Euterpea in Agda

Yiyun Liu

University of Maryland

05/14/2019

Table of Contents

What is Euterpea

Euterpea is an EDSL originally designed for synthesizing music. It is covered in the Haskell School of Music, a book which introduces basic music synthesis and functional programming at the same time. Euterpea provides the programmer a set of primitives, and a set of combinators which can form more complex structures.

What is Euterpea

Euterpea is an EDSL originally designed for synthesizing music. It is covered in the Haskell School of Music, a book which introduces basic music synthesis and functional programming at the same time. Euterpea provides the programmer a set of primitives, and a set of combinators which can form more complex structures.

Primitives

```
data Primitive = Note Dur Pitch
| Rest Dur
```

- There are two types of Primitives: `Note` and `Rest`

Primitives

```
data Primitive = Note Dur Pitch
| Rest Dur
```

- There are two types of Primitives: `Note` and `Rest`
- Each `Note` has a `Duration` and a `Pitch`.

Primitives

```
data Primitive = Note Dur Pitch
| Rest Dur
```

- There are two types of Primitives: `Note` and `Rest`
- Each `Note` has a `Duration` and a `Pitch`.
- Rest represents a pause in music. It does not produce any sound, therefore the `Rest` constructor takes only a `Duration` is its parameter.

Primitives

```
data Primitive = Note Dur Pitch
| Rest Dur
```

- There are two types of Primitives: `Note` and `Rest`
- Each `Note` has a `Duration` and a `Pitch`.
- Rest represents a pause in music. It does not produce any sound, therefore the `Rest` constructor takes only a `Duration` is its parameter.
- Euterpea has a separate `Music` data type. In order to make use of the `Primitive`s, we need to inject them into `Music` type with the constructor `Prim :: Primitive -> Music`

Compositions

```
t251' :: Music Pitch
t251' =
  let dMinor7 = d 3 hn :=: c 4 hn :=: c 5 hn :=:
          f 5 hn :=: c 6 hn
      gDom7 = g 3 hn :=: f 4 hn :=: b 4 hn :=:
          f 5 hn :=: b 5 hn
      cMajor7 = c 3 hn :=: b 3 hn :=: b 4 hn :=:
          e 5 hn :=: b 5 hn
   in dMinor7 :+: gDom7 :+: cMajor7
```

Compositions

```
t251' :: Music Pitch
t251' =
  let dMinor7 = d 3 hn :=: c 4 hn :=: c 5 hn :=:
         f 5 hn :=: c 6 hn
      gDom7 = g 3 hn :=: f 4 hn :=: b 4 hn :=:
         f 5 hn :=: b 5 hn
      cMajor7 = c 3 hn :=: b 3 hn :=: b 4 hn :=:
         e 5 hn :=: b 5 hn
   in dMinor7 :+: gDom7 :+: cMajor7
```

- `:=:` is used for parallel composition (played at the same time)

Compositions

```
t251' :: Music Pitch
t251' =
  let dMinor7 = d 3 hn :=: c 4 hn :=: c 5 hn :=:
          f 5 hn :=: c 6 hn
      gDom7 = g 3 hn :=: f 4 hn :=: b 4 hn :=:
          f 5 hn :=: b 5 hn
      cMajor7 = c 3 hn :=: b 3 hn :=: b 4 hn :=:
          e 5 hn :=: b 5 hn
   in dMinor7 :+: gDom7 :+: cMajor7
```

- `:=:` is used for parallel composition (played at the same time)
- `:+:` is used for sequential composition (played one after the other).

Compositions

```
t251' :: Music Pitch
t251' =
  let dMinor7 = d 3 hn :=: c 4 hn :=: c 5 hn :=:
        f 5 hn :=: c 6 hn
      gDom7 = g 3 hn :=: f 4 hn :=: b 4 hn :=:
        f 5 hn :=: b 5 hn
      cMajor7 = c 3 hn :=: b 3 hn :=: b 4 hn :=:
        e 5 hn :=: b 5 hn
   in dMinor7 :+: gDom7 :+: cMajor7
```

- `:=:` is used for parallel composition (played at the same time)
- `:+:` is used for sequential composition (played one after the other).
- The `dMinor` below uses `:=:` to produce seventh chords.

