Uzulangs: a Community of Musical Pattern Languages

Alex McLean*
alex@slab.org
Then Try This / Alpacalab
Sheffield, UK

Felix Roos
fr00s@proton.me
Lembach, France

Matthew Kaney
matthew.s.kaney@gmail.com
New York City, USA

Martin Gius
gius_martin@proton.me
University of Applied Arts
Vienna, Austria

Diego Villaseñor
diego@echoic.space
UNAM
Mexico City, Mexico

Dagur Kristinn Sigurðsson Björnsson
dagurkris@gmail.com
Independent
Reykjavík, Iceland

Jack Armitage
ja@æ.is
Afhverju Ekki
Húsavík, Iceland

Julian Rohrhuber
julian.rohrhuber@rsh-duesseldorf.de
Robert Schumann Hochschule
Düsseldorf, Germany

Abstract

The family of Uzulangs are introduced, being the family of free/open source live coding environments that have grown around TidalCycles. The question of what makes an Uzulang is raised, and addressed in terms of the representation of pattern as pure functions of rational time, expressive 'mini-notation' shorthand for rhythm, combinator libraries of functions for transforming pattern, in-code visualisations for live feedback, and monadic and monad-like means of aligning combining patterns. Existing Uzulangs are enumerated, with the aims and peculiarities of a few of them explained. The relation between Uzulangs and the communities of practice growing around them is then reflected upon.

Keywords

live coding, uzulangs, uzu languages, tidalcycles, strudel, algorithmic pattern, community

1 Introduction

The Uzu language (or *Uzulang* for short) story began with the development of TidalCycles (*Tidal* for short) from around 2006 (McLean 2014).[1] Along with other contemporary live coding systems, Tidal as we now know it is part of a perhaps surprising turn in technology for live performance – it has practically nothing in the way of a graphical user interface. It also requires the installation and use of Haskell, which has a reputation for being a difficult-to-learn research language. Tidal represents musical pattern using a relatively obscure model of computation, with an interface building on Haskell's advanced 'applicative' and 'monadic' type classes. In spite of all this, a great number of 'non-programmers' have embraced it as a way to make music. Furthermore, it has turned out that Tidal's nature isn't dependent on Haskell's features, and it has now been successfully ported to many other programming languages, boosting its popular use further.

These Tidal ports have not only in some cases become feature-complete relative to Tidal, but have become creative playgrounds in their own right. They share common ideas, but explore their own experiments in pattern representation, transformation, syntax, visualisation and user interface, with some features back-ported to Tidal. As such, it now makes no sense to continue to refer to this growing family of systems as 'tidal-derived'. As a community, we needed a new collective name for these sibling projects, and chose the name *Uzu languages*, generally shortened to *Uzulangs*. Tidal set a water-based metaphor for patterns, continued with vortex (McLean et al. 2022) and strudel (Roos and McLean 2023) – the latter having a German meaning of 'whirlpool', as well as pastry. Following this pattern, *Uzu* was chosen for the collective name, as a Japanese word for *spiral*.

In the following, we look to share our understanding of the features and philosophy that make a live coding environment (Blackwell et al. 2022) an Uzulang, and our work towards describing shared feature sets without placing strict 'standards' that could limit what an Uzulang can be. This paper therefore stands for an opening up of the notion of creative software development, by focussing on collaboration across communities. Here we look beyond the individual communities of practice that develop around a language, for wider cross-language community.

The lack of competition associated with the profit motive behind commercial software allows Uzulang implementers the unique opportunity to share their ideas freely under the Utopian umbrella of free/open source 'copyleft' licenses. This freedom has allowed us to approach the issue of sharing features not in terms of single, authoritative standards documents, but instead in terms of family resemblances, represented as collections of optional tests.

As our first collective paper on the topic of Uzulangs, we will share what we have learned from some of the Uzulangs we have made and used so far, and how we hope to develop a free/open source community of practice that supports further innovation and exchange.

*All authors contributed equally to this research.

[1] There was not a single point that Tidal was conceived or born, but a gradual coming together of ideas through McLean's creative music practice. See McLean's 2025 blog post "When was TidalCycles born?" for reflections.

This work is licensed under a Creative Commons Attribution 4.0 International License.

NIME '26, June 23–26, 2026, London, UK

© 2026 Copyright held by the owner/author(s).

2 What is an Uzulang?

We ask this question because we now have a range of sibling Uzulangs, developing in different directions, and are in the process of forming an interconnected community of practice around them (including writing the present paper together). This seems like a straightforward aim, but perhaps surprisingly, there can often be deep animosity between free/open source communities. Matters of which editor or programming language you use can half-jokingly be referred to as 'war', to the extent that people on different sides simply refuse to talk to each other. We look for a more open, respectful and productive spirit.

