Embedded Domain-Specific Languages: Composable Interpretations with Dependencies

March 29, 2015

Abstract

1 Introduction

2 Composable Interpretations

Additional interpretations can be added naturally for deep embedding. For instance, if we also want to obtain the depth of a circuit, we can define the following interpretation easily.

```
newtype Depth = Depth {depth :: Int}

depthAlg :: CircuitF Depth -> Depth
depthAlg (Identity w)   = Depth 0
depthAlg (Fan w)        = Depth 1
depthAlg (Above x y)    = Depth (depth x + depth y)
depthAlg (Beside x y)   = Depth (depth x `max` depth y)
depthAlg (Stretch xs x) = Depth (depth x)
```

However, with shallow embedding, circuits can only have a single semantic domain. Pairing semantics up can provide two interpretations simultaneously [1], but it is still clumsy and not modular: existing code needs to be revised every time a new interpretations is added. Moreover, for more than two interpretations, we have to either create combinations for each pair of interpretations, or use tuples which generally lack good language support [1]. To allow defining and composing multiple interpretations modularly, we first use a type class to represent the shallow embedding

```
class Circuit circuit where
  identity :: Int -> circuit
  fan      :: Int -> circuit
  above    :: circuit -> circuit -> circuit
```

```
  beside   :: circuit -> circuit -> circuit
  stretch  :: [Int] -> circuit -> circuit
```

Each interpretation corresponds to an instance of the type class for the type of that interpretation. The **newtype** wrapper is needed to allow multiple interpretations over the same underlying type. For example, we can define "width" and "depth" interpretations as follows:

```
newtype Width     = Width     {width :: Int}
newtype Depth     = Depth     {depth :: Int}

instance Circuit Width where
  identity w   = Width w
  fan w        = Width w
  above x y    = x
  beside x y   = Width (gwidth x + gwidth y)
  stretch xs x = Width (sum xs)

instance Circuit Depth where
  identity w   = Depth 0
  fan w        = Depth 1
  above x y    = Depth (gdepth x + gdepth y)
  beside x y   = Depth (depth x `max` gdepth y)
  stretch xs x = x
```

To state that the semantic domain type i represents is part of a larger collection of types, we use the following type class

```
class i :<: e where
  inter :: e -> i
```

We further define an instance for circuit composition, which gives a semantic domain of type (i1, i2) for the interpretation

```
type Compose i1 i2 = (i1, i2)

instance (Circuit i1, Circuit i2) => Circuit (Compose i1 i2) where
  identity w   = (identity w, identity w)
  fan w        = (fan w, fan w)
  above x y    = (above (inter x) (inter y), above (inter x) (inter y))
  beside x y   = (beside (inter x) (inter y), beside (inter x) (inter y))
  stretch xs x = (stretch xs (inter x), stretch xs (inter x))
```

Following is the construction of the Brent-Kung parallel prefix circuit:

```
c1 :: Circuit circuit => circuit
c1 = (fan 2 `beside` fan 2) `above`
     stretch [2,2] (fan 2) `above`
     (identity 1 `beside` fan 2 `beside` identity 1)
```

gwidth and *gdepth* are defined to recover individual interpretations:

```
gwidth :: (Width :<: e) => e -> Int
gwidth = width . inter

gdepth :: (Depth :<: e) => e -> Int
gdepth = depth . inter
```

Then we can use the above functions to get different properties of c1, depending on its type. For example, *gwidth (c1 :: Compose Width Depth)* and *gdepth (c1 :: Compose Width Depth)* will give us its width and depth, respectively.

