Composable Algebra with Dependencies

Draft

Abstract

This paper presents composable algebra with dependencies: a new approach to compose algebras together to allow dependent interpretations with *fold*. For a given datatype represented by two-level-types[1], if one of its interpretations depends on a second one, we compose the algebra correponding to the first interpretation with the one corresponding to the second, and use the composed algebra with *fold* to evaluate the expression. We present our approach on top of the domain specific language for parallel prefix circuits.

1. Introduction

Algebras can often be used with *fold* to evaluate recursive expressions [2]. However, *fold* supports only compositional interpretations, meaning that an interpretation of a whole is determined solely from the interpretations of its parts. The compositionality of an interpretaion is a significant limitation to expressivity: sometimes a 'primary' interpretation of the whole depends also on 'secondary' interpretaions of its parts.

In the context of Embedded Domain Specific Languages (DSL), Jeremy Gibbons[2] proposed two approaches on F-Algebra to tackle the problems of compositionality and dependencies. We will examine the two approaches in section 4 and show that each of them has its problems.

In section 5, We will present an approach that allows us to compose algebras corresponding to different interpretations of a datatype modularly. Next, we will show how dependent interpretations can be achieved using composable algebras in section 6. We will then show that our approach can be integrated with the Modular Rifiable Matching (MRM) [4] approach to allow dependencies brought in by new datatypes.

In this paper, F-Algebra will be used as the primary representation of algebras. In section 6, we will show that the problem of dependent interpretation with *fold* can be handled using other representations of algebras as well.

Contributions In summary, the contributions of this paper are:

- **An approach to compose algebras modularly:** We introduce a type class for membership relations and how it allows us to compose algebras together.
- **Incorporating dependencies in composable algebras:** We show how dependent interpretations can be achieved on top of composable algebras.
- **Extensibility in both dimensions** We show how our algebras can be integrated with the MRM approach to resolve dependencies brought in by newly-added datatypes.
- **Dependent interpretations with type classes** We present another representation of algebras using type classes that also allows dependent interpretations.
- **Dependnt interpretations with records** We present another representation of algebras using records that also allows dependent interpretations

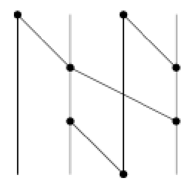


Figure 1. The Brent-Kung parallel prefix circuit of widht 4

2. DSL for parallel prefix circuits

In Jeremy Gibbons's paper[2], parallel prefix circuit is used as an example of a DSL. To make better comparison between his and our approaches, we will also work on top of the DSL for circuits.

Given an associative binary operator $\circ$, a prefix computation of width $n > 0$ takes a sequence $x_1, x_1, \ldots, xn$ of inputs and produces the sequence $x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \ldots \circ xn$ of outputs. A parallel prefix circuit performs this computation in parallel, in a fixed format independent of the input value xi.

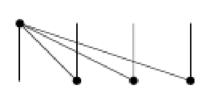
Figure 1 shows an example of a circuit. The inputs are fed in at the top, and the outputs fall out at the bottom. Each node represents a local computation, combining the values on each of its input wires using $\circ$, in left-to-right order, and providing copies of the result on each its output wires.

Such cirucits can be represented by the following algebraic datatype:

```
data Circuit =
  Identity Int
   | Fan Int
   | Above Circuit Circuit
   | Beside Circuit Circuit
   | Stretch [Int] Circuit
```

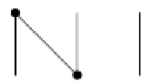
- **Identity:** *Identity n* creates a circuit consisting of n parallel wires that copy input to output. e.g. *Identity* of width 4:

- **Fan:** *Fan n* takes takes n inputs, and adds its first input to each of the others. e.g. *Fan* of width 4:



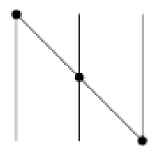
- **Beside:** *Beside x y* is the parallel or horizontal composition. It places c beside d, leaving them unconnected. There are no width constraints on c and d.

  e.g. A 2-*Fan* beside a 1-*Identity*:



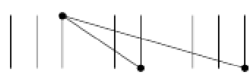
- **Above:** *Above x y* is the seires or veritical composotion. It takes two circuits c and d of the same width, and connects the outputs of c to the inputs of d.

  e.g. Place *Beside (Fan 2) (Identity 1)* above *Beside (Identity 1) (Fan 2)*. Both of the circuits are of width 3:



