A PATTERN SYSTEM FOR SOUND PROCESSES

Hanns Holger Rutz
University of Music and Performing Arts Graz
Institute of Electronic Music and Acoustics (IEM)
rutz@iem.at

ABSTRACT

This article reports on a new library for the ScalaCollider and Sound Processes computer music environments, a translation and adaptation of the patterns subsystem known from SuperCollider. From the perspective of electro-acoustic music, patterns can easily be overlooked by reducing their meaning to the production of "notes" in the manner of "algorithmic composition". However, we show that they can be understood as a particular kind of programming language, considering them as a domain specific language for structures inspired by collection processing. Using examples from SuperCollider created by Ron Kuivila during an artistic research residency embedded in our project *Algorithms that Matter*, we show the challenges in translating this system from one programming language with a particular set of paradigms to another. If this process is studied as a reconfiguration of an algorithmic ensemble, the translated system produces new usage scenarios hitherto not possible.

1. INTRODUCTION

In 2017, we started the three year artistic research project *Algorithms that Matter* (ALMAT)[1] which aims at understanding the agency in algorithmic experimentation, the reciprocal exchange between computer musicians and their (mostly software) machines. In this project, we make use of software systems developed by the investigators, which serve as vehicles for experimentation, artistic practice and research. One way to observe agential properties is to see what happens when these highly personalised systems are exposed to other artists, who may come with very different preconceptions as well as with their own tools. To facilitate this source of friction, we invite guest artists to an in situ residency, preceded by a preparatory online dialogue. This article describes how the first of these exchanges led to a process of reimplementing the patterns subsystem of SuperCollider within our own framework Sound Processes, and how this process generated new conceptual opportunities.

[1] https://almat.iem.at/

2. PATTERNS IN THE COURSE OF RAPPROCHEMENT

In August 2017, I began a written exchange with Ron Kuivila, ALMAT's first guest artist, to see where we could connect in terms of our work process. Since Kuivila is a developer and user of SuperCollider, and my framework Sound Processes [1] incorporates a dialect of SuperCollider for real-time sound synthesis, based on the Scala programming language, it seemed natural to ask him to look into Sound Processes and see if he could imagine implementing sound sketches with it. Kuivila was working heavily with a subsystem within SuperCollider called *patterns*, and in fact he had contributed to the current implementation of this system and written an authoritative tutorial on it [2]. The pattern system is a fairly self-contained part of the SuperCollider language part, and initially functions independently from the sound synthesis part. This may be one reason why I personally had been exposed only superficially to it, the other being that the use cases highlighted by the documentation and online help are mostly those that fall into a narrow category of "algorithmic composition", where symbols—prominently pitch and time—are produced and processed, an approach relatively remote from my own "electro-acoustic" practice. It is thus no surprise that Sound Processes did not feature anything equivalent to patterns.

We stood in front of the curious situation that we both seemingly worked within the same environment of SuperCollider, but that Kuivila needed to find an entry path to Sound Processes, and I needed to find an entry path to patterns. This two way clarification dialogue dominated the first two months of our exchange, and from this emerged the project of trying to understand the "mattering" of a particular set of algorithms, the pattern system, through the attempt of translating it into my own system.[2] Although I described patterns as relatively self-contained, this pertains to the conceptual level, whereas architecturally patterns are deeply intertwined with SuperCollider, as they rely on the composition of abstract functions and on coroutines [3] for their implementation.