The question is then, *how* do we define what an Uzulang is? By dividing live coding languages into those that are Uzulang and those that are not? The 'classical' approach to such a categorisation, dating back to Aristotle, would be to state a set of definitions, where if a system meets all of them, it is an Uzulang. However, practically no psychology researchers still take this classical approach seriously, since advances in cognitive psychology introduced the 1970s (Rosch and Lloyd 1978). Instead, there are a range of competing theories that refine or replace the classical view. One approach is to define concepts by a somewhat vague set of characteristics, or family resemblances. We can then say that dogs are characterised by e.g. being furry and having four legs, but if one is born without fur, or accidentally loses a leg, then it is still a dog.

Without getting into the weeds of the concept of concepts and 'theory-theory', this lay reading of cognitive psychology can already inspire an open, flexible approach to defining what an Uzulang is. Rather than working on a rigid, versioned standards document as is common in software engineering, we can look for a set of optional characteristics. For example, although Uzulangs can be characterised by having a *mini-notation* inspired by the 'polymetric expressions' in Bernard Bel's Bol Processor (Bel 2001), it would be far too limiting to say that a language *must* implement the mini-notation, when there are many other rich Uzu characteristics that can be explored without one. Similarly, it is fair to say that the Bol Processor is not an Uzulang, despite being such an important inspiration – its foundations in logic programming, representation of time, and approach to time-setting really sets it apart from typical Uzulangs.

Accordingly, we are looking for sets of characteristics that help us compare systems. From there, we might not be able mark a clear delineation between Uzulangs and non-Uzulangs, but it should be clear whether it's interesting or useful to consider a given system as one. So that is our marker – a language can be considered an Uzulang, if it shares enough family resemblances with existing Uzulangs to make it interesting to consider it to be one.

2.1 Patterns as pure functions of time

One shared Uzu feature (or family resemblance) is the representation of pattern as functions of time, rather than as data structures. Usually these functions take a timespan (i.e. a begin and end time) as input, and return events that are active during the timespan 'window'. Here an event has two timespans – one giving when the event is 'active' within the query window, and another giving the timespan for the 'whole' event – as events can be broken into parts.

As an example, imagine the repeated pattern of the letters *A B C*, each lasting a third of a cycle. In order to use this pattern, it must be queried for a specific timespan. Figure 1 shows the

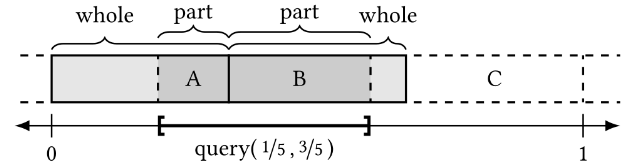

Figure 1: Querying the middle of a three-part pattern. Only the events that intersect with a query window are returned by the pattern function.

result of querying the span of (1/5, 3/5) cycles. This query returns two objects: one with the value *A* and one with the value *B*. The two timespans associated with the events are indicated above. Note in particular the 'part' which represents the intersection between the event and a query. All other events (including event *C*) are not included and can only be accessed with a different query.

The Uzu time*span*-based approach allows discrete events to be represented as above, and also continuously-varying values which are calculated based on query onsets. Continuous patterns can therefore be combined with each other without any discretisation, or combined with discrete patterns by sampling them at the onset of discrete events. That said, some Uzu languages explore a wholly signal-based approach reminiscent of sound synthesis, such as the Zwirn system described later in this paper.

2.2 Rational, 'cyclic' time

The 'time' of an Uzu pattern is expressed in terms of 'cycles'. A cycle is a loose concept; one cycle is not necessarily the repeating period of a pattern (indeed, patterns are not always periodic), but rather is used as the reference point when combining or transforming patterns.

Time is generally represented as a rational number in Uzu languages. This differs from most music software where time is either integral (such as in trackers and step sequencers) or floating point. While floating point operations tend to be more efficient than rational ones, rational numbers are of course well-suited to accurately representing ratios common in musical time.

Patterns can be played back by specifying a tempo in terms of cycles per second (or minute). The choice of cycle duration is arbitrary, but may be considered in terms of tala cycles in Indian music, or to a musical measure in Western classical music traditions.

2.3 Mini-notation

As discussed earlier, the mini-notation inspired by the Bol Processor's polymetric expressions has migrated from TidalCycles to several other Uzulangs, and has also been named as influence in the syntax of otherwise unrelated pattern DSLs. It is a shorthand for what can be expressed through functional composition, but more quickly with far fewer keypresses – an important factor in from-scratch live coding performances. It can be considered a mini-language embedded in a host language, and has been implemented across several Uzulangs in a similar way to how regular expressions have been implemented to POSIX and Perl standards in mainstream general purpose programming languages.

In both Tidal's Haskell and Strudel's JavaScript implementations, they appear in the language as double quoted strings. This

can give the misunderstanding that the functions themselves operate on strings, but through string overloading in Haskell, and transpilation in JavaScript, they are immediately parsed into patterns as functions of time.

