Fuzzifier

Safe Haskell	Trustworthy
Language	Haskell2010
Extensions	DataKinds

Fuzzifier provides both the combinational and sequential definitions of a simple fuzzifier cirtcuit.

It is also meant to serve as a detailed insight into how things are declared in CλaSH, as well as a guideline for how most of all the other circuit specifications should look.

NOTE: for support of configuration environment; everything found below operates under the Reader Monad. One may simply mentally replace `Reader Config` xyz with xyz for all intents and purposes (as was in fact done in the documentation whenever Reader action types were referenced).

| In CλaSH; one may describe their design either as a combinational (and thus to be imagined as pure) circuit or a sequential (synchronized, single global clock) circuit. Pure code is always to be desired, so the library mostly permits the writing of the combinational design and the automatic remodelling as a sequential one by the library's functions. This will be our favored approach, as we shall see shortly.

Documentation

```
fuzzifierT :: Reader Config (Int -> FuzzySet)
```

fuzzifierT is the function representing a combinational fuzzifier. It takes a value and returns the **linear** `FuzzySet` associated to that value. It is the `Reader Config` which returns a component of type `Int` -> `FuzzySet`. It is later used to create an associated component type. It depends on the config keys `fuzzificationDelta` and the `totalSpace`. A `fuzzifierT` can be imagined to work as indicated in the following **pseudo**code:

```
fuzzifierT :: Int -> FuzzySet
fuzzifierT n = do
   let d = fuzzificationDelta
   -- the list which we'll map over, which is initialised with
   -- consecutive numbers to aid with result generation:
   let l = [1, 2..totalSpace]

   -- the function we shall map over said list, which returns the degree
   -- membership for that particular spot.
   let f m = if n is not between (n - d) (n + d)
             then return 0
             else return 100 - floor (100 * n * (abs (n - m)) / (d * n))

   return map f l
```

Note that because we are only mapping a function, so our definition is bounds-safe (as all Haskell is (on Liists, at least...)), and handles all edge-cases.

```
fuzzifier :: Reader Config (Signal Int -> Signal FuzzySet)
```

Now, in order to obtain a sequential cirtcuit modeling the behavior of our combinational circuit from above, we use the general Moore machine modeling helper function CλaSH provides in its Prelude.

Here is the moore machine modeler's type signature:

```
--          trans. f.       out. f.   i.s.      output circ.
moore :: (s -> i -> s) -> (s -> o) -> s -> Signal i -> Signal o
```

We are able to tell that our new design will be modeled as a sequential circuit by the fact that it operates on values of type Signal. It is the mark that all synchronous sequential circuits bear, and really all that differentiates them from their combinational counterparts in terms of behavior (the values packed inside a Signal are processed in the same way as the combinational model does it).

One may notice however some added complexity with the call to the moore function. This is due to the way a Moore machine is represented in CλaSH. The formal definition of Moore machines is preffered, and as such, we must provide the transfer and output functions (tf and of) for the machine. The transfer function describes the current state and the input combine into the next state of the machine, while the output function describes how the Moore machine decides its output from its current state. Lastly, an initial state must be provided, and then we get back a sequential cirtcuit-modelling function which takes a Signal of the input type and returns a Signal of the output type.

Bearing these in mind, our implementation of a fully-fledged sequential fuzzifier would look similar to the following:

```
fuzzifier :: Signal Int -> Signal FuzzySet
fuzzifier = moore tf id initial
             where tf s i = fuzzifierT i
                   initial = replicate totalSpace 0
```

Our transfer function ignores the current state parameter `s` and and just applies fuzzifierT to its input `i` and returns that as a new state. The output function only needs to returns the current state, as the transfer function did all the work. As such, we use the identity function for it. The initial state represents an empty FuzzySet.

The resulting Moore machine may now be instantiated and used in test benches in other parts of the project. For a more detailed look on running our designs, please refer to the FuzzifierTestbench module's documentation.

testFuzzifierT

`:: Int`	the total range we're working on.
`-> Int`	the fuzzification delta.
`-> Int`	the fuzzification set.
`-> FuzzySet`	the resulting fuzzy set.

Some functions used for testing:

testFuzzifierT is a simple function for testing the pure version of the fuzzifier:

testFuzzifierTPrint :: Int -> Int -> Int -> IO ()

testFuzzifierTPrint is the equivalent of the above; but returns an IO action which pretty prints the resulting fuzzy set (presuming it's indexed from 0).