gg\!= k)\end{aligned}$$

The parameter k is a continuation (taking an environment Env) for the tail of the right-associated semantics. In the inductive case, the continuation passed to $[\![P]\!]$ is the pre-composition of the interpretation of the statement S to the parameter continuation k.

At the top-level, we interpret a closed program to a $Temporal\ ()$ value by passing in the trivial continuation returning $()$:

$$[\![P]\!]_{\mathsf{top}} = runTime\ ([\![P]\!]\ (\lambda\_ \to return\ ()))$$

The interpretation of statements maps an environment to a possibly updated environment, inside of a *Temporal* computation, defined:

$$\begin{aligned}[\![S]\!] &:: Env \to Temporal\ Env\\ [\![\_ = E]\!]\ env &= ([\![E]\!] env) \gg\!= (\lambda\_ \to return\ env)\\ [\![v = E]\!]\ env &= ([\![E]\!] env) \gg\!= (\lambda x \to return\ env[v \mapsto x])\end{aligned}$$

For both kinds of statement, with and without variable binding, the expression E is evaluated where $[\![E]\!] :: Env \to Temporal\ Value$. The result of evaluating E is then monadically composed (via $\gg\!=$ of the *Temporal* monad) with a computation returning an environment. For statements without a binding, the environment env is returned unmodified; for statements with a binding, the environment env is extended with a mapping from v to the value x of the evaluated expression, written here as $env[v \mapsto x]$.

For expressions, we show just the interpretation of `sleep` and variables expressions:

$$\begin{aligned}[\![E]\!] &:: Env \to Temporal\ Value\\ [\![\texttt{sleep}\ t]\!]\ env &= sleep\ t\\ [\![v]\!]\ env &= return\ (env\ v)\end{aligned}$$

Thus, `sleep` is interpreted in terms of the *sleep* function (see below), where t is a constant, and variable expressions are interpreted as a projection from the environment. The concrete interpretation of other actions in the language, such as `play`, is ignored here since they does not relate directly to the temporal semantics.

***Interpretation of* sleep** The *sleep* operation, used above, provides the semantics for `sleep` as:

$$\begin{aligned}&sleep :: VTime \to Temporal\ Value\\ &sleep\ delayT = \mathbf{do}\ nowT \leftarrow time\\ &\qquad\qquad vT \leftarrow getVirtualTime\\ &\qquad\qquad \mathbf{let}\ vT' = vT + delayT\\ &\qquad\qquad setVirtualTime\ vT'\\ &\qquad\qquad startT \leftarrow start\\ &\qquad\qquad \mathbf{let}\ diffT = diffTime\ nowT\ startT\\ &\qquad\qquad \mathbf{if}\ (vT' < diffT)\\ &\qquad\qquad\quad \mathbf{then}\ return\ ()\\ &\qquad\qquad\quad \mathbf{else}\ kernelSleep\ (vT' - diffT)\\ &\qquad\qquad return\ NoValue\end{aligned}$$

where $NoValue \in Value$. Thus, *sleep* proceeds first by getting the current time *nowT*, calculating the new virtual time *vT'* and updating the virtual time state. If the new virtual time is less than the elapsed time *diffT* then no actual (kernel) sleeping happens. However, if the new virtual time is ahead of the elapsed time, then the process waits for the difference such that the elapsed time equals the virtual time.

Note that in this definition we have introduced an overhead, an ϵ time, arising from the time elapsed between the first statement $nowT \leftarrow time$ and the *kernelSleep* operation. The initial $time$ operation retrieves the current time and is used to calculate the duration of the preceding program. Any sleeping that happens however occurs after we have calculated the amount of time to sleep and after we have decided whether a sleep is needed (all of which takes some time to compute). Therefore a small ϵ time is introduced here. We will account for this in the following section.

5.3 Soundness of the temporal monad: *time safety*

We replay the previous axiomatic specifications on the temporal behaviour of Sonic Pi programs, and show that the monadic model is sound with respect to these, *i.e.*, that the model meets this specification. We call this a *time safety* property of the language, with respect to the time system provided by the axiomatic specification.