The mini-notation syntax has grown over time, developing into a fully-featured language for describing rhythms, including repetition, alternation, polymeter, polyrhythm, increasing/decreasing density, randomness and so on. For full details, please refer to the documentation of e.g. Tidal and Strudel. As an example of its expressive terseness, the following shows a mini-notation expression together with an equivalent pattern expressed using strudel's method chaining.

```
// Mini notation:
// "<a b c> {d e, f g h}%4 [b c]?0.25"
// Equivalent to the following:
fastcat(slowcat('a', 'b', 'c'),
        stack(fastcat('d', 'e').pace(4),
              fastcat('f', 'g', 'a').pace(4)),
        fastcat('b', 'c').degradeBy(0.25)
       )
```

2.4 Flexible pattern transformation

A limitation of representing patterns as functions is that they are 'opaque', in that they have no structure (at least, until they are called), and cannot be modified. In order to transform a pattern then, you can only create a new pattern as a function that 'wraps' the old one, manipulating its input (the query timespan) and output (event timespan and value). This allows a surprising breadth of transformations; patterning time through e.g. shifting, reversing, speeding up/down, and applying functions to values including time-based transformations. An extensive combinator library of pattern transformations have been built around this representation.

The purity of Uzu patterns means that we lose the core ability to deal with state. Although there are workarounds for this (such as making patterns of state transformations), this means that state-based patterning, such as the use of Markov chains, is a relatively rare use of Uzu languages.

What Uzus lose in limited support for stateful patterns, they gain in the expressive freedom to manipulate and combine patterns. Because events aren't dependent on previous values, they support 'random access', in that you can query a pattern at any point in the past or future without calculating intermediary values. This means a pattern transformation can be freely shifted far into the future or past, and can efficiently represent complex interference patterns (such as compound polyrhythms and polymeters) that may have very long or even notionally infinite repeats, without having to store any data. Such patterns can be freely reversed, shifted and composed with other patterns, with no calculation required until the scheduler needs to query a pattern. Importantly for live coders, switching between one pattern to a new edit is straightforward, because there is no state to maintain and translate between the old and new pattern, apart from time.[2].

[2]TidalCycles does support stateful patterns, by creating patterns of state transitions (McLean 2021)

2.5 Aligning and combining patterns: "It's patterns all the way down"

Composition of patterns in Uzu languages is an interesting topic to try to talk about, as computer science notions of functional composition overlap heavily with musical composition structures.

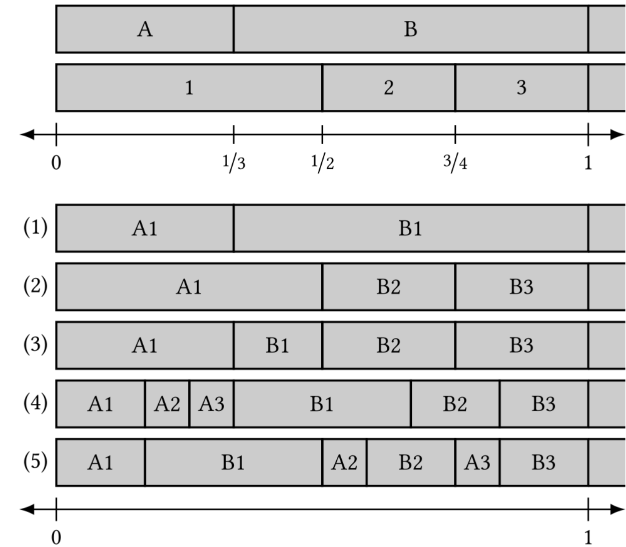

Figure 2: Combining the structure of two patterns using five different joins: (1) in, (2) out, (3) mix, (4) squeeze, and (5) squeezeout.

Although Uzu patterns are opaque, as functions of time, they nonetheless offer a flexible and rich range of techniques for aligning and composing patterns together. A key decision in combining two patterns is how to combine their structure – that is, when combining two partially-overlapping events, what timespan should the resulting event have? Uzu languages address this by implementing multiple abstract 'join' strategies. Figure 2 illustrates five common joins on a pattern of letters (`"A B _"` in mini-notation) and a pattern of numbers (`"1 [2 3]"`). The first two joins (1 and 2, called 'in' and 'out') defer structure to the first and second patterns respectively. Note that in these joins, event values are determined by whichever values the two patterns had at the event's onset. The 'mix' strategy (3) creates its structure from the intersection of both patterns' events. The last two joins (4 and 5) are examples of 'squeeze' strategies. In the regular 'squeeze' (4), each event of the first pattern is combined with an entire cycle of the second pattern scaled to fit. The 'squeezeout' join (5) performs the same operation with the patterns in the other order.

