Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
11.12.2020
Deadline: 21.12.2020, 23:59

Functional Programming and Verification

Sheet 6

Exercise T6.1 Everything is a Fold

Using `foldr`, implement the following functions:

a) `length' :: [a] -> Integer` computing the length of a list
b) `map' :: (a -> b) -> [a] -> [b]` mapping a function over a list
c) `reverse' :: [a] -> [a]` reversing a list
d) `minimum' :: Ord a => [a] -> a -> a` computing the minimum of a given value and all values of a given list.
e) `fib :: Integer -> Integer` computing the Fibonacci numbers
f) `inits' :: [a] -> [[a]]` computing all prefixes of a list
g) Finish the given template for `squareOn :: (Eq a, Num a) => [a] -> a -> a` such that

$$\text{squareOn}\,[x_1,\ldots,x_n]\,y = \begin{cases} y^2, & \text{if } y \in \{x_1,\ldots,x_n\} \\ y, & \text{otherwise} \end{cases}$$

h) `compose :: [(a -> a)] -> a -> a` such that $\text{compose}\,[f_1,\ldots,f_n] = f_1(\ldots(f_n(\cdot)\ldots)$

Exercise T6.2 Everything is an Iteration

a) Write a function `iter :: Int -> (a -> a) -> a -> a` that takes a number n, a function f, and a value x, and applies f n-times with initial value x, that is `iter` n f x computes $f^n(x)$. A negative input for n should have the same effect as passing $n = 0$. For example, `iter 3 sq 2 = 256`, where `sq x = x * x`
b) Use `iter` to implement the following functions *without* recursion:
   a) Exponentiation: `pow :: Int -> Int -> Int` such that `pow` n $k = n^k$ (for all $k \geq 0$).
   b) The function `drop :: Int -> [a] -> [a]` from Haskell's standard library that takes a number k and a list $[x_1,\ldots,x_n]$ and returns $[x_{k+1},\ldots,x_n]$. You can assume that $k \leq n$.
   c) The function `replicate :: Int -> a -> [a]` from Haskell's standard library that takes a number $n \geq 0$ and a value x and returns the list $[\underbrace{x,\ldots,x}_{n\text{-times}}]$.
c) Write a function `iterWhile :: (a -> a -> Bool) -> (a -> a) -> a -> a` such that `iterWhile test f x` iterates `f` until `test x (f x)` is false, and then returns `x`.

d) Use `iterWhile` to implement a function `findSup :: Ord a => (a -> a) -> a -> a -> a` such that `findSup f m x` finds the largest value $f^n(x)$ that is at most `m` assuming that f is strictly monotonically increasing.

Exercise T6.3 (Optional) Everything is a Variable | Lambda | Application and That is All!

Datatypes like `Int`, `Integer`, etc. are overrated. In this exercise, we ask you to build your own natural numbers – à la Church – using only function abstractions and applications.

Our type of numeral will be `type ChurchNum a = (a -> a) -> a -> a`. Sounds crazy at first, but listen for a moment:

The idea is that a numeral can be encoded as the number of applications of a function f to a given initial value x. The numeral 0 is hence defined as `zero f x = x`, the numeral 1 as `one f x = f x`, the numeral 2 as `two f x = f (f x)`, etc.

a) Write a function `fromInt :: Int -> ChurchNum a` returning the Church numeral corresponding to a given `x :: Int`.

b) Write a function `toInt :: ChurchNum Int -> Int` such that `toInt . fromInt = id` for non-negative input values. Write a QuickCheck test.

c) Write a function `succ :: ChurchNum a -> ChurchNum a` returning the successor of a Church numeral.

d) Write a function `plus :: ChurchNum a -> ChurchNum a -> ChurchNum a` adding two Church numerals. Test whether `plus` corresponds to `(+)` defined on `Int` for non-negative inputs.

e) Write a function `mult :: ChurchNum a -> ChurchNum a -> ChurchNum a` multiplying two Church numerals. Test whether `mult` corresponds to `(*)` defined on `Int` for non-negative inputs.

f) Define whatever else comes up to your mind using Church numerals. Get inspired on Wikipedia.