2.1 Anatomy of Patterns

Briefly, patterns are small composable building blocks—in their atomicity and combinatorics not unlike SuperCollider's UGens for DSP blocks—describing the generation,

[2] The entire research process, in which the patterns system only played a partial role, is documented on the Research Catalogue: https://www.researchcatalogue.net/view/386118/386119 (accessed 02-Apr-2018).

filtering and transformation of a stream of symbols (often numbers). Patterns are static descriptions of a structure, and in order to produce actual symbols they are expanded to stateful *streams*, which respond to the method `next` by emitting the respective next symbol. For example, in `Pseq([1, 2, 3], 2).asStream.nextN(8)`, the `Pseq` describes a pattern that iterates twice over the sequence 1, 2, 3; `asStream` creates a fresh stream from the pattern (initialising the state, such as the iteration counter), and `nextN(8)` calls `next` eight times, returning an array of the values thus produced: `[ 1, 2, 3, 1, 2, 3, nil, nil ]`. Any attempt to yield more values than represented by the stream results in the return value of `nil`, indicating that the stream is exhausted. This mechanism is used in the streams implementation to determine the end of stream, for example when concatenating two streams.

The majority of patterns qualify as *value patterns*, simply describing the transformation of values abstracted from particular context, for instance arithmetic operations, grouping, duplicating, filtering elements; a small subset is formed by *event patterns* that assume the element type `Event` (a key-value dictionary) which provides temporal information, and is thus used to combine streams sequentially or parallel in time, and also providing a representation of sounding objects.

3. MATCHES AND MISMATCHES FOR SCALA

Since streams in SuperCollider are implemented based on coroutines, where each coroutine invocation yields the next value of the stream, a straight forward translation into the Scala programming language would be based on coroutines as well. Coroutines fundamentally interact with the control flow of a program, and therefore good coroutine support usually has to be built into a language (this is the case for SuperCollider). Scala however does not have native coroutines. A former research project yielded a compiler extension for delimited continuations [4], which could be used to implement coroutines, but this experimental extension was never absorbed into the language and was abandoned. Scala's relatively new macro meta-programming system facilitated user contributed language transformations which were utilised in a dedicated coroutines library.[3] Nevertheless, macros are still an experimental feature in Scala, and lacking editor and IDE support make them not a first choice for an architectural foundation. More importantly, while coroutine based formulations of the patterns algorithms are concise and easy to devise—since we can apply the procedural thinking of a sequential flow of commands that transform state—the imperative model of state mutation implied by coroutines is not susceptible to adaptation to other needs, such as the software transactional memory (STM) used by Sound Processes, which would conflict with non-transactional modifications.

Patterns are relatively pure in SuperCollider, therefore a good match for Scala whose idiomatic style favours purely functional reasoning. On the other hand, Kuivila's use of patterns heavily relies on hybrid patterns that extend the set

[3] `http://storm-enroute.com/coroutines/` (accessed 02-Apr-2018)

```
Pspawner { | sp |
  var i;
  ~np.set(\stretch, 1);
  ~seqs = ~seq.collect { | v, i | ~seq[i..] };
  ~seqs = ~seqs.mirror;
  ~seqs.pop;
  loop {
    i = 0;
    while({ i < (~seqs.size - 2) }, {
      var syn1, syn2;
      i = ~sectionIndex;
      syn1 = (instrument: 'gated sine1', gate: 1,
        f: ~seqs[i][0], pan: -1, at: 15, decay: 1,
        amp: 80/~seqs[i][0] min: 1).synth;
      topEnvironment[\np][0] = syn1;
      topEnvironment[\np][1] =
        (instrument: 'gated sine2', amp: 0, at: 3);
      sp.wait(0.5);
      sp.seq(
        Pbind(*[
          // ...
        ])
      );
      TempoClock.default.tempo = 1;
      syn2 = (instrument: 'gated sine', gate: 1,
        f: ~seqs[i][0], pan: 1, at: 4 , decay: 4 ,
        amp: 80/~seqs[i][0] min: 1, sustain: 5);
      // ...
      sp.wait(3);
      if (~sectionIndex == i) { ~sectionIndex = i+1 };
    });
    topEnvironment[\np][0] = { Silent.ar };
    ~sectionIndex = 0;
    sp.wait(1.5);
  };
};
```

Figure 1. Principle pattern of the simplified and shortened version of Ron Kuivila's *Electric Wind*, written in *SuperCollider*.

