

Embedded Domain Specific Languages (DSL): Evaluation with Dependencies

Dr. Bruno Oliveira, Boya Peng
The University of Hong Kong

- **General Purpose Languages (GPLs) [1]:**
 - Pros: Great for generality
 - Cons: Takes a lot of effort to establish a suitable context for a particular domain
- **Domain Specific Languages (DSLs) [2]:**
 - More concise
 - Can be written more quickly
 - Easier to maintain
 - Easier to reason about

DSLs

- **Standalone DSLs:**

- Entirely separate ecosystem: compiler, editor, etc
- Reinvention of standard language features
- Common within object-oriented circle

- **Embedded DSLs:**

- Existing facilities and infrastructures of the host environment can be used
- Popular within function programming circle

- **Embedded DSLs within Functional Languages**
 - Core FP feature can be very helpful in defining embedded DSLs
 - e.g. Algebraic datatypes, higher-order functions
 - Define our DSL in Haskell

DSL for Parallel Prefix Circuits

- Prefix computation:
 - Associative binary operator: •
 - Inputs: x_1, x_2, \dots, x_n (width: $n > 0$)
 - Outputs: $x_1, x_1 \bullet x_2, \dots, x_1 \bullet x_2 \bullet \dots \bullet x_n$
- Parallel prefix circuit: performs prefix computation in parallel

DSL for Parallel Prefix Circuits

- Example:

- Input: x_1, x_2, x_3, x_4

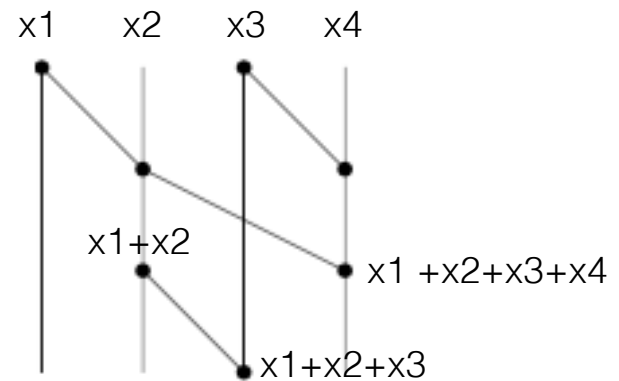


Figure 1. The Brent-Kung parallel prefix circuit of width 4

- Output: $x_1, x_1 \bullet x_2, x_1 \bullet x_2 \bullet x_3, x_1 \bullet x_2 \bullet x_3 \bullet x_4$

DSL for Parallel Prefix Circuits

- Algebraic Datatype:

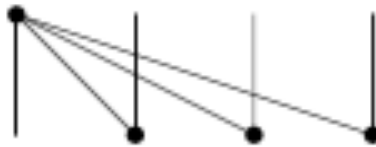
```
data Circuit =  
  Identity Int  
  | Fan Int  
  | Above Circuit Circuit  
  | Beside Circuit Circuit  
  | Stretch [Int] Circuit
```

```
| Stretch [Int] Circuit  
| Beside Circuit Circuit
```

- Identity n : e.g. Identity 4



- Fan n : e.g. Fan 4



- Beside x y : horizontal composition,

- e.g. Beside (Fan 2) (Identity 1)



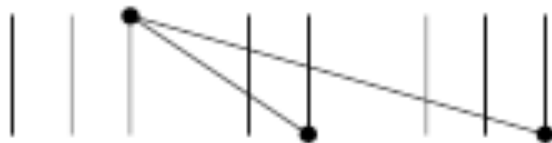
- Above x y : vertical composition, x , y are two circuits with the same width

- e.g. Above (Beside (Fan 2) (Identity 1)) (Beside (Identity 1) (Fan 2))



- Stretch ws x : ws is a list of n integers, width of $x = n$

- e.g. Stretch $[3, 2, 3]$ (Fan 3)



The Expression Problem

- “The goal is to define a datatype by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.” — Phil Wadler [3]

The Expression Problem

- In summary, we want to:

OO language: trivial
Functional language: hard

- Add new cases to a datatype modularly
- Add new evaluation functions over a datatype modularly

OO language: hard
Functional language: trivial

Modular definition of datatypes

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

```
| Stretch [Int] Circuit  
| Beside Circuit Circuit
```

```
data CircuitB =  
  Identity Int  
  | Fan Int  
  | Beside CircuitB CircuitB
```

```
data CircuitE =  
  Above CircuitE CircuitE  
  | Stretch [Int] CircuitE
```

How to add CircuitE
to CircuitB?