Definition 1 (recap). Virtual time is specified for statements of Sonic Pi programs by the following cases:

$$\begin{aligned}[P; v = E]_{\mathsf{v}} &= [P]_{\mathsf{v}} + [E]_{\mathsf{v}} & [\texttt{sleep}\ t]_{\mathsf{v}} &= t & [v]_{\mathsf{v}} &= 0\\ [\emptyset]_{\mathsf{v}} &= 0 & [A^i]_{\mathsf{v}} &= 0\end{aligned}$$

Lemma 2. *$[runTime\ [\![P]\!]]_{\mathsf{v}} = [P]_{\mathsf{v}}$, i.e., the virtual time of the evaluated denotational model matches the virtual time calculated from the axiomatic model.*

Proof. For our model, the proof is straightforward. For the case of $P; S$, we rely on the monotonicity of virtual time: virtual time is only ever increasing, and is only ever incremented by *sleep*. □

Definition 2 (recap). The actual elapsed time $[-]_{\mathsf{t}}$ can be (partially) specified at the level of programs by the following equations:

$$\begin{aligned}[\emptyset]_{\mathsf{t}} &\approx 0\\ [P; \texttt{sleep}\ t]_{\mathsf{t}} &\approx ([P]_{\mathsf{v}} + t)\ max\ [P]_{\mathsf{t}}\\ [P; v = A^i]_{\mathsf{t}} &\approx [P]_{\mathsf{t}} + [A^i]_{\mathsf{t}}\end{aligned}$$

Lemma 3. *$[runTime\ [\![P]\!]]_{\mathsf{t}} \approx [P]_{\mathsf{t}}$, i.e., the actual time of the evaluated denotational model is approximately equal to actual time calculated from the axiomatic model.*

Proof. The key case is for $(P; \texttt{sleep}\ t)$, which we show here. Our model interprets the evaluation of $(P; \texttt{sleep}\ t)$ as:

$$runTime\ ([\![P; \texttt{sleep}\ t]\!]\ (\lambda\_ \to return\ ()))$$

which desugars and simplifies as follows:

$$\begin{aligned}&runTime\ ([\![P]\!]\ (\lambda e \to ([\![\texttt{sleep}\ t]\!]\ e) \gg\!= \lambda\_ \to return\ ()))\\ \equiv\ &runTime\ ([\![P]\!]\ [\![\texttt{sleep}\ t]\!])\end{aligned}$$

The semantics reassociates statements, thus the interpretation for $P = ((\emptyset; S_1); ...); S_n$ is of the form $([\![S_1]\!] \hat{\circ} ... ([\![S_n]\!] \hat{\circ} [\![\texttt{sleep}\ t]\!]))$ (where $f \hat{\circ} g$ is monadic forwards composition, *i.e.*, $f \hat{\circ} g = \lambda x \to (f\ x) \gg\!= g$). Therefore, we can unroll and simplify the semantics further to get the following *IO* computation (where $[\![P]\!]'$ denotes the unrolled interpretation of P):

$$\begin{aligned}&\mathbf{do}\ startT \leftarrow getCurrentTime\\ &\quad (x, vT') \leftarrow [\![P]\!]'\ (startT, startT)\ 0\\ &\quad nowT \leftarrow getCurrentTime\\ &\quad \mathbf{let}\ vT'' = vT' + t\\ &\quad setVirtualTime\ vT''\\ &\quad \mathbf{let}\ diffT = diffTime\ nowT\ startT\\ &\quad \mathbf{if}\ (vT'' < diffT)\ \mathbf{then}\ return\ ()\\ &\qquad\qquad\qquad \mathbf{else}\ kernelSleep'\ (vT'' - diffT)\end{aligned}$$

where $kernelSleep'\ x = threadDelay\ (round\ (x * 1000000))$ is used to simplify the code here (as per the definition of *kernelSleep* in Figure 8).

