Contents

1 Introduction

Dear Kevin Clancy, Heather Miller and Christopher Meiklejohn.

I very much enjoyed reading your whitepaper titled **Monotonicity Types**. While reading I more and more got the suspicion that it might be possible to implement the essence of Monotonicity Types today in Haskell, foregoing the need to build a completely separate language and compiler to annotate functions as monotonic, antitonic or neither.

This turned out to be a bit of a challenge, and I learned a lot about the dependently typed features that Haskell (provided you enable some GHC extensions) offers. I believe the end result to be usable/practical today.

In this literate haskell document, I'll explain the implementation details. As literate haskell, this org-mode document can be read both as LaTeX-PDF as well as Haskell source code.

2 Preamble

As you can see, we require quite a few extensions. Why we need them will be explained when we require them.

```
{-# LANGUAGE DataKinds  #-}
{-# LANGUAGE KindSignatures  #-}
{-# LANGUAGE TypeFamilies  #-}
{-# LANGUAGE PolyKinds  #-}
{-# LANGUAGE GADTs  #-}
```

Now for the head of the module:

```
module MonotonicityTypes where
```

```
import qualified Prelude
import Prelude hiding (id, (.))
import qualified Control.Category
import qualified Control.Arrow
```

3 Building Sfuns as types that change under composition

Let us start by defining a datatype for the different tonicity qualifiers that we want to support.

```
{-| Tonicity Qualifiers from section 4.2 of the paper.-}
data TonicityQualifier = Constant | Monotone | Antitone | Unknown | Discarded
```

But rather than using these as values, we will use them at the type level, restricting the type of **Sfuns**:

```
newtype Sfun (t :: TonicityQualifier) input output = UnsafeMakeSfun { applySfun :: inpu
```

Besides being restricted by 't', Sfuns are normal functions. By declaring 'UnsafeMakeSfun' as 'unsafe', people are discouraged from using it (but can still do so if they really know what they are doing). We use it to tag the primitive functions that we know are monotone as such:

```
-- A few examples:
incrementSfun :: Num a => Sfun 'Monotone a a
incrementSfun = UnsafeMakeSfun (+ 1)

negateSfun :: Num a => Sfun 'Antitone a a
negateSfun = UnsafeMakeSfun (* (-1))
```

Normally, people will instead compose Sfuns together, which will result in an Sfun type that is their composition, as per figure 6 in the paper:

```
-- The composition of two Sfuns is the composition of their two functions,
-- but with the type restricted to the composition of their tonicity qualifiers.
composeSfuns :: Sfun t1 b c -> Sfun t2 a b -> Sfun (ComposeTonicity t1 t2) a c
composeSfuns (UnsafeMakeSfun f) (UnsafeMakeSfun g) = UnsafeMakeSfun (f Prelude.. g)
```

```
type family ComposeTonicity (t1 :: TonicityQualifier) (t2 :: TonicityQualifier)  :: To
  ComposeTonicity 'Monotone 'Monotone = 'Monotone
  ComposeTonicity 'Monotone 'Antitone = 'Antitone
  ComposeTonicity 'Antitone 'Antitone = 'Monotone
  ComposeTonicity 'Antitone 'Monotone = 'Antitone
  ComposeTonicity 'Constant other = other
  ComposeTonicity other 'Constant = other
  ComposeTonicity 'Discarded _ = 'Discarded
  ComposeTonicity _ 'Discarded = 'Discarded
  ComposeTonicity _ _ = 'Unknown
```

Now that we have defined composition, we can compose our sfuns as follows:

```
-- result1 :: Num a => Sfun 'Monotone a a
result1 = composeSfuns incrementSfun incrementSfun

-- result2 :: Num a => Sfun 'Antitone a a
result2 = composeSfuns negateSfun incrementSfun

-- result3 :: Num a => Sfun 'Antitone a a
result3 = composeSfuns incrementSfun negateSfun

-- result4 :: Num a => Sfun 'Monotone a a
result4 = composeSfuns negateSfun negateSfun
```

Those type signatures are inferred. If you were to try for instance to write

```
result4' :: Num a => Sfun 'Antitone a a
result4' = composeSfuns negateSfun negateSfun
```

then GHC will complain with a nice error:

```
• Couldn't match type ''Monotone' with ''Antitone'
  Expected type: Sfun 'Antitone a a
    Actual type: Sfun (ComposeTonicity 'Antitone 'Antitone) a a
• In the expression: composeSfuns negateSfun negateSfun
  In an equation for ‘result4’:
      result4 = composeSfuns negateSfun negateSfun
```

The foundation is there. However, to make this somewhat usable, we need to make it as seamless to use Sfuns: To the user of the library, the fact that these functions are 'special' should mostly be hidden!

