Composable Algebra with Dependencies

Bruno C. d. S. Oliveira
University of Hong Kong
bruno@cs.hku.hk

Shin-Cheng Mu
Academia Sinica
scm@iis.sinica.edu.tw

Shu-Hung You
National Taiwan University
suhorngcsie@gmail.com

Abstract

1. Introduction

Algebras can often be used for evaluation. However, it becomes difficult when we try to compose algebras together, which is necessary when the construction of one algebra depends on another. In the context of Domain Specific Languages (**DSL**), Jeremy Gibbons [**?**] proposed two approaches on F-Algebra to tackle the problems of compositionality and dependencies. In this paper, we will also use F-Algebra as the primary representation of algebras. In section , we will show that the problem can be handled using other representations of algebras as well.

2. An Overview of Existing Approaches

```
data CircuitF r =
  IdentityF Int
   | FanF Int
   | AboveF r r
   | BesideF r r
   | StretchF [Int] r
  deriving Functor

type Width = Int
type Depth = Int
type WellSized = Bool

type CircuitAlg a = CircuitF a → a

data Circuit = In (CircuitF Circuit)

fold :: CircuitAlg a → Circuit → a
fold alg (In x) = alg (fmap (fold alg) x)

widthAlg :: CircuitAlg Width
widthAlg (IdentityF w) = w
widthAlg (FanF w)      = w
widthAlg (AboveF x y) = x
widthAlg (BesideF x y) = x + y
widthAlg (StretchF xs x) = sum xs

depthAlg :: CircuitAlg Depth
depthAlg (IdentityF w) = 0
depthAlg (FanF w)      = 1
depthAlg (AboveF x y) = x + y
depthAlg (BesideF x y) = x ‘max‘ y
depthAlg (StretchF xs x) = x

identity :: Int → Circuit
identity = In ∘ IdentityF

fan :: Int → Circuit
fan = In ∘ FanF

above :: Circuit → Circuit → Circuit
above x y = In (AboveF x y)

beside :: Circuit → Circuit → Circuit
beside x y = In (BesideF x y)

stretch :: [Int] → Circuit → Circuit
stretch xs x = In (StretchF xs x)

circuit1 = above (beside (fan 2) (fan 2))
   (above (stretch [2, 2] (fan 2))
     (beside (identity 1) (beside (fan 2) (identity 1))))
```

2.1 Pairs for multiple interpretations with dependencies

```
wswAlg :: CircuitAlg (WellSized, Width)
wswAlg (IdentityF w) = (True, w)
wswAlg (FanF w)      = (True, w)
wswAlg (AboveF x y) = (fst x && fst y && snd x == snd y, snd x)
wswAlg (BesideF x y) = (fst x && fst y, snd x + snd y)
wswAlg (StretchF ws x) = (fst x && length ws == snd x, sum ws)

wellSized :: Circuit → WellSized
wellSized x = fst (fold wswAlg x)

width :: Circuit → Width
width x = snd (fold wswAlg x)
```

2.2 Church encoding for multiple interpretations

```
newtype Circuit1 = C1 {unC1 :: ∀a.CircuitAlg a → a}

identity1 w = C1 (λalg → alg (IdentityF w))
fan1 w      = C1 (λalg → alg (FanF w))
above1 x y  = C1 (λalg → alg (AboveF (unC1 x alg) (unC1 y alg)))
beside1 x y = C1 (λalg → alg (BesideF (unC1 x alg) (unC1 y alg)))
stretch1 ws x = C1 (λalg → alg (StretchF ws (unC1 x alg)))

width1 :: Circuit1 → Width
width1 x = unC1 x widthAlg

depth1 :: Circuit1 → Depth
depth1 x = unC1 x depthAlg
```

3. Composable Algebras

```
newtype Width2 = Width2 {width :: Int}
newtype Depth2 = Depth2 {depth :: Int}

widthAlg2 :: CircuitAlg Width2
widthAlg2 (IdentityF w) = Width2 w
widthAlg2 (FanF w)      = Width2 w
widthAlg2 (AboveF x y) = Width2 (gwidth x)
widthAlg2 (BesideF x y) = Width2 (gwidth x + gwidth y)
widthAlg2 (StretchF xs x) = Width2 (sum xs)

depthAlg2 :: CircuitAlg Depth2
depthAlg2 (IdentityF w) = Depth2 0
depthAlg2 (FanF w)      = Depth2 1
depthAlg2 (AboveF x y) = Depth2 (gdepth x + gdepth y)
depthAlg2 (BesideF x y) = Depth2 (gdepth x ‘max‘ gdepth y)
depthAlg2 (StretchF xs x) = Depth2 (gdepth x)

type Compose i1 i2 = (i1, i2)

(< + >) :: CircuitAlg a → CircuitAlg b → CircuitAlg (Compose a b)
(< + >) a1 a2 (IdentityF w) = (a1 (IdentityF w), a2 (IdentityF w))
(< + >) a1 a2 (FanF w)      = (a1 (FanF w), a2 (FanF w))
(< + >) a1 a2 (AboveF x y) = (a1 (AboveF (inter x) (inter y)),
                              a2 (AboveF (inter x) (inter y)))
(< + >) a1 a2 (BesideF x y) = (a1 (BesideF (inter x) (inter y)),
                              a2 (BesideF (inter x) (inter y)))
(< + >) a1 a2 (StretchF xs x) = (a1 (StretchF xs (inter x)),
                              a2 (StretchF xs (inter x)))

class i :<: e where
  inter :: e → i

instance i :<: i where
  inter = id

instance i :<: (Compose i i2) where
  inter = fst

instance (i :<: i2) ⇒ i :<: (Compose i1 i2) where
  inter = inter ∘ snd

gwidth :: (Width2 :<: e) ⇒ e → Int
gwidth = width ∘ inter

gdepth :: (Depth2 :<: e) ⇒ e → Int
gdepth = depth ∘ inter

cAlg = widthAlg2 < + > depthAlg2

width1 :: Circuit → Int
width1 x = gwidth (fold cAlg x)

depth1 :: Circuit → Int
depth1 x = gdepth (fold cAlg x)
```