The roots of TidalCycles in Haskell is a great influence here, in particular the functor, applicative and monadic types, providing standard ways to combine containers such as patterns together which Tidal has embraced and extended. This allows most pattern functions to take arguments that are themselves patterns, such as in the example Strudel code `"A B C D".fast("1 2 3")`. Figure 3 illustrates this process: first, a function with a pattern of arguments such as `fast("1 2 3")` is converted to a pattern of functions each with a single argument. Then, each `fast` function is applied to the underlying `"A B C D"` pattern and an inner join is used to apply the structure of the pattern of functions to the various sped-up patterns of values.

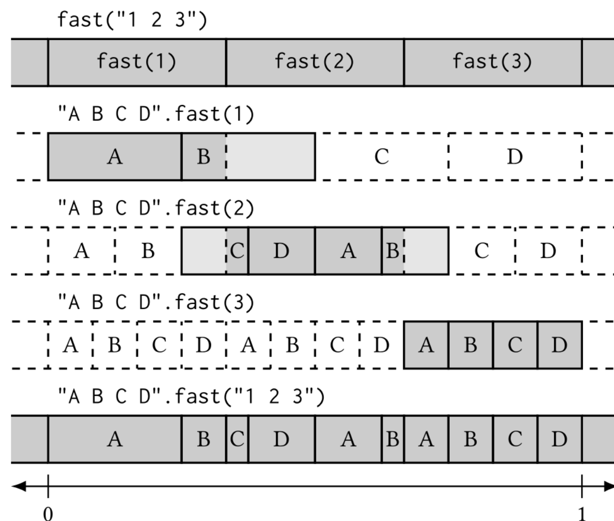

Figure 3: Applying a pattern of speed transformations to an underlying pattern of letters(`"A B C D"`). The pattern of transformations is used for querying the various transformed patterns, and the final output is shown at the bottom.

2.6 In-code visualisation

A visual marker for Uzulangs is the in-built visualisation. 'Event highlighting' has become an important feature of e.g. Strudel, Tidal and Mondo in recent years, where all the source values of currently active events are visually highlighted in the source code, helping live coders and audience members recognise the provenance of the sounds they hear in the code. This feature comes into its own in workshops and other educational settings, where the behaviour of more difficult to conceptualise functions (such as `iter` that progressively shifts a pattern) become tangible and more readily understandable when aural and visual perception of them is integrated through this visualisation. This was originally introduced in Tidal's *feedforward* live coding editor (McLean 2020), following earlier work in a Perl-based system (McLean 2004) but heavily inspired by the work of Roberts et al. (2015) in the Gibber live coding environment.

The Strudel Uzulang provides a strong range of visualisations, including linear and spiral 'pianorolls', oscilloscopes, pitchwheel, and a spectrum analyser. These can either be embedded inline within the code (underneath the function that calls them), or 'globally' in the editor background. Inspiration for this comes from Bret Victor's work on e.g. 'explorable explanations' (Victor 2011), as well as code-based notebook environments common in data science, such as Jupyter.

Providing data to the pattern visualisations is relatively straightforward to implement: this is again thanks to the state-free nature of Uzu patterns, meaning they are idempotent in terms of always returning the same result for a given input. The visualiser can therefore query a window from past to future, in order to build visualisations such as the live pianoroll. This buffer is then added to with further queries for each animation frame. When a live edit is made to a pattern, the pattern is queried such that past events (i.e., ones that have already happened) are maintained in the visualisation, and future ones updated.

3 Cyclic vs stepwise combination

Inspired by South Indian, Carnatic music traditions, a recent adaptation to Strudel and TidalCycles has been to support *stepwise* transformation and combination of pattern. This began with a major rewrite of Tidal, to support both functional and data structure representations of pattern. However following much work, this was abandoned in favour of a simpler approach – adding a rational value to patterns as a single piece of metadata representing the 'step count' of a pattern.

This simple addition allows a wide range of pattern transformations based on steps rather than cycles. These operations include dropping steps from the beginning or end of each cycle, concatenating successive cycles from two or more patterns weighted by their respective step counts, and expanding or 'zipping' steps together from two or more patterns. As with existing cycle-wise operations, parameters of these operations (for example the number of steps to drop from a pattern) may be patterned, with the proviso that the result of that patterning is played out over a single cycle. This tends to result in very dense patterns, so stepwise pattern operations are generally realigned using the pace function, which slows the pattern to match a given number of steps per cycle.

Although this approach allows a wide range of uses, stepwise pattern operations can only be applied when a pattern has a coherent number of steps per cycle, which is often not the case when e.g. other pattern transformations have already been applied. It is therefore still considered experimental. Further detail on the implementation and use of stepwise patterning is provided by McLean (2024), with the latest implementation in strudel documented at strudel.cc/learn/stepwise .

