Livecoding soundscapes in Kotlin with Compose Multiplatform

Repurposing a declarative UI framework for music production

Merlin Pahič

Agenda

1. Kotlin and Compose Multiplatform
2. Functional-declarative paradigm and its role in Livecoding
3. Introducing DaKapo – why repurpose a UI framework?
4. Fundamental building blocks of UI and music – parallels
5. State management – how does DaKapo work?
6. Implementing music building blocks
7. Expressing rhythms more conveniently
8. Connecting building blocks
9. Multiplatform support, optimisations and library integrations
10. Compose UI integration and visualisations

Kotlin and Compose Multiplatform

Compose Multiplatform is a declarative UI framework in the functional paradigm.

There are similarities with React; Compose is a Kotlin framework though.

Kotlin started out as Java with less boilerplate, and has become a flexible multi-paradigm language with support for platforms beyond the JVM and Android (Web, Desktop, iOS)

Livecoding and the functional-declarative paradigm

The functional-declarative paradigm is the foundation of some programming languages, as well as some UI frameworks (Compose, React).

Various livecoding libraries/frameworks also use this paradigm:

Haskell’s Tidal, JavaScript’s Strudel, Clojure’s Overtone

But none of these leverage a UI framework like Compose or React.

Introducing DaKapo

Leveraging Compose Multiplatform for audio generation and livecoding.

MIT licence

Early stages

Why repurpose a UI framework for music/livecoding?

It turns out a lot of features useful for UI development can be repurposed.

Lower barrier to entry for developers familiar with the UI framework.

UI framework does a lot of the work, like state management, animating values, dynamically adding/removing building blocks.

IntelliJ/Android Studio allow previewing UI right in the IDE with live updates.

UI: Fundamental building blocks

In frameworks like Compose (or React), fundamental building blocks are:

- Layout (e.g. rows and columns)
- UI components (e.g. buttons or text fields)
- Stateful values, leveraging state management features
- Animations (values changing over time)
- Effects (perform an action when a component enters or leaves the composition/DOM tree)
- Making values accessible from anywhere in the tree (e.g. theming)

Music: Fundamental building blocks

- Timing: playing simultaneously or in order, expressing rhythms
- Sounds, like those from a synth or playing samples
- Stateful values that update and change the music as it plays
- Values changing over time
- Scheduling/cancelling audio when things change
- Making controls (e.g. volume, BPM) available throughout the program

Managing state

Compose provides *State* (similar to React's *useState* hook)

Compose tree updates when state changes:

```
val synth by remember { mutableStateOf(Synth.Piano) }

Synth(synth = synth) { … }
```

Making it work: Effects

`DaKapo { … }`

Uses effects to schedule/cancel audio

Compose provides *DisposableEffect* (similar to React's *useEffect* hook)

Making values accessible throughout the Compose tree

Useful for things like volume or BPM

Compose provides *CompositionLocal* (similar to React's *Context*)

```
@Composable

fun AbsoluteVolume(volume: Float, sound: @Composable () ->
Unit): Unit =

CompositionLocalProvider(LocalVolume provides volume, sound)
```

Relative volume

```
@Composable
fun RelativeVolume(volume: Float, sound: @Composable () -> Unit) {
    val currentVolume = LocalVolume.current
    CompositionLocalProvider(
        LocalVolume provides volume * currentVolume,
        sound
    )
}
```

We can do the same thing for BPM or relative tempo.

Implementing building blocks: Synths and Samples

For now, basic implementation from the ground up.

Synths and samples are scheduled through effects. The blocks they are embedded in can modify the sound:

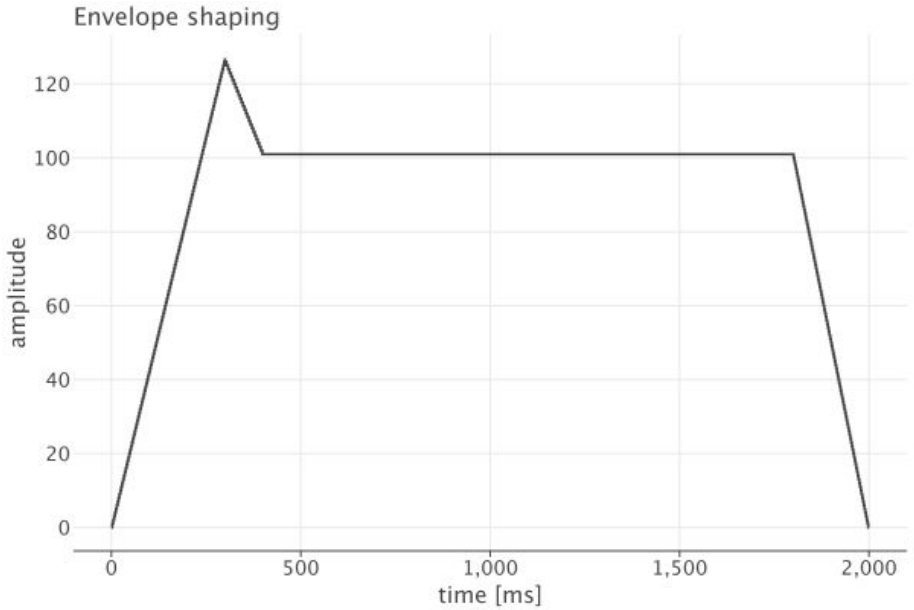
```
Bpm(60) {
    AbsoluteVolume(0.8) {
        Synth(Synth.Piano) {…}
    }
}
```

Requirement: Envelope shaping

Makes sounds more realisitc

E.g. ADSR envelope

What should the API for this look like?



We could use blocks

```
Envelope(attack = …, …) {

    Synth(…) {…}

}
```

Compose: Modifiers

Compose provides *Modifiers:*

```
Column(modifier = Modifier.fillMaxHeight()) { … }
```

Modifiers can be specified on any composable, and affect the way it renders.

Modifiers can be used for sizing, borders, padding etc.

DaKapo: Conditioners

DaKapo uses *Conditioners* in an analogy to Compose *Modifiers:*

```
Synth(conditioner = Conditioner.envelope(attack = …, …), …) {…}
```

Instead of:

```
Reverb(…) {
    Synth(…) {…}
}
```

We can write:

```
Synth(conditioner = conditioner.reverb(…), …) {…}
```

Basic timings and simultaneous play

We introduce timing primitives like *InParallel, CycleOver,* and *SplitEqually*

This allows us to express basic timings:

```
InParallel {
    A()
    CyleOver {
        B()
        SplitEqually {
            C()
            D()
        }
    }
}
```

Creating a drum machine

We can add functions like `Snare()` or `Cymbal()` to play drum samples.

Using a `Drums(machine = …) {…}` block, we can switch out the drum machine for one that plays different samples, or synthesises the drum sounds.

Under the hood, we'll again use *CompositionLocal.*

Expressing rhythms more conveniently

```
Rhythm("xy xx y", 'x' to { Snare() }, 'y' to { Cymbal()})
```

ConstraintLayout

In Compose, we can anchor UI components to other components instead of building grid-based layouts. This is sometimes much more convenient.

```
ConstraintLayout {
     val (button, text) = createRefs()
     Button(…,
         modifier = Modifier.constrainAs(button) {
             top.linkTo(parent.top)
         }
     ) { Text(text = "Button") }
      Text(
         text = "Text",
         modifier = Modifier.constrainAs(text) {
             top.linkTo(button.bottom)
         }
     )
 }
```

Connecting audio building blocks

We can use a similar mechanism to connect audio building blocks:

```
Connections { output ->
    val (generator, delay, reverb) = createRefs()
    SineWave(…, conditioner = Conditioner.connectAs(generator))
    Delay(…,
        conditioner = Conditioner.connectAs(delay) { accept(generator)}
    )
    Reverb(…,
        conditioner = Conditioner.connectAs(reverb) {
            accept(delay)
            feed(output)
        }
    )
 }
```

Supporting multiple platforms

Compose Multiplatform works on various platforms (Android, Desktop/JVM, Web, iOS)

On Desktop, we are using the Java Sound API

On Android, the AudioTrack API

Both allow synchronously pushing bytes into a stream for playback

Created async wrappers

16-bit PCM is well-supported across platforms; we'll leverage this to add iOS support as well as a JavaScript/WASM target

Reinventing the wheel, or leveraging powerful libraries?

Right now, implementing audio building blocks from scratch.

It’s been enjoyable, and a great way to learn the fundamentals.

But eventually, we’ll want something more powerful, e.g. *SuperCollider*
(at least on Desktop)

Requires carefully abstracting features and ensuring audio backends on various platforms support them consistently.

