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,
    TypeFamilies,
    AllowAmbiguousTypes,
    FlexibleContexts,
    FlexibleInstances,
    MultiParamTypeClasses,
    UndecidableInstances,
    TypeApplications
  #-}
```

Now for the head of the module:

```
module MonotonicityTypes where

import Control.Category hiding ((.), id)
import Control.Arrow
import Data.Kind (Type)
```

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 . 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
```

4 Handling multi-parameter functions