of predefined patterns with custom patterns based on stream descriptions and thus coroutines, for example `Pfunc`, which wraps a routine function for the stream's `next` operation inside a pattern, or `Pspawner`, a mechanism to write temporal (event) patterns using a routine that allows easy assembly of parallel and sequential sections of a piece of music. Fig. 1 shows the skeleton of an early example Kuivila sent me to explicate his use of patterns for creating *Electric Wind*, a piece for flute and electronics. What can be seen here, is that patterns have been opened up to become fully fledged programs, issuing commands to the node proxy system (`~np.set`), performing operations on collections (`~seqs`), iterating and branching with `while` and `if`, and controlling the advancement of time (`sp.wait`, `sp.seq`, `TempoClock`).

Clearly, the program of Fig. 1 could not be simply translated into Scala, but would have to undergo major transformations. The following aspects provide challenges for the translation of the pattern system to Scala:

- Lack of viable coroutines support means that not only the streams for the predefined patterns have to be implemented differently, but there is no direct replacement for patterns based on functions executed on a routine, so control structures such as `loop`, `while`, `if` are not instantly available for the definition of user patterns.

- Sound Processes defines a set of clearly separated objects, and sounding objects are represented through "models" in the model-view-controller paradigm, rather than allowing the imperative control of synths, so a suitable interface between patterns and other objects must be found that replaces SuperCollider's `EventStreamPlayer`.
- Programs in Sound Processes are serialised to binary format, because ad-hoc compilation/interpretation from source is too slow, and because it comes with an embedded domain specific language (DSL) that can clearly delineate the representable programs and alleviate the problem of an interference between the evolution of the computer music framework and the reproducibility of the user programs. Lambdas (anonymous functions) are quite difficult to serialise fast and correctly, without accidentally capturing "unsupported API", unserialisable environment variables, and without risking binary incompatbility across language versions. For this reason, programs are entirely reduced to an abstract syntax tree (AST) in the particular DSL, which is a fast and robust representation. The lack of support for storing arbitrary lambdas however means that limitations of expressiveness must be otherwise overcome, if we want to retain a powerful generic approach to writing computer music.
- SuperCollider is a dynamically typed language, while Scala is statically typed. Part of the reason why dynamic languages are popular in the formalisation of musical structures is the ease with which heterogenous elements can be combined, and the effortlessness for the user with respect to defining and invoking polymorphic functions. For example, the expression `(Pseq([1, [2, 3], "a"]) + 1).asStream.all` is rather nonsensical but still valid in SuperCollider, producing the output `[ 2, [ 3, 4 ], "a 1" ]`. Our system will not support heterogenous lists, but we want to support seamless multi-channel-expansion and arithmetic expressions, consequently checking the correctness of a patterns program at compile (and serialisation) time.

4. CASE STUDY: THE FIFTH ROOT OF TWO

One of the sound studies created by Kuivila existed in raw form relatively early during his ALMAT residency. Its title became *The Fifth Root Of Two*, and it was again composed mostly using patterns. In the programme notes for the piece, which is inspired by Javanese gamelan, he writes:

> … a short melodic line of 8 to 12 pitches is generated via Brownian noise. Then every distinct sub-collection of three pitches is taken from the original line. Since pitches are repeated, this yields multiple collections of time points where those three pitches can be found. The three pitches are then played, sequencing through those time points until they are all exhausted and in synchronization with the basic melodic line.

Although patterns are used here once more primarily to generate a stream of time-pitch pairs, the way the program was constructed intrigued me: The compositional algorithm is broken down into a number of separatedly stated functions that permutate, analyse and combine the pitches of the voices, operating on arrays as chunks of data. These chunks of data then become the phrases or sections which are sequenced. Fig. 2 shows most functions that are called from within a `Pspawner` pattern. What happens in that pattern is that auxiliary patterns, such as Brownian motion `Pbrown(-6, 6, 3)`, are converted ad-hoc into a stream and evaluated for a finite number of elements, giving the cantus for each iteration as an array of numbers which is then used as argument to these functions; and they in turn become again patterns in the `Pbind` that represents the notes of the section.