Homework

You need to pass all tests to collect a coin.

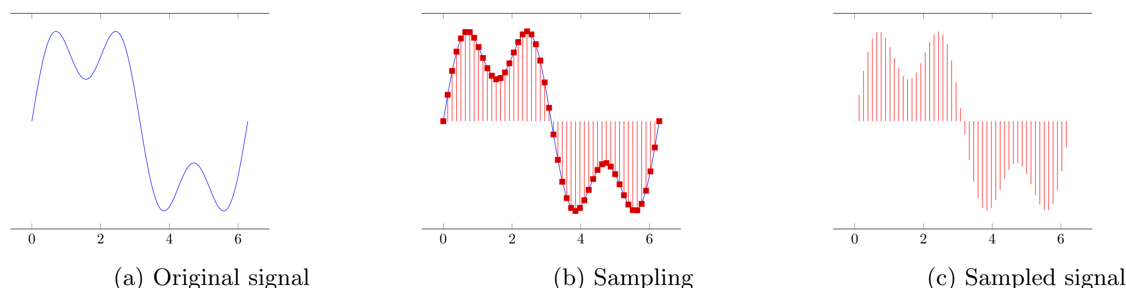
Exercise H6.1 The Haskell School of Music

In this exercise, we will use Haskell to generate some sweet music! More specifically, we use a technique called subtractive synthesis to build a synthesizer that can generate and manipulate audio signals.

Background Mathematically, a sound can be represented as a continuous function that maps every point in time to an amplitude value in the interval $[-1, 1]$. Thinking in digital terms, however, we have to rethink this a bit: to accurately represent an audio signal in a discrete way,

we sample the signal at a fixed interval, determined by the **sample rate**. A common sample rate is, for example, 44100 samples/second, as used for most consumer audio, e.g. for CDs.

This technique is called Pulse-code modulation, (or PCM for short). You can find a visualisation in Figure 1. Many uncompressed audio file formats today, such as WAVE (`.wav`), are indeed PCM-encoded.



(a) Original signal (b) Sampling (c) Sampled signal

Figure 1: Digital sampling of an analog signal

In Haskell we represent this discrete signal as a `List` of `Double`-valued samples:

- `type Sample = Double`
- `type SampledSignal = [Sample]`

Oscillators The foundation of every subtractive synthesizer is an oscillator: a component that generates an oscillating waveform (e.g. sine wave) at a specific frequency. Analog synthesizers use electronic oscillators, but we can simulate the same effect using Haskell functions! Old synthesizers (like the Minimoog) produce a variety of simple waveforms for their signals. The most important ones that we are going to use are displayed in Figure 2.

So how does the synthesizer know what frequencies to generate? Digital protocols like MIDI specify fixed notes that have predefined frequencies. Our synthesizer follows a similar approach by using the following formula:

$$f(n) = 440 * (2^{n/12})$$

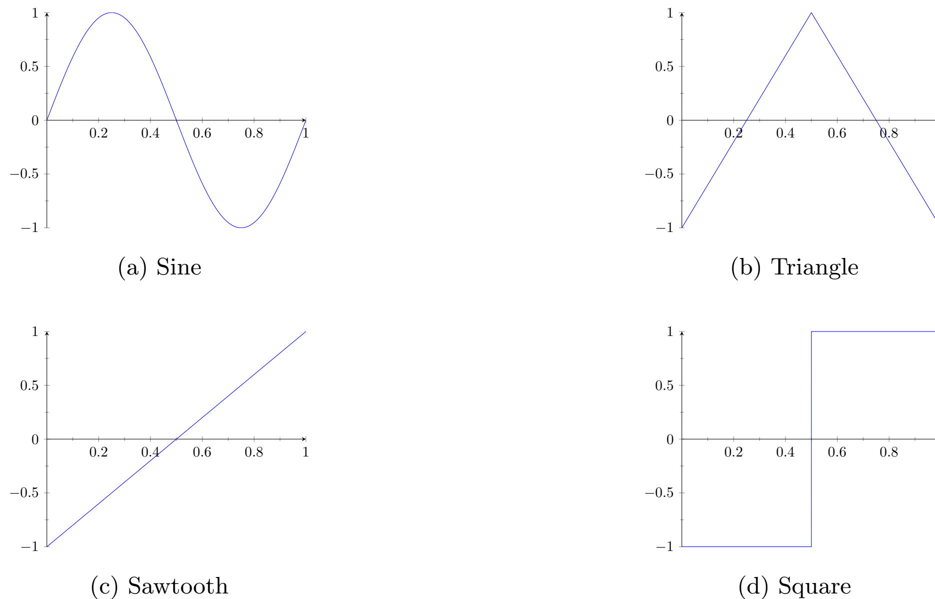
$f(n)$ calculates the frequency of a tone that is n semitones away from A_4. A tone *lower* than A_4 has a negative distance. e.g. A_3 has a distance of -12 from A_4. Our synthesizer will work with the aforementioned distances in semitones to A_4, so don't worry about too much about musical notation.