3 Dependent interpretations

The composable interpretation for shallow embedding in the previous section works well with independent circuits. However, when an interpretation depends on other interpretations, the previous approach becomes clumsy as well. For example, whether a circuit is well-formed or not depends on the widths of its parts. A new instance for the type *(Compose WellSized width)* is defined to capture this property.

```
newtype WellSized = WellSized {wellSized :: Bool}

instance (Circuit width, Width :<: width) =>
          Circuit (Compose WellSized width) where
  identity w   = (WellSized True, identity w)
  fan w        = (WellSized True, fan w)
  above x y    = (WellSized (gwellSized x && gwellSized y
                             && (gwidth x == gwidth y)),
                  above (inter x) (inter y))
  beside x y   = (WellSized (gwellSized x && gwellSized y),
                  beside (inter x) (inter y))
  stretch xs x = (WellSized (gwellSized x && length xs == gwidth x),
                  stretch xs (inter x))

gwellSized :: (WellSized :<: e) => e -> Bool
gwellSized = wellSized . inter
```

However, with this approach, we still need to write out the interpretation for *width* together with that for *WellSized*. Instead of explicitly composing two interpretations together when one depends on the other, we want to rely solely on generic composition defined earlier, regardless of whether the two interpretations are independent or not.

3.1 Type classes with proxies

One way is to differentiate between input and output semantic domains. By expressing that the type of output domain is part of the type of input domain, we no longer need to output a pair of interpretations for the dependent case. Let's start with a multi-parameter type class for circuit construction.

```
class Circuit inn out where
  identity :: Proxy inn -> Int -> out
  fan      :: Proxy inn -> Int -> out
  above    :: inn       -> inn -> out
  beside   :: inn       -> inn -> out
  stretch  :: [Int]     -> inn -> out

data Proxy a = Proxy
```

The interpretation for *WellSized* can thus be defined in a nicer way:

```
instance (Circuit inn WellSized, Width :<: inn, WellSized :<: inn) =>
          Circuit inn WellSized where
  identity (Proxy :: Proxy inn) w = WellSized True
  fan      (Proxy :: Proxy inn) w = WellSized True
  above x y    = WellSized (gwellSized x && gwellSized y
                            && gwidth x == gwidth y)
  beside x y   = WellSized (gwellSized x && gwellSized y)
  stretch xs x = WellSized (gwellSized x && length xs == gwidth x)
```

We also need to modify the instance for circuit composition:

```
instance (Circuit inn inn1, Circuit inn inn2) =>
          Circuit inn (Compose inn1 inn2) where
  identity (Proxy :: Proxy inn) w =
           ((identity (Proxy :: Proxy inn) w) :: inn1,
            (identity (Proxy :: Proxy inn) w) :: inn2)
  fan      (Proxy :: Proxy inn) w =
           ((fan (Proxy :: Proxy inn) w)      :: inn1,
            (fan (Proxy :: Proxy inn) w)      :: inn2)
  above x y    = ((above x y)    :: inn1, (above x y)    :: inn2)
  beside x y   = ((beside x y)   :: inn1, (beside x y)   :: inn2)
  stretch xs x = ((stretch xs x) :: inn1, (stretch xs x) :: inn2)
```

The only problem is that due to the restriction of Haskell's type classes, all of the class type variables must be reachable from the free variables of each method type [2]. Therefore, we need the Proxy here for *identity* and *fan* to allow the use of class type *inn*. Also, explicit type annotations are needed for circuit composition.

3.2 Records

Another way is to use records to represent circuits. This approach allows us to define dependent interpretations nicely and alleviate us from the pain of using explicit type annotations everywhere. We define the following data type with record syntax for circuit construction:

```
data Circuit inn out = Circuit {
  identity :: Int -> out,
  fan      :: Int -> out,
  above    :: inn -> inn -> out,
  beside   :: inn -> inn -> out,
  stretch  :: [Int] -> inn -> out
}
```

Each interpretation corresponds to a value of the data type. For example, to see whether a circuit is well-formed or not, we define the value *wsAlg*:

```
wsAlg :: (Width :<: inn, WellSized :<: inn) => Circuit inn WellSized
wsAlg = Circuit {
  identity = \w    -> WellSized True,
  fan      = \w    -> WellSized True,
  above    = \x y  -> WellSized (gwellSized x && gwellSized y &&
                                 gwidth x == gwidth y),
  beside   = \x y  -> WellSized (gwellSized x && gwellSized y),
  stretch  = \xs x -> WellSized (gwellSized x && length xs == gwidth x)
}
```