4 Defining Uzu lexicon

To find a common ground between different Uzu language implementations, one approach we're working towards is to define a *lexicon* for Uzu functions. Similar to a monolingual lexicon for natural language, it would contain Uzu *words* that are explained by other Uzu words, forming a body of text that contains different sets of characteristics in a structured format. Each Uzulang could be tested against the lexicon, or contribute its own unique characteristics for other Uzu languages to be tested against. The primary challenge is to find a common language that can be shared across implementations. That language should be easy to implement and focus on semantics, rather than syntax. We went with s-expressions, because they are easy to parse and interpret in any language. Here's an example lexicon entry:

```
(eq (seq a b c) (fast 3 (cat a b c)))
```

The `eq` declares equality between the two following expressions. The `seq` word is "explained" using the words `fast` and `cat`. This entry is not meant to be an exhaustive definition, but rather an example of an expected equality in the language, similar to a unit test in a software system. In the above example, the first argument of `fast` is implicitly chosen based on the number of arguments passed to `seq`. An Uzu language aiming to fulfil this equality needs to at least implement `eq`, `seq`, `fast` and `cat`. Then it would parse and interpret the s-expression in its own language.

Because we're dealing with functions of time, it's not clear what equality means, as time is potentially infinite. The `eq` function could by default test equality of events within the first cycle,

with the option of declaring the timespan to test against in the third and fourth arguments.

Because different Uzu languages have only partially overlapping sets of characteristics, the lexicon could be divided into multiple chapters, where each one focusses on a different characteristic. Up to this point, the lexicon is still only a concept without a concrete body of text that can be tested against.

5 Free/open source community

Free/open source software projects often mirror proprietary ones, for example:

- the separation of user and developer with communication via 'feature requests'
- relationships between free/open source projects couched in competitive language, undermining collaboration and exchange
- projects judged by user counts and 'stars' on source code hosts
- actually mirroring a proprietary system, constraining the free software to the feature set of the proprietary one
- hosted on proprietary platforms like Microsoft/Azure's github platform

With Uzulangs we are working to create space for an alternative approach, where ideas are shared freely. In particular, all users are also considered to be programmers and vice-versa - natural for a 'live coding' community based around 'end-user programming'. Therefore we don't have separate, gatekept 'user help' and 'core development' channels, instead cultivating a unified community, although with 'innards' subchannels for those interested in Uzu internals. In this scenario it's interesting how often even experienced developers find themselves asking fundamental 'beginner' questions about use, and how apparently simple questions spark deep conversation ultimately leading to new features for everyone to enjoy.

The approach to describing language features through equivalent tests described in §4 is alternative to a more centralised approach of standardisation. It encourages experimentation by promoting agency of individual language-makers, while still providing a rigorous way to establish compatibility of core features where desired.

Perhaps these ideals are as much part of the definition of an Uzulang as a formal specification of syntax and semantics would be. However, impacts of the current wave of AI investment and tools which allow free/open source software to be 'ingested' into large language models (and stripped of their open source licenses), are only now beginning to be felt and understood. With some concern for the future, we nonetheless hope this provides the free/open source community to collectively reflect on its aims.

6 Uzu languages in the wild

There are currently around seventeen Uzulangs available. Because they have all (apart from the original Tidal) involved translation of part of the source code of another Uzulang, they are all licensed using a GNU Public License or Affero GNU Public License.

Table 1 lists systems which can broadly be described as ports and expansions of the TidalCycles domain specific language. They each have their own individual aims and unique features and affordances, but can be said to largely keep within the TidalCycles paradigm.

The Uzulangs listed in Table 2 signal a different approach, rather than being embedded in a particular mainstream programming language, they are fully parsed languages with their own grammar. This gives them a freedom to experiment more freely with a custom syntax, in particular discarding the difference between mini-notation and the rest of the language, instead exploring unified language. These have been known as 'maxi-notation' or 'mondo-notation' Uzulangs. They may also experiment with alternative takes on the representation of pattern, for example with Zwirn.

7 Implementing Uzulangs

The following contains reflections on the development of a selection on individual uzulangs – the surprises and other learnings that have come from their development.

7.1 Minimal Viable Uzu – the TidalCycles Remake

The first Uzu language after TidalCycles was a 'remake', created during a two hour live stream on the 21st July 2021 (McLean 2021). This was done in Haskell, the same language used for TidalCycles, but without reference to the original source code. The aim was to gain and share understanding of the fundamental pattern representation behind Tidal (and now Uzu) beyond its current implementation. This minimal 'remake' was then used as a reference in implementing ports to other languages – to Python as 'Vortex' and to JavaScript as 'Strudel'. A similar but more complete 'minimal viable Uzu' is *idlecycles*, explained further in §7.6 below.

7.2 Strudel

Strudel (Roos and McLean 2023) is a TidalCycles port to JavaScript, started shortly after Vortex, but quickly becoming a free/open source software development focus, reaching feature parity with Tidal, while developing its own language, visualisation/UI and sound synthesis features, some of which have been backported to Tidal.