From this we see that *diffT* $= [P]_{\mathsf{t}}$ and *vT'* $= [P]_{\mathsf{v}}$ and *vT"* $= [P]_{\mathsf{v}} + t$. Therefore, the guard of the **if** expression is $([P]_{\mathsf{v}} + t) < [P]_{\mathsf{t}}$. If the updating of the virtual time state and the computing of the guard takes e then the overall time taken is:

$$[P; \texttt{sleep}\ t]_{\mathsf{t}} = \begin{cases} [P]_{\mathsf{t}} + e & ([P]_{\mathsf{v}} + t) < [P]_{\mathsf{t}} \\ [P]_{\mathsf{t}} + e + ([P]_{\mathsf{v}} + t) - [P]_{\mathsf{t}} & \textit{otherwise} \end{cases}$$
$$= \begin{cases} [P]_{\mathsf{t}} + e & ([P]_{\mathsf{v}} + t) < [P]_{\mathsf{t}} \\ [P]_{\mathsf{v}} + t + e & \textit{otherwise} \end{cases}$$
$$= ([P]_{\mathsf{t}} + e)\ max\ ([P]_{\mathsf{v}} + t + e)$$
$$\approx [P]_{\mathsf{t}}\ max\ ([P]_{\mathsf{v}} + t)$$

where the third step follows by monotonicity of $+e$ on each side of the guard, The final stage in this simplification holds if $e \leqslant \epsilon$ and if the reduction to the interpretation to get to the above code takes less than ϵ. This ϵ is not however a drift, but a single overhead that can be be masked by a small `scheduleAheadTime` (see Section 3). □

Thus our model is *time safe*, in the sense that it satisfies the invariants described by the time system of Section 4. Following from these definitions we then get Lemma 1 "for free", that for all P, $[P]_{\mathsf{t}} \geqslant [P]_{\mathsf{v}}$, *i.e.*, a program never "under-runs" its virtual time specification. The lemma holds for free, since its proof relies only on the satisfaction of the specifications on $[-]_{\mathsf{t}}$ and $[-]_{\mathsf{v}}$, which we have shown above for our model.

5.4 Monad laws and equational theory for Sonic Pi programs

The *Temporal* monad is "weak", in the sense that the standard monad laws do not always hold. For example, consider the law:

$$(m \gg\!= \mathit{return}) \equiv m \tag{2}$$

where $m :: \mathit{Temporal}\ \ a$. In our model this corresponds to the following equality on programs:

$$[\![x = P; y = x]\!] \equiv [\![y = P]\!]$$

This should seem an intuitive rule to most programmers. However, for the *Temporal* monad, this law is violated in cases where m depends on the current time. For example, let m be defined:

```
m = do kernelSleep 1
       start ← start
       end ← time
       return (diffTime end start)   -- duration
```

Then we can run an experiment in GHCi to see that different results are possible:

```
*Main> runTime $ m >>= return
1.001113s
*Main> runTime $ m
1.00114s
```

(note, these results are also non-deterministic). The difference in results follows from the additional reduction required on $(\gg\!=)$ in the first case (left-hand side of (2)). Note that we calculate a duration here since using the absolute time produced by *time* would be disingenuous, since we are evaluating $m \gg\!= \mathit{return}$ and m at different start times.

In the above example, we have computed a time-dependent value (the duration). Due to variations in timing (and in the ϵ overheads), this disrupts the monad laws as seen above with the monad law shown in equation (2). However, in the programming model of Sonic Pi, there are no operations that expose the actual time (or current) time to the user– that is, the above program is not the model of any Sonic Pi program. We can therefore "quotient" the model by operations that do not expose the time, *i.e.*, we exclude *start* and *time*, which are not part of the Sonic Pi language. From this we regain the monad laws, up to $\approx$ due to small variations (as seen above). These are then:

$$\begin{aligned} (\mathit{return}\ x) \gg\!= f &\approx f\ x \\ m \gg\!= \mathit{return} &\approx m \\ m \gg\!= (\lambda x \to (f\ x) \gg\!= g) &\approx (m \gg\!= f) \gg\!= g \end{aligned}$$