Compositions

```
t251' :: Music Pitch
t251' =
  let dMinor7 = d 3 hn :=: c 4 hn :=: c 5 hn :=:
        f 5 hn :=: c 6 hn
      gDom7 = g 3 hn :=: f 4 hn :=: b 4 hn :=:
        f 5 hn :=: b 5 hn
      cMajor7 = c 3 hn :=: b 3 hn :=: b 4 hn :=:
        e 5 hn :=: b 5 hn
   in dMinor7 :+: gDom7 :+: cMajor7
```

- `:=:` is used for parallel composition (played at the same time)
- `:+:` is used for sequential composition (played one after the other).
- The `dMinor` below uses `:=:` to produce seventh chords.
- We can assemble those chords with `:+:` to form the II-V-I Jazz progression.

Modifiers

Euterpea also introduces a few modifiers into the object language. Those modifiers, when interpreted, change the pitch or duration of the music.

A Snippet from Canon in D

```
canonInD = tempo (90/120) (mainVoice :=: bassLine)

bassPhrase :: Music Pitch
bassPhrase = line . fmap ($ hn) $
    [d 3, a 2, b 2, fs 2, g 2, d 2, g 2, a 2]

bassLine :: Music Pitch
bassLine = times 5 bassPhrase -- & keysig D Major

mainVoice :: Music Pitch
mainVoice = line [phrase0, phrase1, phrase2
                 ,phrase3, phrase4] -- & keysig D Major
```

A Snippet from Canon in D

```haskell
where
phrase0 = rest (dur bassPhrase)
phrase1 = line . fmap ($ hn) $
    [fs 5, e 5, d 5, cs 5, b 4, a 4, b 4, cs 5]
phrase2 = phrase1 :=: (fmap ($ hn)
    [d 5, cs 5, b 4, a 4, g 4, fs 4, g 4, a 4] & line)
phrase3 = phrase2 :=: (fmap (\n -> rest qn :+: n qn)
    [a 4, a 4, fs 4, fs 4, d 4, d 4, d 4, g 4] & line)
phrase4 = phrase2 :=: ((fmap (\(n0,n1) ->
    line [rest en, n0 en, n1 en, n0 en, rest qn, n0 qn])
    [(a 4, d 5),(fs 4, b 4), (d 4, g 4)] & line) :+:
    line [rest en, d 4 en, g 4 en, d 4 en, rest qn, g 4 qn])
```

Table of Contents

Informal Equational Reasoning from the Book

- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0`

Informal Equational Reasoning from the Book

- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0`
- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3`

Informal Equational Reasoning from the Book

- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0`
- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3`
- What is the relation between these two pieces of `Music`?

Informal Equational Reasoning from the Book

- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0`
- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3`
- What is the relation between these two pieces of `Music`?
- Even though they are not syntactically equal (since constructors do not reduce), they are semantically equal (e.g. generating the same sound when played).

Informal Equational Reasoning from the Book

- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0`
- `hNote qn p1 :+: hNote qn p2 :+: hNote qn p3`
- What is the relation between these two pieces of `Music`?
- Even though they are not syntactically equal (since constructors do not reduce), they are semantically equal (e.g. generating the same sound when played).
- The book admits the semantic equality implicitly for equational reasoning.

Equality Semantics

- In Agda, we need to distinguish between semantic and syntactic equivalence. To define the former, we need to define an equivalence relation.

Equality Semantics

- In Agda, we need to distinguish between semantic and syntactic equivalence. To define the former, we need to define an equivalence relation.
- First, we construct a unidirectional arrow.

Equality Semantics

- In Agda, we need to distinguish between semantic and syntactic equivalence. To define the former, we need to define an equivalence relation.
- First, we construct a unidirectional arrow.
- Second, we wrap the unidirectional arrow with a symmetric closure, since we want the equality to go in both directions.

Equality Semantics

- In Agda, we need to distinguish between semantic and syntactic equivalence. To define the former, we need to define an equivalence relation.
- First, we construct a unidirectional arrow.
- Second, we wrap the unidirectional arrow with a symmetric closure, since we want the equality to go in both directions.
- Wrapping the symmetric closure with a transitive closure, and we obtain the equivalence relation we wanted.

Symmetric Closure

- Given a Poset (not necessarily total order) $\mathbb{P}$

Symmetric Closure

- Given a Poset (not necessarily total order) $\mathbb{P}$
- We define a new relation $<>$ such that

$$a <> b \iff a < b \vee b < a$$

Symmetric Closure

- Given a Poset (not necessarily total order) $\mathbb{P}$
- We define a new relation $<>$ such that

$$a <> b \iff a < b \vee b < a$$

- Intuitively, if $a <> b$ is true, we know that a and b are comparable.

Transitive Closure

Same as what we learned in class.