In other words, there is an *excursion* pattern $\rightarrow$ stream $\rightarrow$ array $\rightarrow$ pattern. What if we could solve the translation problem in Scala by removing that excursion and providing all the necessary operations on patterns themselves that make them akin to any other collection (array)? Through multiple iterations, I prototyped this idea, leading first to a translation of the functions on arrays of Fig. 2 into functions on regular Scala collections—arguably one of the strong parts of Scala's standard library—as shown in Fig. 3, and in the second step to a translation from regular collections to patterns in Scala, as shown in Fig. 4.

4.1 From Collection to Pattern

When comparing the code, one can observe several things:

- The number of lines remains approximately the same. Scala is shorter where we can make use of the powerful functions of its collections library, and the "body" of the functions is similarly expressive. The method signatures are more formal in Scala, as static types need to be specified. We can use generic methods (e.g. `extract` being generic in the element type `A`), but we have to model operational constraints, such as the ability to perform arithmetic operations on generic elements, using so-called type-class parameters, here `Integral` for Scala collections, and our variant `Num` in the patterns case. Type-classes are passed implicitly in Scala, so the caller of these functions generally does not carry the burden to figure these out, however the API design of the patterns library must carefully define and choose these type-classes.
- Each language has its idiomatic names for collection operations, which means that translating patterns user code from one language to the other needs careful matching. For example, the monadic operations `map` and `flatMap` in Scala correspond with `collect` and combined `collect` / `flatten` in SuperCollider.
- When transitioning to our patterns library, we use a trick to avoid the need to serialise lambdas that would normally occur as arguments to `map`, `flatMap`, etc.: **we introduce a new constraint such that the element type of the pattern on which these methods are called must be another pattern.** In other words, the methods are

```
// all pairs from two arrays
~directProduct = { | a, b |
  a.collect({| v | b.collect{ | w | v.copy.add(w) }})
    .flatten;
};

// collects the indices of every occurrence of
// elements of t in s
~extract = { | s, t |
  var locales = t.size.collect({ | j |
    var hits = [];
    s.do({ | b, i | if (b == t[j]) {
      hits = hits.add(i) } });
    hits;
  });
  locales;
};

// generate all tuplets from within x, an array where
// each element is an array of occurences of a value
~allTuples = { | x |
  var index, size, current;
  size = x.size;
  if (size < 2) {
    x.first
  } {
    current = x[0].collect(_.asArray);
    (1..size-1).do { | i |
      current = ~directProduct.(current, x[i])
    };
    current
  };
};

// dur of a set of time points relative to a cycle.
~computeDur = { | tps, cycle |
  var dur = tps.differentiate[1..];
  dur = dur mod: cycle;
  dur = dur.collect({| v , i |
    if (v == 0){ v = cycle}; v });
  dur.sum;
};

// sort groups of time points based on total dur
~sortTuples = { | array, cycle |
  array.sort({| a, b| ~computeDur.(a, cycle) <=
      ~computeDur.(b, cycle) })
};

// computes and sorts all sub patterns of a pattern
~computeDurs = { | pattern, cantus, start = 0|
  var positions, tuples, durs;
  positions = ~extract.(cantus, pattern);
  tuples = ~allTuples.(positions);
  tuples = tuples.sort({| a, b |
    ~computeDur.(a, 7) > ~computeDur.(b, 7) });
  durs = ([start mod: cantus.size] ++ tuples.flat)
    .slide(2).clump(2).collect({ | pr |
      var dur = pr[1] - pr[0] mod: cantus.size;
      if (dur == 0) { dur = cantus.size}; dur });
  durs;


};
```