which each provide the following standard equational theory on Sonic Pi programs respectively:

$$\begin{aligned} y\ \texttt{=}\ x\texttt{;}\ P &\equiv P\{y \mapsto x\} \\ x\ \texttt{=}\ P\texttt{;}\ y\ \texttt{=}\ x &\equiv y\ \texttt{=}\ P \\ (x\ \texttt{=}\ P\texttt{;}\ y\ \texttt{=}\ Q)\texttt{;}\ z\ \texttt{=}\ R &\equiv x\ \texttt{=}\ P\texttt{;}\ (y\ \texttt{=}\ Q\texttt{;}\ z\ \texttt{=}\ R) \end{aligned}$$

5.5 Subsets of the semantics

We briefly discuss alternative structuring of the model here.

For most of our example Sonic Pi programs here, the full structure of a monad is not needed to give their semantics as there is no use of binding between statements (and thus no dataflow). In these case just an *applicative functor* [MP08] or even a monoid would suffice. These can be derived from the monad structure on *Temporal* since all monads are applicative functors and all monads m define a monoid over $m\ ()$.

Applicative subset Applicative functors are described by the following interface in Haskell:

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

The *Applicative* class describes how to compose effectful computations encoded as values of type $f\ a$ (the effectful computation of a pure value of type a). Thus, *pure* constructs a trivially effectful computation from a pure value and <*> (akin to application) takes an effectful computation of a function and an effectful computation of an argument and evaluates the two effects in order to apply the function to the argument.

Our *Temporal* denotations have the applicative functor structure with the following derivation in terms of the monad:

```
instance Functor Temporal where
  fmap f x = do {x' ← x; return (f x')}
instance Applicative Temporal where
  pure a  = return a
  f <*> x = do {f' ← f; x' ← x; return (f' x')}
```

Note that the definition of <*> here orders the effects left-to-right.

The interpretation of sequential composition for statements (with no dataflow) is then $[\![P; Q]\!] = (\lambda() \to [\![P]\!])$ <*> $[\![Q]\!]$.

Monoid subset Alternatively, the full structure of an applicative functor is not even needed in this restricted case. Instead, a monoid over *Temporal ()* would suffice:

```
class Monoid m where
  mempty :: m
  mappend :: m → m → m
instance Monoid (Temporal ()) where
  mempty = return ()
  a `mappend` b = do {a; b; return ()}
```

with the interpretation $[\![P; Q]\!] = [\![P]\!]$ `` `mappend` `` Q and where *mempty* provides a *no-op*.

```
weakWarn :: VTime → TemporalE ()
weakWarn t = TE (λ_ → return ((), [Weak t])) ≫
  (warn $ "warning: overran by " ++ (show t))

strongWarn :: VTime → TemporalE ()
strongWarn t = TE (λ_ → return ((), [Strong t])) ≫
  (warn $ "WARNING: overran by " ++ (show t))

warn :: String → TemporalE ()
warn s = lift (T (λ_ → λvt → do putStrLn s
                                return ((), vt)))

epsilonTime :: TemporalE VTime
epsilonTime = TE (λeps → return (eps, []))

lift :: Temporal a → TemporalE a
lift t = TE (λ_ → fmap (λa → (a, [])) t)
```

Figure 9. Simple *TemporalE* computations

6. Emitting overrun warnings

We extend the *Temporal* monad with an additional parameter for the ϵ time, which we interpret as the maximum allowable overrun, and an output stream for sending "warnings" when overruns occur.