Modular definition of datatypes

- Modular Reifiable Matching (**MRM**) [4]
 - Use a fixpoint of list-of-functors approach to two-level types

MRM:

Adding datatypes

- Two-level types:
- Functors:

```
data CircuitFB r =  
  IdentityF Int  
  | FanF Int  
  | BesideF r r  
  deriving Functor
```

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

- Fixpoint of list-of-functors:

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

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

| Stretch [Int] Circuit

MRM —

Adding evaluation functions

- Algebras corresponding to different functors:

```
type CircuitAlgB a = CircuitFB a → a
```

```
widthAlgB :: CircuitAlgB Int
```

```
widthAlgB (IdentityF n) = n
```

```
widthAlgB (FanF n) = n
```

```
widthAlgB (BesideF x y) = x + y
```

```
type CircuitAlgE a = CircuitFE a → a
```

```
widthAlgE :: CircuitAlgE Int
```

```
widthAlgE (AboveF x y) = x
```

```
widthAlgE (StretchF ws x) = sum ws
```

- Evaluation with *fold* [4]:

```
width :: Circuit → Int
```

```
width = fold (widthAlgB ::: widthAlgE ::: Void)
```

Problem with *fold*

- The evaluation needs to be *compositional*:
 - Limitation to expressivity
- Non-compositional (Dependent) evaluation:
 - *e.g. wellSized* — Whether a circuit is well-formed or not depends on the widths of its constituent parts

```
wellSized :: CircuitE → Bool
wellSized (Above x y) =
    wellSized x && wellSized y && width x == width y
wellSized (Stretch ws x) =
    wellSized x && length ws == width x
```

```
data CircuitE =
    Above CircuitE CircuitE
  | Stretch [Int] CircuitE
```


Our work:

Dependent Evaluation with *fold*

- Idea: compose two algebras together, feed it to *fold*
 - How to compose algebras together modularly?
 - How to allow dependencies?

Composable Algebra

- Type class for membership relationship:

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

- Composition operator:

```
type Compose a b = (a, b)
```

```
(⊕) :: CircuitAlg a → CircuitAlg b → CircuitAlg (Compose a b)
```

```
(⊕) 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)))
```

Composable Algebra: Example

- Compose algebras together:

```
cAlgE :: Compose Width Depth  
cAlgE = widthAlgE2  $\oplus$  depthAlgE
```

- Retrieve individual interpretation:

```
gwidth :: (Width2 <: e)  $\Rightarrow$  e  $\rightarrow$  Int  
gwidth = width  $\circ$  inter
```

```
gdepth :: (Depth2 <: e)  $\Rightarrow$  e  $\rightarrow$  Int  
gdepth = depth  $\circ$  inter
```

- Example algebras:

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

```
widthAlgE2 :: CircuitAlgE Width  
widthAlgE2 (AboveF x y) = Width (width x)  
widthAlgE2 (StretchF ws x) = Width (sum ws)
```

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

```
depthAlgE :: CircuitAlgE Depth  
depthAlgE (AboveF x y) = Depth (depth x + depth y)  
depthAlgE (StretchF ws x) = Depth (sum ws)
```

Dependent Interpretation

- New form of algebra:

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

- Example of algebras (for CircuitFE):

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

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

```
wsAlgE :: (Width ≤: r, WellSized ≤: 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)
```

Dependencies

- Composition operator:

```
class Comb f where
```

```
  ( $\oplus$ ) :: (a <: r, b <: r)  $\Rightarrow$  (f r  $\rightarrow$  a)  $\rightarrow$  (f r  $\rightarrow$  b)  $\rightarrow$  (f r  $\rightarrow$  (Compose a b))
```

```
instance Comb CircuitFB where
```

```
  ( $\oplus$ ) a1 a2 (IdentityF w) =  
    (a1 (IdentityF w), a2 (IdentityF w))
```

```
  ( $\oplus$ ) a1 a2 (FanF w) =  
    (a1 (FanF w), a2 (FanF w))
```

```
  ( $\oplus$ ) a1 a2 (BesideF x y) =  
    (a1 (BesideF (inter x) (inter y)), a2 (BesideF (inter x) (inter y)))
```

```
instance Comb CircuitFE where
```

```
  ( $\oplus$ ) a1 a2 (AboveF x y) =  
    (a1 (AboveF (inter x) (inter y)), a2 (AboveF (inter x) (inter y)))
```

```
  ( $\oplus$ ) a1 a2 (StretchF xs x) =  
    (a1 (StretchF xs (inter x)), a2 (StretchF xs (inter x)))
```

Dependent Interpretation

- Composing algebras together:

$\text{compAlgB} = \text{widthAlgB} \oplus \text{wsAlgB}$

$\text{compAlgE} = \text{widthAlgE} \oplus \text{wsAlgE}$

- Evaluation with *fold*:

$\text{eval} :: \text{Circuit} \rightarrow \text{Compose Width WellSize}$

$\text{eval} = \text{fold} (\text{compAlgB} :: (\text{compAlgE} :: \text{Void}))$

Modularly extensible DSL with dependent interpretations:

- An approach to compose algebras modularly
- Allow dependent interpretations

Contributions

Modularly extensible DSL with dependent interpretations:

- An approach to compose algebras modularly
- Allow dependent interpretations

Thank you!

References

- [1] Jeremy Gibbons, Nicolas Wu. *Folding Domain-Specific Languages: Deep and Shallow Embeddings*
- [2] Paul Hudak. *Domain Specific Languages*
- [3] Phil Wadler. *The Expression Problem*
- [4] Bruno C.d.S. Oliveira, Shin-Cheng Mu, Shu-Hung You. *Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types*
- [5] <http://hspec.github.io/>