Optimisations

Integrating with a library like SuperCollider allows us to take advantage of its scheduling features.

Scheduling needs to be precise as small differences in timing matter in music.

But for now, there are ways to improve timing
(e.g. Audio thread priority on Android)

Integration with Compose UI

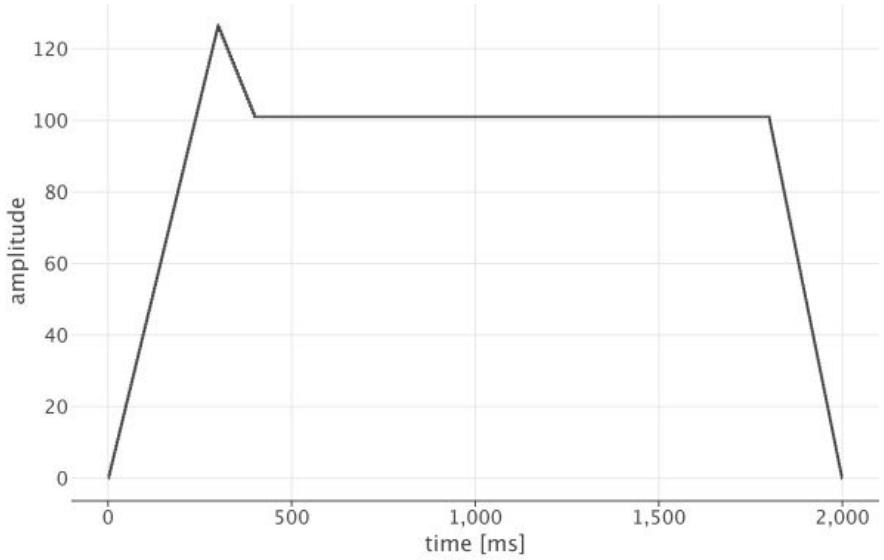
DaKapo block can be integrated anywhere in the Compose tree

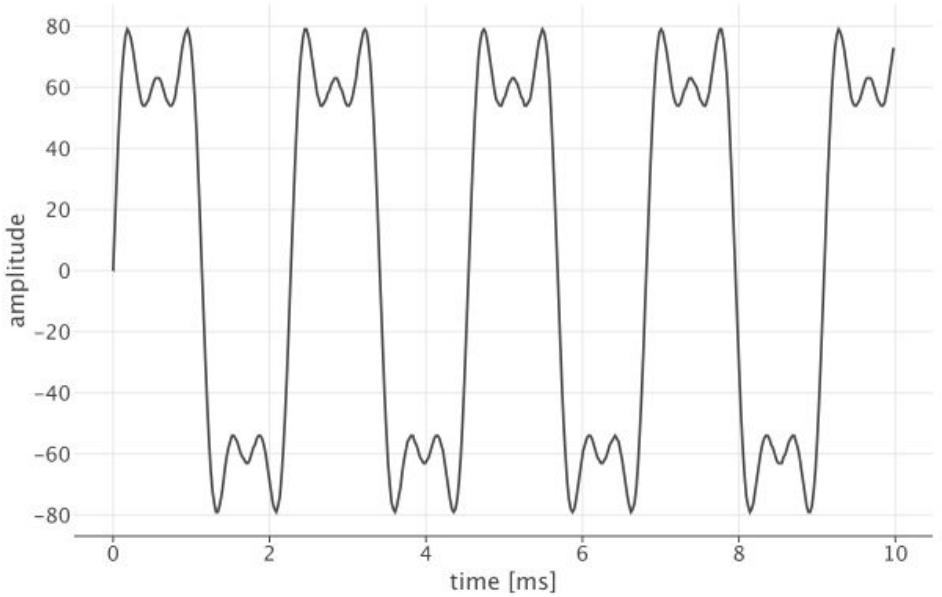
There’s really no requirement for Composables to be UI elements, but …

… they can be, e.g. to implement visualisations.

Visualisations

We can leverage the fact that we're using a UI framework already to create visualisations. Compose provides an integration with Kotlin's Let's Plot:





Thanks for listening – questions?

DaKapo will be available at https://github.com/O-O-Balance/DaKapo

Also, if you have done something similar, or have any expertise to share, please reach out!

Merlin Pahic