Overrun warnings are either *strong*, when the real time is more than ϵ ahead of virtual time, or *weak* when the real time is less than ϵ ahead of virtual time. That is:

- $[P]_t > ([P]_v + \epsilon) \quad \Rightarrow [\![P]\!] \rightsquigarrow$ *strong warning*
- $[P]_v \leqslant [P]_t < ([P]_v + \epsilon) \quad \Rightarrow [\![P]\!] \rightsquigarrow$ *weak warning*

We redefine the interpretation $[\![-]\!]$ to produce computations described by the following type $TemporalE$, thus $[\![P]\!] : TemporalE\ ()$:

```
data Warning = Strong VTime | Weak VTime
data TemporalE a =
  TE (VTime → Temporal (a, [Warning]))
```

Therefore, *TemporalE* wraps the previous *Temporal* type with a *VTime* parameter for ϵ and pairs the result with a list, representing the output stream of warnings. The *lift* function (shown in Figure 9) allows the previous effectful operations on *Temporal* to be promoted to the *TemporalE* type (by ignoring the new parameter for ϵ and producing the empty output stream), of type $lift :: Temporal\ a \rightarrow TemporalE\ a$. Figure 9 shows a number of other simple *TemporalE* computations for raising warnings and accessing the ϵ parameter.

The *TemporalE* encoding has the following instance of *Monad* which is simply a combination of the usual reader monad behaviour (for the ϵ parameter) and the writer monad (for the output stream), lifted to the *Temporal* monad:

```
instance Monad TemporalE where
  return a = TE (\_ → return (a, []))
  (TE p) ≫= q = TE (λeps → do (a, es) ← p eps
                              let (TE q') = q a
                              (b, es') ← q' eps
                              return (b, es ++ es'))
```

The **do** here is a *Temporal* computation, with the previous monadic semantics.

Evaluating *TemporalE* computations is much the same as before, with the ϵ time passed as a parameter:

```
runTime :: VTime → TemporalE a → IO (a, [Warning])
runTime eps (TE c) = do startT ← getCurrentTime
                        let (T c') = c eps
                        (y, _) ← c' (startT, startT) 0
                        return y
```

Finally, the new definition of *sleep* for *TemporalE* is the point at which warnings may be emitted:

```
sleep :: VTime → TemporalE ()
sleep delayT =
  do nowT   ← time
     vT     ← getVirtualTime
     let vT' = vT + delayT
     setVirtualTime vT'
     startT ← start
     eps    ← epsilonTime
     let diffT = diffTime nowT startT
     if (vT' < diffT)
       then if ((vT' + eps) < diffT)
              then strongWarn (diffT − vT')
              else weakWarn (diffT − vT')
       else kernelSleep (vT' − diffT)
```

The definition is similar to *sleep* for *Temporal*, except that in the true-branch of the conditional there is additional testing to see if *diffT* is greater than the new virtual time $+\epsilon$ (in which case a strong exception is raised), or if *diffT* is between vT' and $vT' + \epsilon$ (in which case a weak exception is raised).

The implementation of Sonic Pi has a similar semantics and warning system, for which this provides a general description.

7. Related work

Section 1.1 considered some related live coding languages and approaches. Here we highlight related approaches to temporal programming and reasoning in logic, artificial intelligence, and dataflow programming.

Logics There has been various work on reasoning about time in logic. For example, modal CTL (Computational Tree Logic) has been extended with time bounds for deadlines [EMSS91] (RCTL, Real-Time Computational Tree Logic) and for soft deadlines with probabilities on time bounds [HJ94]. In these logics, temporal modalities are indexed with time bounds, *e.g.*, $AF^{\leqslant 50}p$ means p is true after at least 50 "time units" (where A is the operator for *along all paths* and F for *finally/eventually*). Our approach is less prescriptive and explicit, but has some resemblance in the use of `sleep`. For example, the program `sleep` $t; P$ roughly corresponds to $AF^{\leqslant t}[\![P]\!]$, *i.e.*, after at least t then whatever P does will have happened. Our framework is not motivated by logic and we do not have a model checking process for answering questions such as, at time t what formula hold (what statements have been evaluated). The time system approach of Section 4 does however provide a basis for programmers to reason about time in their programs. In practice, we find that such reasoning can be done by children in a completely informal but highly useful way; the language has reached a sweet-spot between expressivity and ease of understanding.