As a web application, Strudel does not require installation, which makes it very approachable for beginners and less technically minded people. For that reason, it also became a common choice for live coding workshops, quickly growing communities of practice. Through the versatility and openness of the web ecosystem, Strudel could quickly integrate different systems and protocols, such as WebAudio, MIDI, OSC and WebSerial, as well as adjacent languages like Hydra, CSound (*Csound*, n.d.) and Kabelsalat (Roos and Forment 2025).

In recent months, the Strudel community has grown faster than usual, after becoming viral on mainstream social media platforms. That virality has been a double edged sword. On the one hand, it has led to an influx of new players and contributors, on the other it attracted the attention of grifters and spammers. After the hype has faded, the community is now very active, with many people that probably wouldn't have heard about live coding otherwise.

7.3 Zwirn

The original motivation for developing Zwirn was partially from reading the paper "Alternate Timelines for TidalCycles" (McLean 2021), which raised the question, why the following expression shouldn't be valid Tidal code:

Name	Host language	License	Links	Notes
TidalCycles (aka Tidal)	Haskell	GPLv3	Homepage, Source	
Strudel	JavaScript	AGPLv3	Homepage, Source	Includes Tidal-like syntax and mondo
idlecycles	JavaScript	AGPLv3	Chap 1 2 3 4 5 6	Minimal, mostly intended for learning
Estuary	JavaScript / Haskell	GPLv3	Homepage, Source	Includes 'mini-Tidal' among many others
Tranquility	Lua	GPLv3	Source	
Vortex	Python	GPLv3	Source	
Cycles	Rust	GPLv3	Doc, Source	
Kidal	Kotlin	GPLv3	Source	
Piratidal	Clojure	GPLv3	Source	Can compile to the JVM and JavaScript
Jaffle	YAML	AGPLv3	Source	A YAML interface to Strudel
Weft	TypeScript	AGPLv3	Source	Applying Haskell syntax/types to Strudel
Marea.sc	SuperCollider	GPLv3	Source	Proof-of-concept with minitidal

Table 1: 'Conventional' Uzulangs

Name	Host language	License	Links	Notes
Zwirn	Haskell/SuperCollider	GPLv3	Source	
Modal	Lua	GPLv3	Source, Announce	
Godwit	Haskell	AGPLv3	Source	Based on simply-typed lambda calculus
froos's uzunotation	JavaScript	AGPLv3	Chap 1 2 3	
mondonotation	JavaScript	AGPLv3	Source (PR)	part of Strudel (see Table 1)
Rista-vél	Strudel	AGPLv3	Source	Icelandic dialect

Table 2: 'New-wave' Uzulangs

```
strudel REPL
1 $: s("bd*2 sd, hh oh rim hh").clip(.75)._spiral({size:120,stretch:1})

2 $: n(rand.range(0,7).rib(0,1).seg(8)).gain(rand.range(.2,1))
3   .off(1/8,add(n("4,2"))).scale("<D A>:dorian")._punchcard({height:80})
  ._pitchwheel({size:150,mode:'polygon'})
```

Figure 4: The Strudel user interface.

```
jux <(rev) (iter 4)> $
  sound [bd (every 3 (fast 4) [sn])]
```

Whereas Tidal's mini-notation is normally a domain specific language for expressing rhythm separate from its wider pattern combinator library, here it is applied on the top level of the language, a seemingly simple change with a lot of consequences. Zwirn became about following through with all of these consequences. The second big motivation came from a paper called "Lambda Calculus and Music Calculi" (Orlarey et al. 1994), which describes a way to augment the simple lambda calculus with a new construct for creating domain specific languages. The first prototype (titled *MiniLambda*) was based on the methods described here and dates back to late 2022. Unfortunately in the end, while mostly working, this approach did not prove to be feasible.

In early 2023 via an internship with Julian Rohrhuber, Martin Gius began to experiment with an Uzu language in SuperCollider. Doing this, they discovered some differences in the workings of our implementation and the 'original', mainly to do with the simplification of representing everything with functions of time instead of spans of time. To compare these different implementations better, they wrote another one in Haskell, which has now become *zwirn-core*, the semantic base for zwirn. Developing these ideas into a proper live coding language still took quite some time and zwirn is by no means a finished project. In fact, having the ability to change and adapt the language at a very low level will provide some interesting possibilities for more live coding specific features.

7.4 Piratidal

Piratidal was developed with the intention of understanding how TidalCycles works and to provide an alternative sequencer for the Clojure/Overtone community. Syntactically it tries to remain as close to TidalCycles as possible, within Clojure's lisp-like syntax. A simple program looks like this:

```
(p 1 (-> (note "<-12 0 7 -5>")
         (sound "superhammond")
         (slow 2)
         (gain 0.7)))
```

Perhaps the most interesting thing about Piratidal from a technical standpoint is that its implementation does not rely on porting the native Haskell constructs (monads, applicatives, etc.) to

Clojure. Rather, it follows the idiomatic style of the host language community; functional but data-driven. Therefore, it constructs tree structures representing the compiled cyclic pattern, which are then traversed as required by the `query` function.