Circuit composition can also be defined as a value of the data type:

```
(<+>) :: (inn1 :<: inn, inn2 :<: inn) =>
         Circuit inn inn1 -> Circuit inn inn2 -> Circuit inn (Compose inn1 inn2)
(<+>) a1 a2 = Circuit {
  identity = \w    -> (identity a1 w, identity a2 w),
  fan      = \w    -> (fan a1 w, fan a2 2),
  above    = \x y  -> (above a1 (inter x) (inter y), above a2 (inter x) (inter y)),
  beside   = \x y  -> (beside a1 (inter x) (inter y), beside a2 (inter x) (inter y)),
  stretch  = \xs x -> (stretch a1 xs (inter x), stretch a2 xs (inter x))
}
```

Now we can compose multiple interpretations smoothly. For example, we can compose individual interpretations *widthAlg*, *depthAlg* and *wsAlg* together to get a new value *cAlg* for composed interpretation:

```
cAlg :: Circuit (Compose Depth (Compose Width WellSized))
                (Compose Depth (Compose Width WellSized))
cAlg = depthAlg <+> (widthAlg <+> wsAlg)
```

4 Extensibility in Both Dimensions

So far we've only talked about extensibility in one dimension:

4.1 F-Algebras

We've shown that the type class

```
class i :<: e where
  inter :: e -> i
```

works well with both composable and dependent interpretations for shallow embedding. In this section, we will show that it also allows adding and composing interpretations modularly for deep embedding. We will still use the CircuitF functor for circuit construction, while incorporating our observation functions (i.e. algebras for CircuitF) with the above type class. An algebra for CircuitF is defined as:

```
type GAlg r a = CircuitF r -> a
```

It consists of type r and a, and a function taking a CircuitF of r-values to an a-value. For example, algebras for *width* and *wellSized* can be defined as follows:

```
widthAlg :: (Width :<: r) => GAlg r Width
widthAlg (Identity w)   = Width w
widthAlg (Fan w)        = Width w
widthAlg (Above x y)    = Width (gwidth x)
widthAlg (Beside x y)   = Width (gwidth x + gwidth y)
widthAlg (Stretch xs x) = Width (sum xs)

wsAlg :: (WellSized :<: r, Width :<: r) => GAlg r WellSized
wsAlg (Identity w)   = WellSized True
wsAlg (Fan w)        = WellSized True
wsAlg (Above x y)    = WellSized (gwellSized x && gwellSized y &&
                                  gwidth x == gwidth y)
wsAlg (Beside x y)   = WellSized (gwellSized x && gwellSized y)
wsAlg (Stretch xs x) = WellSized (gwellSized x && length xs == gwidth x)
```

For composition, we introduce the operator (<+>), which takes two algebras as inputs and gives back an algebra that returns a composed type:

```
(<+>) :: (a :<: r, b :<: r) => GAlg r a -> GAlg r b -> GAlg r (Compose a b)
(<+>) a1 a2 (Identity w)   = (a1 (Identity w), a2 (Identity w))
(<+>) a1 a2 (Fan w)        = (a1 (Fan w), a2 (Fan w))
(<+>) a1 a2 (Above x y)    = (a1 (Above (inter x) (inter y)),
                              a2 (Above (inter x) (inter y)))
```

```
(<+>) a1 a2 (Beside x y)   = (a1 (Beside (inter x) (inter y)),
                              a2 (Beside (inter x) (inter y)))
(<+>) a1 a2 (Stretch xs x) = (a1 (Stretch xs (inter x)),
                              a2 (Stretch xs (inter x)))
```

4.2 Modular Reifiable Matching

5 Related Work

6 Conclusion

References

[1] Jeremy Gibbons, Nicolas Wu, *Folding Domain-Specific Languages: Deep and Shallow Embeddings.* ICFP' 14, September 1-6, 2014, Gothenburg, Sweden.

[2]