Artificial intelligence Reasoning about the temporal component of events and action is a classic problem in artificial intelligence (*e.g.*, Shoham [Sho88], Shanahan [Sha95], Fisher *et al.* [FGV05]), in which causal mechanisms and metrical description may be more or less tightly coupled. Human interaction with systems that employ temporal reasoning can be considered either from a software engineering perspective (the usability of formal time notations, for example as in Kutar *et al.* [KBN01]), or from a cognitive science

standpoint, as in Honing's discussion of music cognition from a representational perspective [Hon93]. This is particularly relevant to Sonic Pi, where we are trying to invent a novel representation for music. Honing observes that representation of time in music can be both declarative and procedural, drawing on propositional and analogical cognitive resources. These representations may have conflicting implications for efficiency of control and accessibility of knowledge, for example trills or vibrato can be readily performed by an expert musician, but use mechanisms that are difficult to describe. Honing suggests that computer music systems should be distinguished according to whether they support only tacit time representation (events are encoded only as occurring "now"), implicit time representation (events are ordered in a metrical sequence) or explicit time representation (temporal structure can be described and manipulated). These principles can be used to compare alternative design options for systems such as Sonic Pi. Bellingham *et al.* [BHM14] provide a survey of 32 algorithmic composition systems, in which they apply Honing's framework to discuss the problem of notating the hierarchical combinations of cyclical and linear time that result in musical perception of pattern and tempo.

Dataflow Various dataflow languages incorporate real or virtual times into their semantics and core language constructs. For example, clocked dataflow languages (*e.g.*, LUSTRE [PHP87]) provide stream-based abstractions over time with a notion of discrete clock which may or may not correspond to real time.

Related to the dataflow tradition, the *functional reactive programming* (FRP) paradigm abstracts over time-varying, reactive values and discretely-timed sequences of events in a declarative language [NCP02]. Related to our work, FRP has been used for designing modular synthesisers [GN08], for which Sonic Pi v2.0 has related functionality. The FRP notion of *events* (a discrete stream) is used to encode sequences of notes. Our approach is less declarative, but requires a smaller set of constructions in order to support our first-language and educational goals.

Although the general approach is very different, the overarching theme of a semantics for time is common to both this work and FRP. There are related notions to *time safety* in the semantics of Wan and Hudak, where an idealised denotational semantics for FRP is compared to operational (implementation-oriented) semantics [WH00]. In their work, an implementation is said to be "faithful" when its actual implementation differs only by an ϵ-time to the denotational model. This is similar to the conditions of our time safety property between our axiomatic time system (Section 4) and the monadic model (Section 5).

8. Conclusion

This paper described an enhancement to the Sonic Pi language that improves the quality of musical experience for novice programmers in a live coding context. This is achieved by modifying the semantics of the familiar "sleep" operator in a manner that is consistent with musical expectations, while differing from the conventional interpretation in many languages. As a result, the enhanced Sonic Pi is able to retain identical concrete syntax to earlier versions, while implementing behaviour that is simple and predictable from a programmer perspective. Other music programming systems often provide similar mechanisms in order to achieve predictable timing behaviour, and our solution is comparable to those that have been implemented in other systems. We therefore introduced a formal semantics that can be used to prove the desirable properties of this kind of temporal behaviour. This combination of simple syntax, with formally defined semantics that correspond to user expectations, promises to be beneficial beyond the domain of music programming, to other types of physical world interface.

We used the phrases *time system* and *time safety* to draw analogy with traditional notions of *type* system and *type* safety. Further work is to expand the power of time systems and the notion of time safety, beyond what we have introduced here, exploring their use in live coding languages and languages for temporal coordination (such as in robotics). We considered the safety property of "not being too early", which is an invariant of the Sonic Pi language. Further work is to explore language invariants relating to deadlines (similar to the real-times logics discussed earlier).

Acknowledgements Thanks are due to the anonymous reviewers for their extremely helpful comments which in particular have improved the semantic part of this paper. Further thanks to Henrik Nilsson for his assistance and comments, Andrew Rice for helpful discussion about the temporal analysis, and Andy Hopper for his support. This work was kindly supported by the Raspberry Pi foundation.