Figure 2. Functions used in the SuperCollider code of *The Fifth Root Of Two*.

```
// all pairs from two arrays
def directProduct[A](a: Seq[Seq[A]], b: Seq[A]) =
  a.flatMap { v =>
    b.map { w => v :+ w }
  }

// collects the indices of every occurrence of
// elements of t in s
def extract[A](s: Seq[A], t: Seq[A]): Seq[Seq[Int]] =
  t.map { tj =>
    s.zipWithIndex.collect {
      case (b, i) if b == tj => i
    }
  }

// generate all tuplets from within x, an array where
// each element is an array of occurences of a value
def allTuples[A](x: Seq[Seq[A]]): Seq[Seq[A]] = {
  val hd +: tl = x
  tl.foldLeft(hd.map(Seq(_)))((ys, xi) =>
     directProduct(ys, xi))
}

// dur of a set of time points relative to a cycle.
def computeDur[A](tps: Seq[A], cycle: A)
                 (implicit num: Integral[A]): A = {
  import num._
  val dur0  = tps.differentiate
  val dur1  = dur0.map(mod(_, cycle))
  val dur   = dur1.map { v =>
    if (v == zero) cycle else v }
  dur.sum
}

// sort groups of time points based on total dur
def sortTuples[A](xs: Seq[Seq[A]], cycle: A)
                 (implicit num: Integral[A]) =
  xs.sortWith { (a, b) =>
    num.lteq(computeDur[A](a, cycle),
     computeDur[A](b, cycle))
  }

// computes and sorts all sub patterns of a pattern
def computeDurs[A](pattern: Seq[A], cantus: Seq[A],
     start: Int = 0): Seq[Int] = {
  val positions = extract(cantus, pattern)
  val tuples0   = allTuples(positions)
  val tuples    = tuples0.sortWith { (a, b) =>
    computeDur(a, 7) > computeDur(b, 7)
  }
  val clump = (Seq(mod(start, cantus.size)) ++
     tuples.flatten).sliding(2).toList
  clump.map { case Seq(pr0, pr1) =>
    val dur0 = mod(pr1 - pr0, cantus.size)
    if (dur0 == 0) { cantus.size } else dur0
  }
}
```

Figure 3. Translation of Fig. 2 to regular Scala collections.

```
// all pairs from two arrays
def directProduct[A](a: Pat[Pat[A]], b: Pat[A]) =
  a.flatMap { v =>
    b.bubble.map { w => v ++ w }
  }

// collects the indices of every occurrence of
// elements of t in s
def extract[A: ScalarEq](s: Pat[A], t: Pat[A]):
    Pat[Pat[Int]] =
  t.bubble.map { tj =>
    val same      = s sig_== tj.hold()
    val indices   = s.indices
    val indicesF  = Gate(indices, same)
    indicesF
  }

// generate all tuplets from within x, an array where
// each element is an array of occurrences of a value
def allTuples[A](x: Pat[Pat[A]]): Pat[Pat[A]] = {
  val hd = x.head.bubble
  val tl = x.tail
  tl.foldLeft(hd)(directProduct)
}

// dur of a set of time points relative to a cycle.
def computeDur[A](tps: Pat[A], cycle: Pat[A])
                 (implicit num: Num[A]): Pat[A] = {
  val one  = Constant(num.one)
  val dur0 = tps.differentiate
  val dur  = ((dur0 - one) mod cycle) + one
  dur.sum
}

// computes and sorts all sub patterns of a pattern
def computeDurs[A: ScalarEq](pattern: Pat[A],
   cantus: Pat[A], start: Pat[Int] = 0): Pat[Int] = {
  val positions = extract(cantus, pattern)
  val tuples0   = allTuples(positions)
  val tuples    = tuples0.sortWith { (a, b) =>
    computeDur(a, 7) > computeDur(b, 7)
  }
  val cantusSz  = cantus.size
  val clump     = ((start mod cantusSz) ++
    tuples.flatten).sliding(2)
  clump.flatMap { pr =>
    val (pr0, pr1) = pr.splitAt(1)
    ((pr1 - pr0 - 1) mod cantusSz) + 1
  }
}
```

Figure 4. Translation from Scala collections to our Scala patterns library.

defined only on nested patterns. This allows us to eagerly apply the lambda parameter, resulting in another *pattern program* (AST) that can thus be easily serialised. Where this is not possible, we make use of the `bubble` operation that transforms a `Pat[A]` (pattern of generic element-type `A`) to a `Pat[Pat[A]]`, wrapping each input element in a single-element pattern. Although it may look strange at first sight, it works hand and hand with the fact that our patterns somehow blur the difference between single values and patterns of single values. For example, if you would prepend and append an element to a regular Scala collection using `x +: xs` and `xs :+ x`, the single element as a pattern has the same kind as the sequence pattern to which is prepended or appended, the operations thus becoming simple concatenations `x ++ xs` and `xs ++ x`.