- **Stretch:** *Stretch ws x* takes a non-empty list of positive widths ws = [w1, ..., wn] of length n, and "stretchs" c out to width *sum ws* by interleaving some additional wires. Of the first bundle of w1 inputs, the last is routed to the first input of c and the rest pass straight through; of the next bundle of w2 inputs, the last is routed to the second input of c and the rest pass straight through; and so on.

  e.g. A 3-Fan stretched out by widths [3, 2, 3]



  On possible construction of the Brent-Kung parallel prefix circuit in Figure 1 is:

```
circuit =
  Above (Beside (Fan 2) (Fan 2))
  (Above (Stretch [2, 2] (Fan 2))
  (Beside (Identity 1) (Beside (Fan 2) (Identity 1))))
```

3. F-Algebras

Alternatively, the circuit presented above can be represented using *two-level-types*[1]. The shape of the circuit is given by functor *CircuitF* as follows:

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

CircuitF abstracts the recursive occurrences of the datatype away, using a type parameter r. We can then recover the datatype of Circuit:

```
data Circuit = In (CircuitF Circuit)
```

An algebra for CircuitF consists of a type a and a function taking a CircuitF of a-values to an a-value:

```
type CircuitAlg a = CircuitF a → a
```

Suppose we want to obtain the width of a circuit, we can pick *Width* as our evaluation target (i.e. the carrier type of *widthAlg*):

```
type Width = Int

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
```

widthAlg here will give us the final evaluation result (i.e. the width) of a circuit, assuming all children of *AboveF*, *BesideF* and *StretchF* are already evaluated and are of type *Width*.

Similarly, we can define *depthAlg* to obtain the depth of a circuit:

```
type Depth = Int

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
```

Given a nested circuit, we need a fold to traverse the recursive data structure, using algebras defined earlier for evaluation at each recursive step:

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

Each observation function for our circuit can be defined as a *fold*:

```
width :: Circuit → Width
width = fold widthAlg

depth :: Circuit → Depth
depth = fold depthAlg
```

In order to conveniently construct circuits with *CircuitF*, we define the following smart constructos:

```
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)
```

Therefore, the Brent-Kung parallel prefix circuit in Figure 1 can be constructed as:

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

It can be directly evaluated using observation functions defined earlier:

```
test1 = width circuit1
test2 = depth circuit1
```

4. Existing Approaches

To maintain the compositionality of an interpretation while bringing in dependencies, Jeremy Gibbons proposed two approaches based on F-Algebra. One example of a dependent interpretation is to see whether a circuit is well formed or not, as it depends on the widths of the circuit's constituent parts. Since the interpretation is non-compositional[2], there is no corresponding *CircuitAlg* and the circuit cannot be evaluated using *fold*.

4.1 Pairs for multiple interpretations with dependencies

To allow multiple interpretations with dependencies using *fold*, Gibbons[2] proposed the following *zygomorphism* [3], making the semantic domain of the interpretaion (i.e. the carrier type of an algebra) a pair:

```
type WellSized = Bool

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)
```

In this way, *fold wswAlg* is still a fold, and individual interpretations can be recovered as follows:

```
wellSized1 :: Circuit → WellSized
wellSized1 x = fst (fold wswAlg x)

width1 :: Circuit → Width
width1 x = snd (fold wswAlg x)
```

4.2 Church encoding for multiple interpretations

From the previous section we can see that it is possible to provide dependent interpretaions by pairing semantics up and projecting the desired interpretation from the tuple. However, it is still clumsy and not modular: existing code needs to be revised every time a new interpretation is added. Moreover, for more than two interpretations, we have to either create a combination for each pair of interpretations, or use tuples which generally lack good language support.

Therefore, Gibbons[2] presented a single parametrized interpretation, which provides a universal generic interpretation as the *Church encoding*:

```
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)))
```

It can then specialize to *width* and *depth*:

```
width2 :: Circuit1 → Width
width2 x = unC1 x widthAlg

depth2 :: Circuit1 → Depth
depth2 x = unC1 x depthAlg
```

However, one big problem with the above church encoding approach is that it still does not support dependent interpretations.

5. Composable Algebras

Our first goal is to compose algebras modularly. It will allow us to bring in dependent interpretions later. By composing two algebras together, we can get a new algebra with a carrier type containing the types of its components. In this section, we will use algebras for *width* and *depth* as examples and show how we can compose the two together.