Equivalence Closure

```
EqClosure : ∀ {a ℓ} {A : Set a} → Rel A ℓ → Rel A (a ⊔ ℓ)
EqClosure _∼_ = Star (SymClosure _∼_)
```

- Equivalence Closure is defined as the composition of the ReflexiveTransitive Closure (`Star`) and Symmetric Closure (`SymClosure`)

Table of Contents

Definitions

```
data Primitive (A : Set) : Set where
  Note : Dur → A → Primitive A
  Rest : Dur → Primitive A

data Music (A : Set) : Set where
  Prim : (Primitive A) → Music A
  _:+:_ : Music A → Music A → Music A
  _:=:_ : Music A → Music A → Music A
  Modify : Control → Music A → Music A
```

Definitions

```
data Primitive (A : Set) : Set where
  Note : Dur → A → Primitive A
  Rest : Dur → Primitive A

data Music (A : Set) : Set where
  Prim : (Primitive A) → Music A
  _:+:_ : Music A → Music A → Music A
  _:=:_ : Music A → Music A → Music A
  Modify : Control → Music A → Music A
```

Dur is a synonym of the set of natural numbers. While the set of natural numbers is clearly not a good representation of the duration, it has the nice semiring and lattice properties which will be handy in the proofs. Rational numbers share the same properties but they are not well-supported in the Agda standard library.

One-step Relation

The relation music-step captures the basic algebraic properties of Music. We need to apply the EqClosure to support multiple-steps reasoning. The definition is mutually recursive because we want a more useful congruence axiom.

```
mutual
  data music-step : Music A → Music A →  Set where
    tempo-mult : (r1 r2 : Dur)  (m : Music A) →
      music-step
        (Modify (Tempo r1) (Modify (Tempo r2) m))
        (Modify (Tempo ( r1 * r2 )) m)

    trans-add : (p1 p2 : AbsPitch)  (m : Music A) →
      music-step
        (Modify (Transpose p1) (Modify (Transpose p2) m))
        (Modify (Transpose ( p1 + p2 )) m)
```

One-step Relation Cont.

```
-- ...
  :+:-cong : {m m' n n' : Music A}  →
             -- music-step m m' ?
             m ≈ m' →
             n ≈ n' →
             music-step (m :+: n) (m' :+: n')

  :=:-cong : {m m' n n' : Music A}  →
             m ≈ m' →
             n ≈ n' →
             music-step (m :=: n) (m' :=: n')
music-equiv : Rel (Music A) Level.zero
music-equiv = EqClosure music-step
private
  _≈_ = Setoid._≈_ (setoid (music-step))
```

Eliminating Rest 0

`Rest 0` is essentially a neutral element in the algebra of Music. Some of them are redundant and can be optimized away. The function `optimize-take2` (suggesting 1 failed attempt) eliminates all redundant `Rest 0`s. Clearly, the syntactic equality is no longer true, but that's the very reason why we defined the semantic equality _≈_. With the equivalence relation we defined, we can reason about the soundness of the optimization function.

```
optimize-take2 : (m : Music A) → Music A
optimize-take2 (Prim x) = Prim x
optimize-take2 (m :+: m1) with
  optimize-take2 m | empty-music? (optimize-take2 m)
optimize-take2 (m :+: m1)
  | .(Prim (Rest 0)) | yes empty = optimize-take2 m1
-- ... rest of the definition omitted
```

Soundness Proof

```
optimize-sound-take2 : (m : Music A)
  → (m ≈ optimize-take2 m)
```

The soundness proof and the definition of optimization make extensive use of the view pattern:

```
data Empty-Music? : Music A → Set where
  empty : Empty-Music? (Prim (Rest 0))

empty-music? : (m : Music A) → Dec (Empty-Music? m)
```

In the definition of the optimization function, we could use a catch-all pattern to simply the definition. However, Agda does not "remember" the catch-all pattern in the proofs which make use of the definition and would therefore require us to explicitly destruct all terms. The number of cases grows exponentially with respect to the number of parameters. The view pattern can mitigate the issue by reducing the base of the exponential function.

Idempotence Proof

One extra property of the optimization function is idempotence. If the optimization invoked twice produces a result that is different from optimization invoked only once, that would imply our optimization is not exhaustive.

```
optimize-idempotent-take2 : (m : Music A) →
  optimize-take2 m ≡ optimize-take2 (optimize-take2 m)
```

Note how the theorem switches from semantic equality we used for soundness proof to syntactic equality.

Fin