5. ENTIRE PATTERN PROGRAMS

With the help of the powerful collections originated methods as equivalent patterns operations, we are now capable of translating most of the central `Pspawner` pattern of *The Fifth Root Of Two*, shown in Fig. 5, into a purely functional patterns program, shown in Fig. 6. Notice how the `asStream` transitions to imperative style have disappeared. The following details of the translation and implementation are noteworthy:

- In the original code there was an imperative loop in each iteration of which a number of values was drawn from an ad-hoc stream `cantus`; instead in Scala, we use a nested pattern with `flatMap` operations, neatly aligned in the code by using Scala's `for`-comprehension syntax. In order to make the necessary "re-expansion" of patterns to streams work, **we introduce the concept of automatic stream reset**. For example, the `lPat` pattern that moves up and down to determine the cantus length is defined in the outer scope, the expression `len <- lPat.bubble` creates a stream of single-element patterns with each of the length values, and maps that stream; in every iteration of the mapping, the streams corresponding to the inner patterns, e.g. `offset` and `cantus0`, are *reset*, correctly producing a newly rooted Brownian movement in each iteration.
- We introduced a simplified version of the `Pbind` pattern (called `Bind` in Scala as we do not have to worry about the flat name space restriction that SuperCollider has), based on a dictionary from strings to patterns. We have not yet implemented the equivalents for the `wait` and `suspend` operations provided by `Pspawner`, and neither is there any notion yet of a `TempoClock`. Appropriate replacements for these functions will be addressed in a future update of the library.
- Special handling of resting symbol `\r` is not available yet (and in fact difficult to model in the nominal type system). Instead we treat -100 as a pitch value representing resting notes.
- Although we originally followed the distinction between `asStream` and `embedInStream` of SuperCollider's library,

which introduces a subtlety in the treatment of constants in patterns, we eventually eliminated this distinction, requiring the user to explicitly specify the desired behaviour of constants. By default a constant produces an infinite stream of its value, and one can use `take()` to produce a single-element pattern; the opposite operation is `hold()` which repeats the first value of a pattern indefinitely.

6. INTEGRATION WITH SOUND PROCESSES

The patterns library presented here is available as a standalone open source project,[4] which could be used with vanilla ScalaCollider. However, the actual aim is integration with the Sound Processes framework. In the current version, patterns consisting of time-value tuples can be used as control input to parameters of the `Proc` object, which roughly corresponds to a `Synth` or `NodeProxy` in SuperCollider. Therefore, the direction of flow here is pattern → proc. An interesting possibility is opened by the `Grapheme` breakpoint function object, into which patterns can be translated, allowing the manual or algorithmic modification of values produced by the evaluation of a pattern across time. We are currently complementing this usage direction of patterns with a possibility of using the opposite direction, where a `Proc` is used as an attribute of a `Pattern` object, which may then be looked up for the `Bind`'s *instrument* key, allowing proc → pattern and the equivalent of an event-stream-player.

While the first iteration of our library used simple mutable state for the implementation of streams, these have now been retrofitted with the transactional memory model of Sound Processes. That means that on the one hand patterns embodied by the `Pattern` object formalism in Sound Processes will be stored in the workspace as immutable patterns which are expanded to transient in-memory streams when used as parameters to sounding objects or reproduced as a sequence of sounds through `Bind`. On the other hand, we can now introduce a second object `Stream` as object formalism as well: A pattern is expanded into a state machine stored as such in the workspace of Sound Processes! Values from this stream can be polled at arbitrary instants in a sound program, and the state of the stream is preserved across very long durations and even closing and re-opening a workspace. What is essentially introduced is a form of persistent Max/PD or Open Music type of "patcher" (where values correspond to messages), the state of which is automatically preserved. I believe that this is a powerful new abstraction for creating compositions, a pathway I wish to explore in the future.