Since a composed algebra has a composed carrier type, instead of using *Width* and *Depth* defined earlier to represent the semantic domain of each interpretation, we make use of the *newtype* wrapper to allow multiple interpretations over the same underlying type:

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

Algebras for each interpretation can be defined in the same way as before:

```
widthAlg2 :: CircuitAlg Width2
widthAlg2 (IdentityF w) = Width2 w
widthAlg2 (FanF w)      = Width2 w
widthAlg2 (AboveF x y) = Width2 (width x)
widthAlg2 (BesideF x y) = Width2 (width x + width 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 (depth x + depth y)
depthAlg2 (BesideF x y) = Depth2 (depth x ‘max‘ depth y)
depthAlg2 (StretchF xs x) = Depth2 (depth x)
```

Next we introduce the following type class to state a membership relationship between type i and e [5]:

```
class i :<: e where
  inter :: e → i
```

Here i :<: e means that type i is a component of a larger collection of types represented by e, and gives the corresponding projection functions:

```
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
```

To actually compose two algebras together, we define the operator (⊕):

```
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)))
```

(⊕) takes two algebras with carrier types a and b as inputs and gives back an algebra with a composed carrier type (Compose a b). For *AboveF*, *BesideF* and *StretchF*, their children x and y are of type e, where Width2 :<: e and WellSized2 :<: e. In the output tuple, (inter x) and (inter y) will have types corresponding to the carrier type of a1 and a2 respectively.

Now it is straightforward to compose algebras together:

```
cAlg = widthAlg2 ⊕ depthAlg2
```

cAlg is composed of widthAlg2 and depthAlg2, with a carrier type of (Compose Width2 Depth2).

We can define the evaluation function of our circuit as a *fold*:

```
eval = fold cAlg
```

To retrieve a target evaluation type from a composed type, we define *gwidth* and *gdepth*:

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

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

Individual interpretations can be defined as:

```
width3 :: Circuit → Int
width3 = gwidth ∘ eval

depth3 :: Circuit → Int
depth3 = gdepth ∘ eval
```

They can be used to evaluate the Brent-Kung parallel prefix circuit defined in section 3:

```
test1 = width3 circuit1
test2 = depth3 circuit1
```

6. Dependent Algebras

In the previous section we talked about how algebras can be composed together to allow multiple interpretations. In this section, we will introduce an approach that allows dependent interpretations. With our approach, each property we want to evaluate has a corresponding algebra. There is no need to construct a pair of interpretations when one depends on the other. For example, unlike wswAlg in section 4.1, we have wsAlg that corresponds to *wellSized*, where the definition of widthAlg is no longer needed.

The first step is to change our definition of alegebra from CircuitAlg to GAlg:

```
type GAlg r a = CircuitF r → a
```

GAlg stands for *generic algebra*. It consists of two types r and a, and a function taking CiruictF of r-vlaues to an a-value, where a :<: r. The idea is to distinguish between the uses of carrier types with respect to whether they are inputs (r) or outputs (a)[6]. For wsAlg, the first type r represents a collection of types containing both WellSized2 and Width2 (specified by (WellSized2 :<: r, Width2 :<: r)). Since each child of AboveF, BesideF and StretchF is of type r, gwidth can be used to retrieve the width of a circuit. Therefore, wsAlg can be defined as follows:

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

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)
```

Since *Width2* needs to be part of the carrier type of wsAlg such that we can retreive the width of a circuit and test if it is well-formed, we need to compose *widthAlg3* and *wsAlg* together for evaluation. While the (⊕) operator is very similar to the one defined in the previous section, we need to specify the relationships between types of algebras we are compsoing. Given an algebra from type r to type a, and another from type r to type b, where r contains both a and b, it gives back a new algebra from type r to type (Compose a b).