4. Dependent Algebras

```
newtype WellSized2 = WellSized2 {wellSized :: Bool}

type GAlg r a = CircuitF r → a

widthAlg2 :: (Width2 :<: r) ⇒ GAlg r Width2
widthAlg2 (IdentityF w) = Width2 w
widthAlg2 (FanF w)      = Width2 w
widthAlg2 (AboveF x y) = Width2 (gwidth x)
widthAlg2 (BesideF x y) = Width2 (gwidth x + gwidth y)
widthAlg2 (StretchF xs x) = Width2 (sum xs)

wsAlg :: (WellSized2 :<: r, Width2 :<: r) ⇒ GAlg r WellSized2
wsAlg (IdentityF w) = WellSized2 True
wsAlg (FanF w)      = WellSized2 True
wsAlg (AboveF x y) = WellSized2 (gwellSized x && gwellSized y &&
                     gwidth x ≡ gwidth y)
wsAlg (BesideF x y) = WellSized2 (gwellSized x && gwellSized y)
wsAlg (StretchF xs x) = WellSized2 (gwellSized x &&
                     length xs ≡ gwidth x)

(< + >) :: (a :<: r, b :<: r) ⇒ GAlg r a → GAlg r b →
                              GAlg r (Compose a b)
(< + >) a1 a2 (IdentityF w) = (a1 (IdentityF w), a2 (IdentityF w))
(< + >) a1 a2 (FanF w)      = (a1 (FanF w), a2 (FanF w))
(< + >) a1 a2 (AboveF x y) = (a1 (AboveF (inter x) (inter y)),
                              a2 (AboveF (inter x) (inter y)))
(< + >) a1 a2 (BesideF x y) = (a1 (BesideF (inter x) (inter y)),
                              a2 (BesideF (inter x) (inter y)))
(< + >) a1 a2 (StretchF xs x) = (a1 (StretchF xs (inter x)),
                              a2 (StretchF xs (inter x)))

cAlg2 = widthAlg2 < + > wsAlg

width2 :: Circuit → Int
width2 x = gwidth (fold cAlg2 x)

wellSized2 :: Circuit → Bool
wellSized2 x = gwellSized (fold cAlg2 x)
```

5. Extensibility in Both Dimension

6. Other representations of algebra

6.1 Type Class with Proxies

```
data Proxy a = Proxy

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

instance (Circuit inn Width2, Width2 :<: inn) ⇒
  Circuit inn Width2 where
  identity (Proxy :: Proxy inn) w = Width2 w
  fan (Proxy :: Proxy inn) w = Width2 w
  above x y    = Width2 (gwidth x)
  beside x y   = Width2 (gwidth x + gwidth y)
  stretch xs x = Width2 (sum xs)

instance (Circuit inn WellSized2,
  Width2 :<: inn,
  WellSized2 :<: inn) ⇒ Circuit inn WellSized2 where
  identity (Proxy :: Proxy inn) w = WellSized2 True
  fan (Proxy :: Proxy inn) w = WellSized2 True
  above x y  = WellSized2 (gwellSized x && gwellSized y &&
                 gwidth x == gwidth y)
  beside x y = WellSized2 (gwellSized x && gwellSized y)
  stretch xs x = WellSized2 (gwellSized x &&
                 length xs == gwidth x)

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)

type ComposedType = Compose Width2 WellSized2

gfan w       = fan (Proxy :: Proxy ComposedType) w :: ComposedType
gidentity w = identity (Proxy :: Proxy ComposedType) w :: ComposedType
gbeside x y = (beside x y)                                 :: ComposedType
gabove x y  = (above x y)                                  :: ComposedType
gstretch xs x = (stretch xs x)                             :: ComposedType

c = (gfan 2 `gbeside` gfan 2) `gabove`
  gstretch [2, 2] (gfan 2) `gabove`
  (gidentity 1 `gbeside` gfan 2 `gbeside` gidentity 1)

width3 :: (Width2 :<: e) ⇒ e → Int
width3 = gwidth

wellSized3 :: (WellSized2 :<: e) ⇒ e → Bool
wellSized3 = gwellSized
```

6.2 Records

7. Related Work

8. Conclusion