7. CONCLUSIONS

I have introduced a new patterns library inspired by SuperCollider and implemented in the Scala programming language to be used in ScalaCollider and Sound Processes. The challenges and possibilities provided by such translation process were discussed, showing how patterns can be incorporated in a purely functional model and expanded as stateful streams that can be stored and recalled across the boundary of a particular working session. The `Pattern` object thus becomes a third core abstraction within Sound Processes, next to real-time DSP functions embodied by `Proc` and offline or non-realtime DSP functions embodied by `FScape` [5]. The conceptual implications of this translation are only beginning to become clear. For example, the notion of resetable pattern sub-programs may carry a profound opportunity for understanding and integrating resetable programs in the `FScape` formalism as well, which is currently too closely tied to the constraints of real-time UGens after which it is modelled.

[4] `https://git.iem.at/sciss/Patterns`

Acknowledgments

I am very grateful to Ron Kuivila for having worked with us during the ALMAT artistic research residency, providing us with these case examples, engaging in fundamental discussions about the architecture of the patterns library, and giving feedback on our new and different implementation. ALMAT is funded by the Austrian Science Fund (FWF), as project PEEK AR 403-GBL.

References

[1] H. H. Rutz, 'Sound processes: A new computer music framework', in *Proceedings of the Joint 11th Sound and Music Computing Conference and the 40th International Computer Music Conference*, A. Georgaki and G. Kouroupetroglou, Eds., Athens: National and Kapodistrian University of Athens, 2014, pp. 1618–1626. [Online]. Available: `http://hdl.handle.net/2027/spo.bbp2372.2014.245`.

[2] R. Kuivila, 'Events and Patterns', in *The SuperCollider Book*, S. Wilson, D. Cottle and N. Collins, Eds., Cambridge, MA: MIT Press, 2011, pp. 179–205.

[3] A. Wang and O.-J. Dahl, 'Coroutine sequencing in a block structured environment', *BIT Numerical Mathematics*, vol. 11, no. 4, pp. 425–449, 1971. DOI: `10.1007/BF01939412`.

[4] T. Rompf, I. Maier and M. Odersky, 'Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform', in *ACM SIGPLAN Notices*, vol. 44, 2009, pp. 317–328.

[5] H. H. Rutz and R. Höldrich, 'A Sonification Interface Unifying Real-Time and Offline Processing', in *Proceedings of the 14th Sound and Music Computing Conference (SMC)*, Espoo, 2017.

```
~spawner = Pspawner({|sp|
  var pat;
  var length, percentRests;
  var lPat, rPat;
  var offset = 0;
  var catpat =
  Pbind(*[
    instrument: \sine4,
    note: Prout({ loop{ Pseq(~cantus).embedInStream
         }}),
    note: Pkey(\note) * 2.4 + 4,
    dur: 0.2,
    db: -45,
    octave:  5,
    detune: Pwhite(-2.0,2.0),
    pan: 0,
    out: Pwhite(0, 23),
    i: Pseq((0..23), inf),
    ar: 0.001,
    dr: 0.1,
    stretch: 1,
    legato: 1,
  ]);
  lPat = Pseq( (8..12).mirror, inf).asStream;
  rPat = Pseq( (5..8).mirror/25, inf).asStream;

  loop {
    length = lPat.next;
    ~cantus = (Pbrown(-6, 6, 3)).asStream
      .nextN(length);
    { ~cantus[~cantus.size.rand] = \r }.dup((length *
         rPat.next).asInteger.postln);

    ~catter = sp.par(catpat);

    ~pitchsets = ~cantus.asSet.asArray.powerset
      .select{ | v | v.size == 3};
    ~pitchsets = ~pitchsets.collect(_.scramble);
    ~durs = ~pitchsets.collect({ | pset |
      ~computeDurs.(pset, ~cantus).sum });

    ~pats = ~pitchsets.collect({ | part, i |
      i = i + offset mod: 24;
      Pbind(*[
        instrument: \sine4,
        #[note, dur], Pseq([~makePart.(part,~cantus,0,
              [1,1,2,2,4].choose),
          Pfunc({ ("voice" + i + "done").postln; nil
               })]),
        note: Pkey(\note) * 2.4 + 4,
        db: -15,
        octave:  5,
        legato: i.lincurve(0, ~parts.size, 0.02, 1,0),
        detune: Pwhite(-2.0,2.0),
        i: Pseq((0..23), inf, i),
        ar: 0.001,
        dr: 0.1,
        stretch: 1,
        db: i.linlin(0, ~parts.size, -40, -30),
      ]);
    });
    offset = offset + 1 mod: 24;

    sp.par(Ppar(~pats) );
    sp.wait((~cantus.size * (~durs.sort.last div:
         ~cantus.size + 1) * 0.1).postln );
    // ...
    sp.suspend(~catter);
    sp.suspendAll;
  }
});
```