References

[AB13] Samuel Aaron and Alan F Blackwell, *From Sonic Pi to Overtone : Creative Musical Experiences with Domain-Specific and Functional Languages*, The First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (Boston, Massachusetts, USA), ACM, 2013, pp. 35–46.

[Bal02] Mieke Bal, *Travelling concepts in the humanities: A rough guide*, University of Toronto Press, 2002.

[BHM14] Matt Bellingham, Simon Holland, and Paul Mulholland, *A cognitive dimensions analysis of interaction design for algorithmic composition software*, Proceedings of Psychology of Programming Interest Group Annual Conference 2014 (Benedict du Boulay and Judith Good, eds.), University of Sussex, 2014, pp. 135–140.

[BMNR14] Alan Blackwell, Alex McLean, James Noble, and Julian Rohrhuber, *Collaboration and learning through live coding (Dagstuhl Seminar 13382)*, Dagstuhl Reports **3** (2014), no. 9, 130–168.

[EMSS91] E Allen Emerson, Aloysius K Mok, A Prasad Sistla, and Jai Srinivasan, *Quantitative temporal reasoning*, Computer-Aided Verification, Springer, 1991, pp. 136–145.

[FGV05] Michael David Fisher, Dov M. Gabbay, and Lluis Vila (eds.), *Handbook of Temporal Reasoning in Artificial Intelligence*, Elsevier B.V., 2005.

[GN08] George Giorgidze and Henrik Nilsson, *Switched-on Yampa – Declarative Programming of Modular Synthesizers*, Practical Aspects of Declarative Languages, Springer, 2008, pp. 282–298.

[HJ94] Hans Hansson and Bengt Jonsson, *A logic for reasoning about time and reliability*, Formal aspects of computing **6** (1994), no. 5, 512–535.

[Hon93] Henkjan Honing, *Issues on the representation of time and structure in music*, Contemporary Music Review **9** (1993), no. 1, 221–238.

[IG13] The IEEE and The Open Group, *sleep – the open group base specifications issue 7*, 2013, `http://pubs.opengroup.org/onlinepubs/9699919799/functions/sleep.html` Retrieved 15 May, 2014.

[KBN01] M. Kutar, C. Britton, and C.L. Nehaniv, *Specifying multiple time granularities in interactive systems*, Lecture Notes in Computer Science **1946** (2001), 51–63.

[Lee09] Edward A Lee, *Computing needs time*, Communications of the ACM **52** (2009), no. 5, 70–79.

[McL13] Alex McLean, *The textural x*, Proceedings of xCoAx2013: Computation Communication Aesthetics and X (2013), 81–88.

[ME14] Sean McDirmid and Jonathan Edwards, *Programming with Managed Time*, Tech. report, Microsoft, 2014.

[MP08] Conor McBride and Ross Paterson, *Functional pearl: Applicative programming with effects*, Journal of functional programming **18** (2008), no. 1, 1–13.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson, *Functional reactive programming, continued*, Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, ACM, 2002, pp. 51–64.

[PHP87] Daniel Pilaud, N Halbwachs, and JA Plaice, *Lustre: A declarative language for programming synchronous systems*, Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY, vol. 178, 1987, p. 188.

[SG10] Andrew Sorensen and Henry Gardner, *Programming with time: cyber-physical programming with impromptu*, ACM Sigplan Notices **45** (2010), no. 10, 822–834.

[Sha95] Murray Shanahan, *A circumscriptive calculus of events*, Artificial Intelligence **77** (1995), no. 2, 249–284.

[Sho88] Yoav Shoham, *Reasoning about change : time and causation from the standpoint of artificial intelligence*, The MIT Press series in artificial intelligence, Cambridge, Mass. MIT Press, 1988, Includes index.

[WC03] Ge Wang and Perry R Cook, *ChucK : A Concurrent, On-the-fly, Audio Programming Language*, International Computer Music Conference, 2003, pp. 1–8.

[WH00] Zhanyong Wan and Paul Hudak, *Functional reactive programming from first principles*, ACM SIGPLAN Notices, vol. 35, ACM, 2000, pp. 242–252.