It can be recognized that Sfuns are almost an instance of Arrow: The fact that the type might change under composition is not something that the normal Arrow typeclass can cope with.

Instead, we implement an 'Indexed Category' and an 'Indexed Arrow', which explicity captures this notion of types being able to change under composition.

```
-- | Behaves to ‘Control.Category‘ but has an extra ‘index‘ kind parameter,
-- | whose type-value might change under composition.
class IndexedCategory (ic :: index -> * -> * -> *) where
  type ComposeIndexes (index1 :: index) (index2 :: index) :: index -- ^ The resulting t
  id :: ic (index' :: index) a a
  (.) :: ic i1 b c -> ic i2 a b -> ic (ComposeIndexes i1 i2) a c

-- | Common composition function
infixr 1 >>>
(>>>) :: IndexedCategory ic => ic i1 a b -> ic i2 b c -> ic (ComposeIndexes i2 i1) a c
a >>> b = b . a

infixr 1 <<<
(<<<) :: IndexedCategory ic => ic i1 b c -> ic i2 a b -> ic (ComposeIndexes i1 i2) a c
(<<<) = flip (>>>)

class (IndexedCategory ic) => IndexedArrow ic where
  type DefaultIndex ic :: index
  arr :: (index ~ DefaultIndex ic) => (a -> b) -> ic index a b
  first :: ic index b c -> ic index (b, d) (c, d)
```

These IndexedCategory and IndexedArrow typeclasses can be seen as a more general variant of the normal Category/Arrow, since all normal categories/arrows can be lifted to their indexed variant by indexing them by a placeholder type like ‘()‘.

```
-- | Wrapper to lift normal categories to 'indexed' categories.
-- | We use the singleton kind ‘()‘ as 'index'.
```

```
newtype FreeIndexed (c :: * -> * -> *) (single :: ()) a b = FreeIndexed { getCategory
```

```
instance Control.Category.Category c => IndexedCategory (FreeIndexed c) where
  type ComposeIndexes a b = '()
  id = FreeIndexed Control.Category.id
  (FreeIndexed a) . (FreeIndexed b) = FreeIndexed (a Control.Category.. b)

instance Control.Arrow.Arrow c => IndexedArrow (FreeIndexed c) where
  type DefaultIndex (FreeIndexed c) = '()
  arr fun = FreeIndexed (Control.Arrow.arr fun)
  first (FreeIndexed cat) = FreeIndexed (Control.Arrow.first cat)
```

Time to implement the indexed category and indexed arrow instances for our Sfun datatype.

```
instance IndexedCategory Sfun where
  type ComposeIndexes i1 i2 = ComposeTonicity i1 i2
  id = UnsafeMakeSfun Prelude.id
  f . g = composeSfuns f g

instance IndexedArrow Sfun where
  type DefaultIndex Sfun = 'Unknown
  arr = UnsafeMakeSfun
  first (UnsafeMakeSfun fun) = UnsafeMakeSfun (Control.Arrow.first fun)
```

Now we're able to write the earlier compositions in a much nicer way, which feels just the same as composing normal functions:

```
-- result1' :: Num a => Sfun 'Monotone a a
result1' = incrementSfun . incrementSfun
```

4 Handling multi-parameter functions

We now can perform the composition of Sfuns. However, if we try to build an Sfun that takes more than one parameter, Haskell will interpret this in its usual, curried, sense, seeing it as an Sfun that takes one parameter, and returns a new (non-Sfun) function that takes the rest of the parameters.

This is not what we want, because we want the Sfun to be qualified by the tonicity qualifiers of all of its parameter types, not only the first.