Figure 5. Central pattern (shortened and simplified) in *The Fifth Root Of Two*, original SuperCollider variant.

```
def mkGraph[Tx](): Pat[Event] = {

  def mkNotes(in: Pat[Int]): Pat[Double] =
    in * 2.4 + 4.0

  val baseBind: Bind.Map = Map(
    "proc"    -> "sine4",
    "octave"  -> 5,
    "detune"  -> White(-2.0, 2.0),
    "dr"      -> 0.1,
    "stretch" -> 1
  )

  def catPat(cantus: Pat[Int]): Pat[Event] = {
    val map = baseBind ++ Bind.Map(
      "note"  -> mkNotes(cantus),
      "rest"  -> (cantus sig_== -100),
      "dur"   -> 0.2,
      "db"    -> -45,
      "pan"   -> 0,
      "out"   -> White(0, 23),
      "i"     -> ArithmSeq(0, 1).mod(24),
      "ar"    -> 0.001
    )
    Bind(map)
  }

  val lPat = Pat.loop()((8 to 12).mirror)
  val rPat = Pat.loop()((5 to  8).mirror) / 25.0

  val xs = for {
    len     <- lPat.bubble
    rests   <- rPat.bubble
    offset  <- ArithmSeq(0, 1).bubble
    cantus0 <- Brown(-6, 6, 3).grouped(len)
  } yield {
    val numPause    = (len * rests).toInt
    val cantus      = cantus0.updatedAll(
      len.hold().take(numPause).rand, -100)
    val cantusEvt   = catPat(cantus)
    val pitchSets0  =
     cantus.distinct.sorted.combinations(3)
    val pitchSets   = pitchSets0.map(_.shuffle)
    val numP        = pitchSets.size.hold()

    val pats        = pitchSets.mapWithIndex {
     (part, partsIdx) =>
      val partsIdxH = partsIdx.hold()
      val (notePat, durPat) = makePart(part, cantus,
        rest = -100, stutter = Pat(1,1,2,2,4).choose)
      val l  = partsIdxH.linlin(0, numP, 0.02, 1.0)
      val db = partsIdxH.linlin(0, numP, -40, -30)
      val map = baseBind ++ Bind.Map(
        "note"    -> mkNotes(notePat),
        "rest"    -> (notePat sig_== -100),
        "dur"     -> durPat,
        "legato"  -> l,
        "i"       -> (partsIdxH + offset).mod(24),
        "ar"      -> 0.001,
        "db"      -> db
      )
      Bind(map)
    }

    Par(pats :+ cantusEvt)
  }
  xs.flatten


}
```

Figure 6. Translation of Fig. 5 into our Scala patterns library.