Tasks

1. We define the type `type Signal = Seconds -> Sample` (cf. template file). Write the following functions that are expected to return a single period of the signals as depicted in Figure 2. Since these functions only represent a single period, they only need to be defined on the interval $[0, 1)$.

   a) `sinPeriod :: Signal`

   b) `triPeriod :: Signal`



(a) Sine (b) Triangle

(c) Sawtooth (d) Square

Figure 2: Common synthesizer waveforms

c) `sawPeriod :: Signal`

d) `sqwPeriod :: Signal`

Make sure that your samples stay in the interval $[-1, 1]$, they closely align with the graphs from Figure 2, and that your sawtooth and triangle waveforms are perfectly piecewise linear.

After the oscillators have created the waveforms, we can start manipulating the returned signals to generate rich and dynamic sounds. We start with a so-called ADSR envelope that is used to modify the amplitude of the signal. It creates a sound that first takes a bit of time to reach its maximum amplitude, then fades, stays steady, and finally fades again until the amplitude is zero (see Figure 3). This is used to ensure that there are no unexpected jumps from/to zero, which generally tend to produce annoying "clicky" sounds. Instead, we obtain a smooth audio wave as depicted in Figure 4.

2. We define `type ADSR = (Seconds, Seconds, Sample, Seconds)`. Write a function `adsr :: ADSR -> Seconds -> Signal -> Signal` such that `adsr (attack, decay, sustain, release) duration signal` modifies the amplitude of a given signal with a given duration as shown in Figure 4. In this example the ADSR-paramters are `(2.0, 1.0, 0.5, 2.5)` and the duration is 10. While `attack`, `decay` and `release` control the length of the respective phases, `sustain` specifies the amplitude of the signal during the sustain phase. In our synthesizer, the release happens *before* the note ends, which means you need to apply it to the end of the signal that the function receives. Make sure that the transitions of attack, decay, and release are perfectly linear.

3. Next we implement an oscillator. An oscillator is defined as a function that returns a

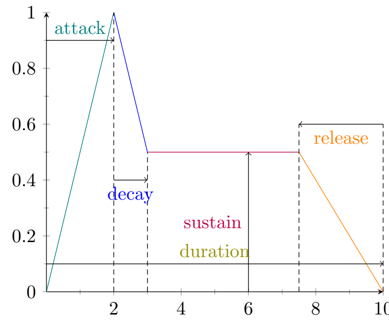


Figure 3: Example of Attack, Decay, Sustain, Release and Duration

(a) Input

(b) Output

Figure 4: Input and Output signals of `adsr`

signal for a given semitone and duration. We thus define `type Oscillator = Semitone -> Seconds -> Signal`.

Write a function `osc :: Signal -> ADSR -> Oscillator` that receives one of the signal functions defined earlier (defining the shape of a single period) as well as ADSR parameters and returns an oscillator. It has to repeat the input signal using the right frequency (use aforementioned function f to calculate the frequency of the semitone) and then has to apply the ADSR to this signal. To repeat the wave form in every period, the function `floor` can be useful.