```
(⊕) :: (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)))

widthAlg3 :: (Width2 :<: r) ⇒ GAlg r Width2
widthAlg3 (IdentityF w) = Width2 w
widthAlg3 (FanF w)      = Width2 w
widthAlg3 (AboveF x y) = Width2 (gwidth x)
widthAlg3 (BesideF x y) = Width2 (gwidth x + gwidth y)
widthAlg3 (StretchF xs x) = Width2 (sum xs)
```

Now we can define cAlg2 that is composed of widthAlg3 and wsAlg:

```
cAlg2 = widthAlg3 ⊕ wsAlg
```

With observation functions width2 and wellSized2 defined as:

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

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

7. Extensibility in Both Dimensions

So far we have only talked about extensibility in one dimension, namely, how to add dependent observation functions in a modular way with *fold* for our DSL. What if we want to have extensibility in a second dimension, which is to extend our grammer by adding new data constructors modularly? To make the problem more interesting, these additional constructors may also bring dependencies in their corresponding observation functions at the same time. In this section, we will show that our approach of composing algebras while incorporating dependencies works well with the Modular Refiable Matching (MRM) approach[4], which allows us to add additional constructors modularly. We will present a two-level composition of algebras: for each modular component, we compose its algebras together if an interpretation is dependent; for different components, we combine their corresponding algebras together to allow evaluation of a composed data structure.

For example, say at first we only have three constructs in our DSL of circuits: *IdentityF*, *FanF*, and *BesideF*. We can define a functor *CircuitFB* to represent this datatype, where B stands for *Base*:

```
data CircuitFB r =
  IdentityF Int
```

```
  | FanF Int
  | BesideF r r
  deriving Functor
```

There is no dependencies involved for *wellSized* of this circuit, since with only *IdentityF*, *FanF* and *BesideF*, whether a circuit is well formed or not is not dependent on the width of its parts. However, we will keep our representation for dependent algebras, to be consistent with algeras we will later define for extended datatypes:

```
type GAlgB r a = CircuitFB r → a
```

Algebras for *width* and *wellSized* are exactly the same as before:

```
widthAlgB :: (Width2 :<: r) ⇒ CircuitFB r → Width2
widthAlgB (IdentityF w) = Width2 w
widthAlgB (FanF w)      = Width2 w
widthAlgB (BesideF x y) = Width2 (gwidth x + gwidth y)

wsAlgB :: (Width2 :<: r, WellSized2 :<: r) ⇒
  CircuitFB r → WellSized2
wsAlgB (IdentityF w) = WellSized2 True
wsAlgB (FanF w)      = WellSized2 True
wsAlgB (BesideF x y) =
  WellSized2 (gwellSized x && gwellSized y)
```

Now suppose we want to extend our circuits by adding new constructs *AboveF* and *StretchF*. We add the datatype constructors as a functor *CircuitFE*, where E stands for *Extended*:

```
data CircuitFE r =
  AboveF r r
   | StretchF [Int] r
  deriving Functor
```

Algebras correspond to this functor are similar to the ones above. The only difference is that the interpretation for checking if a circuit is well formed now depends on the widths of its part. Same as in section 6, we use *gwidth* to retrieve the width of a circuit:

```
type GAlgE r a = CircuitFE r → a

widthAlgE :: (Width2 :<: r) ⇒ CircuitFE r → Width2
widthAlgE (AboveF x y) = Width2 (gwidth x)
widthAlgE (StretchF xs x) = Width2 (sum xs)

wsAlgE :: (Width2 :<: r, WellSized2 :<: r) ⇒
  CircuitFE r → WellSized2
wsAlgE (AboveF x y) =
  WellSized2 (gwellSized x && gwellSized y && gwidth x == gwidth y)
wsAlgE (StretchF xs x) =
  WellSized2 (gwellSized x && length xs == gwidth x)
```

Unlike the ⊕ operator defined in previous sections, here we associate it with a type class to compose algebras correponding to different functors. With this approach, we don't have to define a different operator for algebra composition each time a new functor is added. Instead, all we have to do is to make a new instance of type class *Comb* and define the corresponding behavior of ⊕. Since we have two functors *CircuitFB* and *CircuitFE*, we create two instances of *Comb* and define ⊕ for each of them:

```
class Comb f where
  (⊕) :: (a :<: r, b :<: r) ⇒ (f r → a) → (f r → b) → (f r → (Compose a b))

instance Comb CircuitFB where
  (⊕) a1 a2 (IdentityF w) =
    (a1 (IdentityF w), a2 (IdentityF w))
  (⊕) a1 a2 (FanF w) =
    (a1 (FanF w), a2 (FanF w))
  (⊕) a1 a2 (BesideF x y) =
    (a1 (BesideF (inter x) (inter y)), a2 (BesideF (inter x) (inter y)))

instance Comb CircuitFE where
  (⊕) a1 a2 (AboveF x y)    =
    (a1 (AboveF (inter x) (inter y)), a2 (AboveF (inter x) (inter y)))
  (⊕) a1 a2 (StretchF xs x) =
    (a1 (StretchF xs (inter x)), a2 (StretchF xs (inter x)))
```

A circuit with all five constructs can be built from the modular components. First we define the type of the circuit:

```
type Circuit2 = Fix'[CircuitFB, CircuitFE]
```

The type *Circuit2* denotes circuits that have *IdentityF*, *FanF*, *BesideF*, *AboveF* and *StretchF* as their components.

Since *Width2* needs to be part of the carrier type of wsAlgE such that we can retreive the width of a circuit and test if it is well-formed, for *CircuitFE*, we need to compose *widthAlgE* and *wsAlgE* together and use *compAlgE* for evaluation.

```
compAlgE = widthAlgE ⊕ wsAlgE
```

Then we use (:::) to combine algebras correspond to different functors together [4]. Since the algebras in the list constructed by (:::) need to have the same carrier and return type, we compose *widthAlgB* and *wsAlgB* for *CircuitFB* and get *compAlgB*:

```
compAlgB = widthAlgB ⊕ wsAlgB
```

The *fold* operator defined in MRM library [4] takes an fs-algebra and Fix fs arguments. We define the evaluation function for our circuit as a fold using the combined algebras:

```
eval :: Circuit2 → Compose Width2 WellSized2
eval = fold (compAlgB ::: (compAlgE ::: Void))
```

Invidual interpretations can then be retrieved by *gwidth* and *gwellSized*:

```
width3 :: Circuit2 → Int
width3 = gwidth ∘ eval

wellSized3 :: Circuit2 → Bool
wellSized3 = gwellSized ∘ eval
```

They can be used with smart constructors to evaluate a concrete circuit:

```
circuit2 =
  (fan 2 `beside` fan 2) `above`
  stretch [2, 2] (fan 2) `above`
  (identity 1 `beside` fan 2 `beside` identity 1)

test1 = width3 circuit2
test2 = wellSized3 circuit2
```

8. Other representations of algebra

Apart from *two-level-types*, there are other ways to represent and evaluate the DSL for parallel prefix circuits. In this section, we will show two other representations, and how they can be used to allow dependent interpretations.

8.1 Type Class with Proxies

One way to represent the circuit is to use a type class. Each interpretation corresponds to an instance of the type class for the type of that interpretation. The two class type variables stand for input and output domains of an interpretation:

```
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
```

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. Therefore, we need the *Proxy* here for *identity* and *fan* to allow the use of class type *inn*:

```
data Proxy a = Proxy
```

For example, the interpretation for *width* can be defined as:

```
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)
```

On the other hand, the interpretation for *wellSized* is dependent. For member functions *above* and *stretch*, the inputs are of type *inn* which contains both *Width2* and *WellSized2*. We can retrieve the width of x and y with the help of *gwidth*:

```
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)
```

Instead of using a composition operator as before, we make another instance for interpretations with composed type:

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

Here we support interpretations for composed type by making the output of member functions a pair. The first element in the pair represents the interpretation for the first type *inn1*, while the second represents the interpretation for *inn2*.

For example, if we want to have an interpretation for type *(Compose Width2 WellSized2)*, we annotate each member function with type *ComposedType* to associate it with the instance of interpretation for composed types:

```
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
```

The Brent-Kung circuit in Figure 1 can be constructed as:

```
circuit3 =
  (gfan 2 `gbeside` gfan 2) `gabove`
  gstretch [2, 2] (gfan 2) `gabove`
  (gidentity 1 `gbeside` gfan 2 `gbeside` gidentity 1)
```

We can project individual interpretations out using *gwidth* and *gwellSized*:

```
test1 = gwidth circuit3
test3 = gwellSized circuit3
```

8.2 Records

Alternatively, circuits can be represented using records.
We define the following datatype with record syntax for circuit constructions:

```
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 datatype. For example, for *width* and *wellSized* interpretations, we define two values *widthAlg* and *wsAlg*:

```
widthAlg :: (Width2 :<: inn) ⇒ Circuit inn Width2
widthAlg = Circuit {
  identity = λw   → Width2 w,
  fan      = λw   → Width2 w,
  above    = λx y → Width2 (gwidth x),
  beside   = λx y → Width2 (gwidth x + gwidth y),
  stretch  = λxs x → Width2 (sum xs)
}

wsAlg :: (Width2 :<: inn, WellSized2 :<: inn) ⇒
  Circuit inn WellSized2
wsAlg = Circuit {
  identity = λw   → WellSized2 True,
  fan      = λw   → WellSized2 True,
  above    = λx y → WellSized2 (gwellSized x && gwellSized y &&
                    gwidth x == gwidth y),
  beside   = λx y → WellSized2 (gwellSized x && gwellSized y),
  stretch  = λxs x → WellSized2 (gwellSized x &&
                    length xs == gwidth x)
}
```

Circuit composition is also defined as a value of the datatype:

```
(⊕) :: (inn1 :<: inn, inn2 :<: inn) ⇒
  Circuit inn inn1 → Circuit inn inn2 →
  Circuit inn (Compose inn1 inn2)
(⊕) a1 a2 = Circuit {
  identity = λw  → (identity a1 w, identity a2 w),
  fan      = λw  → (fan a1 w, fan a2 2),
```

```
  above   = λx y → (above a1 (inter x) (inter y),
                      above a2 (inter x) (inter y)),
  beside  = λx y → (beside a1 (inter x) (inter y),
                      beside a2 (inter x) (inter y)),
  stretch = λxs x → (stretch a1 xs (inter x),
                      stretch a2 xs (inter x))
}
```

Now we can compose interpretations smoothly. For example, *widthAlg* and *wsAlg* can be composed together as follows:

```
cAlg :: Circuit (Compose Width2 WellSized2)
  (Compose Width2 WellSized2)
cAlg = widthAlg ⊕ wsAlg
```

Each construct is associated with the corresponding field in cAlg:

```
cidentity = identity cAlg
cfan = fan cAlg
cabove = above cAlg
cbeside = beside cAlg
cstretch = stretch cAlg
```

The Brent-Kung circuit in Figure 1 can be constructed as follows:

```
c =
  (cfan 2 `cbeside` cfan 2) `cabove`
  cstretch [2, 2] (cfan 2) `cabove`
  (cidentity 1 `cbeside` cfan 2 `cbeside` cidentity 1)
```

It can be evaluated directly using *gwidth* and *gwellSized*.

9. Related Work

Throughout this paper we have talked about the relationship with Jeremy Gibbons's work on deep and shallow embeddings for DSLs. In this section, we will discuss additional related works.

Type class for membership relations Bahr and Axelesson[5] presented a recursion scheme based on attribute grammars that can be transparently applied to trees and acyclic graphs. When discussing synthesisd attributes, they introduced a type class to express that an attribute of type c is part of a larger collection of attributes, and corresponding projection functions. However, they restricted the usage of the type class to synthesised attributes without generalizing it to algebras. Moreover, no dependencies were involved throughout their discussion of attributes.

10. Conclusion

We have presented composable algebra with dependencies. To support dependent interpretation with *fold*, we introduced a way to compose algebras together, and approaches to encode dependent relations in composed algebras. We showed that dependent interpretations can be achieved using composed algebras with *fold*. Apart from F-Algebra, we presented two other representations of algebra: type class and record, and showed how they can be used to support dependent interpretations.

References

[1] Sheard, Tim and Pasalic, Emir. Two-level types and parameterized modules. *Journal of Functional Programming*. 14(5):547-587, September 2014.

[2] Gibbons, Jeremy and Wu, Nicolas. Folding Domain-Specific Languages: Deep and Shallow Embeddings. *International Conference on Functional Programming*. 2014.

[3] Fokkinga, Maarten M. Tupling and mutumorphisms. 1990.

[4] Oliveira, B.C.d.S., Mu, Shin-Cheng and You, Shu-Hung. Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types. February 2015.

[5] Bahr, Patrick and Axelsson, Emil. Generalising Tree Traversals to DAGs. *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 27-38. 2015

[6] Oliveira, B.C.d.S., Storm, Tijs van der, Loh, Alex and Cook, William R. Feature-Oriented Programming with Object Algebra *ECOOP'13 Proceedings of the 27th European conference on Object-Oriented Programming*, pages 27-51. 2013