Unfortunately, Piratidal hasn't gathered much interest from the Clojure community, perhaps because musical live coding itself is not, at the moment, a common practice there. Additionally, since the author's interests in patterns have shifted towards the isorhythmic-like ones (for which TidalCycles has not been the best fit), Piratidal is not currently under active development.

However, personally and as a live coder, I regard TidalCycles as an important reference that continues to engage me conceptually and aesthetically; I consider my current sequencers as referring to but not following the Uzulang model (as well as other models), and so transversally proposing a dialogue with them. Likewise, they do bear its influence in their design and concepts, particularly those most fitting to isorhythmic exploration, the patterning coming from the mini-notation, and the intent to arrive at elegant and expressive syntaxes for constructing such patterns.

7.5 Rista-vél

Rista-vél explores the idea of an Icelandic live coding language, inspired by esolangs, Inform and programming language localisation (Swidan and Hermans 2023). We did not create our own Uzu from scratch, rather we extended Strudel with two packages: `locale` which allows redefining Strudel functions and colours in user-defined dictionaries, and `ohm` for live coding Ohmjs grammars. With this prototype we translated Strudel into Icelandic and experimented with (non-LLM based) natural language programming. We wrote pedagogical materials for a workshop in Icelandic, and held a premiere concert at Mengi in Reykjavík in August 2025 and reflected on the project afterwards.

```
// Strudel
d1: s("organ").gain(1.2).room(1.5)

// @strudel/locale
davíð: hljóð("orgel").styrkur(1.2).ómur(1.5)

// @strudel/ohm with Rista-vél grammar:
Davíð spilar hljóð „orgel“,
    styrkur „1.2“ og ómur „1.5“.
```

We think Rista-vél can be considered as a curious flavour of Uzu-ing that bridges into languaging and poetry on the one hand, and also foregrounds hyper-local computing cultures, with potential implications for so-called "low resource languages" such as Icelandic. The project is ongoing and has recently received a grant from the Iceland Music Fund to support further practice-based research and pedagogy in Iceland. We also aim to encourage more exploration of Uzu dialects outside of the Icelandic context.

7.6 idlecycles

The "idlecycles" language aims to implement a minimal Uzu language in JavaScript. Its development took place in the form of six interactive blog posts[3] that explore various aspects of patterns. The syntax is identical to Strudel: using method chaining for pattern transformations and a basic mini notation for compact rhythmic expressions. The first five chapters use colored rectangles for pattern visualizations, only the last chapter transitions to outputting sound.

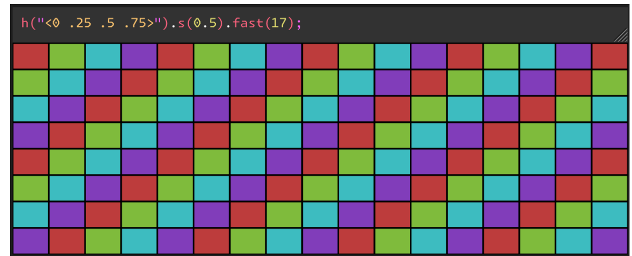

Figure 5: A visualized IdleCycles pattern.

One main motivation for this language was to learn more about the core mechanics of patterns, which were still a bit opaque to the author, even after having worked on Strudel for a few years. The other motivation was to find out what happens when certain seemingly important Uzu characteristics were abolished. For example, idlecycles uses floating point numbers instead of rational numbers to represent points in time (or space). Also, it doesn't have a concept of parts, only wholes. While the use of floating point numbers did not cause problems, the absence of parts caused certain patterns to have duplicated events. For that reason, the language could be considered an incomplete Uzu language, which might be revisited in the future.

7.7 mondo

As a continuation of idlecycles, *mondo* started as another series of interactive blog posts[4] to find a version of mini notation that can also compose functions. It was influenced by *Zwirn* and *Alternate Timelines for TidalCycles*. The control flow of *mondo* is very close to Strudel, where the transformations are applied from left to right:

```
s [bd cp*2] # fast <1 2>
```

When an expression is evaluated, a desugaring step removes all syntax sugar and leaves plain s-expressions. The above expression is turned into the following:

```
(fast (cat 1 2) (s (seq bd (fast 2 cp))))
```

This expression can be considered the syntax tree, where each list is a function call. *mondo* could be considered the "frontend" of an Uzulang, as its syntax tree can be fed to any "backend" that turns it into a pattern. In practice, mondo was connected to both *idlecycles* and *Strudel*. The above syntax tree ends up as the following Strudel code:

```
fast(cat(1, 2), s(seq('bd', fast(2, 'cp'))))
```

Absent of any method chaining, it doesn't look like typical Strudel code, but the expression is equivalent to `s("bd cp*2").fast("<1 2>")`, which is very similar to the initial mondo code with syntax sugar.