4. So far, we have worked with signals represented as functions of type `Signal = Seconds -> Sample`. In this task, you can assume that the signal is already sampled. To encode sampled signals, we use the type `SampledSignal = [Sample]`.

   We can create a polyphonic synthesizer by using multiple sources for notes and multiple oscillators (cf Figure 5). For this, we need a function that mixes signals.

   Write a function `mix :: [SampledSignal] -> SampledSignal` that normalizes the signals (divides the samples of each signal by the number of signals to be mixed) and combines them into a single signal by adding them together. The resulting signal should have the same length as the longest signal of the input.

Making Music We can now put all components together to create a synthesizer. Don't worry, we have already done this for you – you can focus on the fun part (see below).

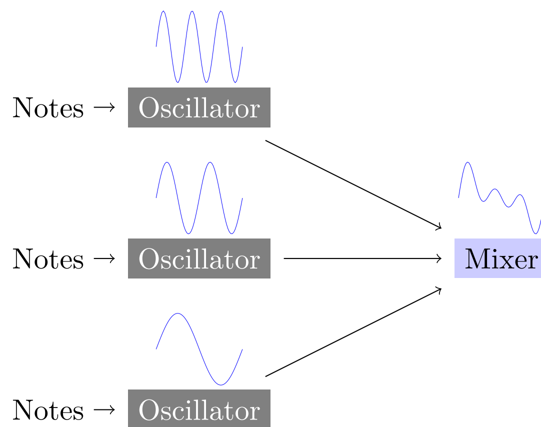


Figure 5: Polyphonic Synthesizer

Figure 6: Example MIDI file structure

Our program first reads a bunch of notes from a MIDI file. Figure 6 shows the structure of such a file. MIDI files consist of multiple tracks, each containing a bunch of notes. For those who are more familiar with MIDI: we only use a small subset of the MIDI format, ignoring almost all MIDI messages except for `Note On` and `Note Off`. Velocities are also ignored. Some MIDI editors also create empty tracks storing additional information. Our synthesizer will automatically ignore those.

The notes of the MIDI file will be passed to the function

```
notesToSignal :: (Oscillator -> [Note] -> SampledSignal)
              -> [[Note]] -> SampledSignal}
```

which, per default, uses the `playNotes` function in `lib/Synth.hs` to produce a signal; however, you don't have to worry about `playNotes` to customize the synthesizer. Instead you just have to modify `notesToSignal`. In this function, you can map the MIDI file's tracks to different oscillators and use `mix` to combine them into a signal. You can even apply effects to the produced signal to further enhance your tune.

Our program will then convert your signal into a WAV file. To run your synthesizer use:

```
stack run synth <filepath to MIDI file> <filepath to WAV file>
```

Wettbewerb: The FPV-Grammy Award It is time to get *really* creative. We provided you with a selection of DSP-effects (Digital Signal Processing). You can use and mix them into the output of your synthesizer – or even better, write your own effects!

Use `addEffects :: [DSPEffect] -> Signal -> Signal` or `applyEffectToInterval :: (Seconds, Seconds) -> Signal -> DSPEffect -> Signal` from `Effects` to add your effects to signals and create a musical masterpiece.

Here are some ideas to get started with this:

- Amplitude Modulation
- Phase Shifting
- Time Modulation
- Bit crushing
- Echo & Delay

The MCs and tutors – which will act as mini-MCs for the upcoming Wettbewerb – expect no less than award-worthy remixes using stunning effects and mesmerising oscillators. Your final composition will be evaluated based on the originality and technical ingenuity of implemented oscillators and effects as well as overall melodiousness.

You can work on one of the the MIDI files provided in the repository or, if you wish, use another file of your choice. Note that just submitting an interesting MIDI file won't cut it.

This exercise was designed and implemented in coorporation with our tutors. Special thanks to all of them!

> The ultimate goal for me in making music, or at least one of the main goals for me, is to create memorable melodies. That goal is there regardless of the tools we have.
>
> — Koji Kondo