```
class Tuple a => Sigs a
```

Spectrums (Spec)

We can extract a spectrum from the signal. It's an advanced type. The simplest function to create a spectrum is:

```
toSpec   :: Sig -> Spec
fromSpec :: Spec -> Sig
mapSpec  :: (Spec -> Spec) -> Sig -> Sig
```

With `Spec` we can apply spectral transformations to signal. we can create a vocoder effect with it for instance or scale a pitch or crossfade between several timbres.

We can interpolate between several signals:

```
cfdSpec :: Sig -> Spec -> Spec -> Spec
cfdSpec4 :: Sig -> Sig -> Spec -> Spec -> Spec -> Spec -> Spec
cfdsSpec :: [Sig] -> [Spec] -> Spec
```

To scale the pitch there are handy shortcuts:

```
scaleSpec :: Sig -> Sig -> Sig
scalePitch :: Sig -> Sig -> Sig
```

`scaleSpec` scales the frequency of the signal in Hz ratios but `scalePitch` does it in semitones.

If we have a spectrum we can process it with many functions from the module Spectral processing.

Arrays (Arr)

We can create arrays of values. The data type of array is parametrized with index and value. This typing scheme prevents us from reading or writing the wrong values to the arrays although it doesn't prevents us from out of bounds errors.

```
data Arr ix a
```

Notice that the data types for indexes and values can be tuples. It let's us easily create multidimensional array. If we want say 2D array we can use pairs as indexes.

Creation of arrays

Arrays can be global and local. Local arrays are accessible only within the body of single Csound instrument where they are created. The scope translated to Haskell is somewhat obscure. The global arrays are accessible at any point of the code.

We can create arrays with functions:

```
newLocalArr  :: Tuple a => [D] -> [a] -> SE (Arr ix a)
newGlobalArr :: Tuple a => [D] -> [a] -> SE (Arr ix a)
```

They accept the list of dimensions and list of initial values. Also arrays can contain audio or control rate signals. The aforementioned functions create audio-rate signals. If we want to create control rate signals we should use the functions:

```
newLocalCtrlArr :: Tuple a => [D] -> [a] -> SE (Arr ix a)
newGlobalCtrlArr :: Tuple a => [D] -> [a] -> SE (Arr ix a)
```

Read and write operations

To read and write the values from array we have two functions:

```
writeArr :: (Tuple ix, Tuple a) => Arr ix a -> ix -> a -> SE ()
```

```
readArr  :: (Tuple a, Tuple ix) => Arr ix a -> ix -> SE a
```

We can modify the value in the arry with function:

```
modifyArr :: (Tuple a, Tuple ix) => Arr ix a -> ix -> (a -> a) -> SE ()
```

Type synonyms for often used array data-types

To save some typing there are some aliases defined for most frequntly used array data types:

```
type Arr1  a = Arr Sig a
type DArr1 a = Arr D a

type Arr2  a = Arr (Sig, Sig) a
type DArr2 a = Arr (D, D) a

type Arr3  a = Arr (Sig, Sig, Sig) a
type DArr3 a = Arr (D, D, D) a
```

Arrays that are parametrized with constant index (like `DArr1`) can be manipulated only at a single moment. We can only read and write constants to it.

If an array is parametrized with signal index (like `Arr1`) it can be manipulated continuously. We can read and write signals to it.

Also to help the type inference we can use the functions that do nothing with the values (just pass them through) but they have strict data type so that type inference can derive the desired data type:

```
arr1  :: SE (Arr Sig a) -> SE (Arr Sig a)
darr1 :: SE (Arr D a) -> SE (Arr D a)

arr2  :: SE (Arr (Sig, Sig) a) -> SE (Arr (Sig, Sig) a)
darr2 :: SE (Arr (D, D) a) -> SE (Arr (D, D) a)

arr3  :: SE (Arr (Sig, Sig, Sig) a) -> SE (Arr (Sig, Sig, Sig) a)
darr3 :: SE (Arr (D, D, D) a) -> SE (Arr (D, D, D) a)
```

Csound opcodes

In Csound there are plenty of opcodes to work with arrays. Almost all of them are supported. We can find out the complete list at the documentation for the module `Csound.Types` (see section for `Arrays`). Many functions are dedicated to manipulate spectral data.

Copy vs Allocation

Some peculiarity of transition form Csound to Haskell way of thinking lies in array functions. In the Csound almost all array functions can perform two different operations. Thy are overloaded. If we write:

```
kOut[] array_operation kWin
```

It can do two distinct operations:

- It can create new array if the value `kOut` was not previously initialized
- It can copy the data of the result of operation to the array `kOut` if it was already allocated.

In Haskell we often find two operations coresponding to the single Csound operation. Take for example the function `fft` . It performs fast Fourier transform. In Haskell we have two operations:

```
type SpecArr = Arr Sig Sig

fftNew  :: SpecArr -> SE SpecArr
fftCopy :: SpecArr -> SpecArr -> SE ()
```

The function `fftNew` allocates new array. But `fftCopy` just copies the data to existing array. Notice how the roles of the functions are signified with the signatures.

Functional traversals and folds

There are special functions that make traversal and folding very easy.

Traverse

We can traverse all elements in the array with function:

```
foreachArr :: (Tuple ix, Tuple a) => Arr ix a -> ((ix, a) -> SE ()) -> SE ()
foreachArr array proc
```

It takes an array and procedure that is defined on pairs of index and the value. These procedure is applied to all elments in the array. Notice that it can be applied to arrays of any sizes. All of them are going to be processed in the uniform way.

There are two useful special cases for 2D arrays:

```
forRowArr, forColumnArr :: Tuple a => Sig -> Arr Sig2 a -> ((Sig, a) -> SE ()) -> SE ()

forRowArr rowId array proc
```

They traverse only specific rows or columns. The index of the row (column) is the first argument of the function.

Fold

We can fold the array with the function. The process of folding is a traversal with value accumulation.

```
foldArr :: (Tuple ix, Tuple a, Tuple b) =>
     ((ix, a) -> b -> SE b) -> b -> Arr ix a -> SE b
```

The `foldArr` function takes a procedure that updates the result of type `b` based on the current index and value and the value of accumulator from the previous step. Also it takes initial value for accumulator and array.

There are specific foldfunctions to fold on rows and columns of 2D matrix:

```
foldRowArr, foldColumnArr
  :: (Tuple a, Tuple b) =>
     ((Sig, a) -> b -> SE b) -> b -> Sig -> Arr Sig2 a -> SE b
```

Init vs control rate traversals

In Csound there is a distinction between initial pass of the instrument. When everything gets initialized and control rate. When the audio goes on and we can control it. The aforementioned traversal functions work at control rate. But it's useful to be able to run them at init pass. To do it we have to use special variants of them with suffix `D`. The `D` is a synonym for constant number in the library that gets initialized and never changed at the control rate. So we have the functions:

```
foreachArrD     forRowArrD     forColumnArrD
foldArrD        foldRowArrD    foldColumnArrD
```

- <= Introduction
- => Rendering Csound files
- Home