It still doesn't convey the unique characteristic of a single notation that can compose at any level. To illustrate that point, here's the mondo version of the original example in "Alternate Timelines for TidalCycles" (McLean 2021), quoted in §7.3:

[3]The Idlecycles chapters are available at https://garten.salat.dev/field/idlecycles.html

[4]The first post on mondo is available at https://garten.salat.dev/059-uzulang-I-s-expressions/

```
s [bd (# every 3 (# fast 4) sn)]
  # jux <rev (# iter 4)>
```

After the initial blog posts, mondo found its way into Strudel and is now an alternative, still experimental Uzu language syntax. The integration of mondo into Strudel leans on the stepwise family of functions, which allow certain mini notation semantics to be expressed as function compositions.

8 Concluding thoughts

Defining what makes an Uzulang, also helps us creatively imagine possibilities beyond Uzulangs. The representation of functions rather than data structures is a tradeoff, so one theme to explore is how differently a system would feel if it applied all the learnings from Uzulang development, to the manipulation of pattern as structured data.

Finally, while we have focussed on software implementations of Uzulangs in the above, we should also highlight the core importance of the wider community around it. After all, Uzulangs exist for music makers, who perform for dancing audiences. With live coding with Uzulangs going viral on social media, some end-users beginning to sustain their career using it, and local grass-roots communities developing weird and wonderful live coding practice, what live coding with an Uzulang *really* means in terms of culture and community is only beginning to take shape.

As traditional software engineering use of programming languages is up-ended by centralised scraping of code and training of autonomous agents, this is a special time for live coding practices that only really make sense in the context of fully human expression. Not only that, but as live coding has reached a quarter-century of development, younger artists are now using live coding languages that are older than them, with the knowledge and culture around them therefore becoming ancestral. Humans have been passing down encoded knowledge of patterns for millennia, through textiles, dance and music, so it is exciting to see live coding patterns slowly become part of that lineage.

9 Ethical Standards

This paper and the systems described have been written by humans, and shared under free/open source (GPL and AGPL) licenses. This is a collective, jointly-authored work, with all Uzulang implementers and other community members invited to contribute.

References

Bel, Bernard. 2001. "Rationalizing Musical Time: Syntactic and Symbolic-Numeric Approaches." In *The Ratio Book*, edited by Clarence Barlow. Feedback Studio.

Blackwell, Alan, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual.* MIT Press. https://doi.org/10.5281/zenodo.7383848.

Csound: A Sound and Music Computing System | Guide Books | ACM Digital Library. n.d.

McLean, Alex. 2004. "Hacking Perl in Nightclubs." *Perl.com*, August.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design.* https://doi.org/10.1145/2633638.2633647.

McLean, Alex. 2020. "Feedforward." *Proceedings of New Interfaces for Musical Expression* (Birmingham), July. https://doi.org/10.5281/zenodo.6353969.

McLean, Alex. 2021. "Alternate Timelines for TidalCycles." *Proceedings of the 6th International Conference on Live Coding* (Valdivia, Chile), December. https://doi.org/10.5281/zenodo.5788732.

McLean, Alex. 2024. "From Konnakol to Live Coding." *Proceedings of the 12th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (New York, NY, USA), FARM 2024, September, 36–41. https://doi.org/10.1145/3677996.3678290.

McLean, Alex, Raphaël Forment, Sylvain Le Beux, and Damián Silvani. 2022. "TidalVortex Zero." *International Conference on Computer Music (ICMC)* (Limerick, Ireland), April. https://doi.org/10.5281/zenodo.6456380.

Orlarey, Y., D. Fober, S. Letz, and M. Bilton. 1994. "Lambda Calculus and Music Calculi." *International Conference on Mathematics and Computing.*

Roberts, Charles, Matthew Wright, and JoAnn Kuchera-Morin. 2015. "Beyond Editing: Extended Interaction with Textual Code Fragments." *Proceedings of the International Conference on New Interfaces for Musical Expression* (Baton Rouge, Louisiana, USA), NIME 2015, May, 126–31.

Roos, Felix, and Raphaël Maurice Forment. 2025. *KabelSalat: Live Coding Audio-Visual Graphs on the Web and Beyond.* Zenodo. https://doi.org/10.5281/zenodo.15527772.

Roos, Felix, and Alex McLean. 2023. "Strudel: Live Coding Patterns on the Web." *Proceedings of the 7th International Conference on Live Coding* (Utrecht, Netherlands), April. https://doi.org/10.5281/zenodo.7842142.

Rosch, Eleanor, and Barbara B. Lloyd, eds. 1978. *Cognition and Categorization.* Routledge. https://doi.org/10.4324/9781032633275.

Swidan, Alaaeddin, and Felienne Hermans. 2023. "A Framework for the Localization of Programming Languages." In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*, 13–25. SPLASH-E 2023. New York, NY, USA: Association for Computing Machinery.

Victor, Bret. 2011. *Explorable Explanations.* https://worrydream.com/ExplorableExplanations.