# Embedded Domain Specific Languages (DSL): Evaluation with Dependencies

- **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: $\bullet$
    - Inputs: $x_1, x_2, \ldots, x_n$ (width: $n > 0$)
    - Outputs: $x_1, x_1 \bullet x_2, \ldots, x_1 \bullet x_2 \bullet \cdots \bullet x_n$
- Parallel prefix circuit: performs prefix computation in parallel

# DSL for Parallel Prefix Circuits

- Example:

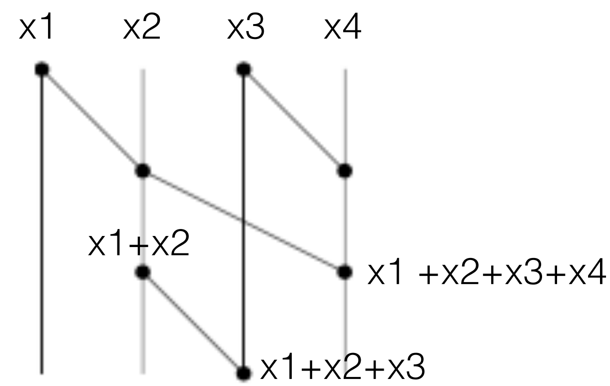



**Figure 1.** The Brent–Kung parallel prefix circuit of width 4

- Input: $x_1, x_2, x_3, x_4$
- 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
```

- Identity n: e.g. Identity 4

- Fan n: e.g. Fan 4

- 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))

# 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
```

```
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
```

# 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?

# 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)
```

# Dependent Interpretation

- Composition operator:

$$\textbf{class } \mathsf{Comb\ f}\ \textbf{where}$$
$$(\oplus) :: (\mathsf{a} :<: \mathsf{r}, \mathsf{b} :<: \mathsf{r}) \Rightarrow (\mathsf{f\ r} \to \mathsf{a}) \to (\mathsf{f\ r} \to \mathsf{b}) \to (\mathsf{f\ r} \to (\mathsf{Compose\ a\ b}))$$

- Composing algebras together:

$$\mathsf{compAlgB} = \mathsf{widthAlgB} \oplus \mathsf{wsAlgB} \qquad \mathsf{compAlgE} = \mathsf{widthAlgE} \oplus \mathsf{wsAlgE}$$

- Evaluation with *fold*:

$$\mathsf{eval} :: \mathsf{Circuit} \to \mathsf{Compose\ Width\ WellSize}$$
$$\mathsf{eval} = \mathsf{fold}\ (\mathsf{compAlgB} ::: (\mathsf{compAlgE} ::: \mathsf{Void}))$$

`

- 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/](http://hspec.github.io/)

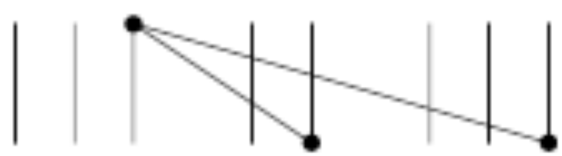
- Identity n: e.g. Identity 4

- 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)



# 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:

$$\mathsf{cAlgE} :: \mathsf{Compose\ Width\ Depth}$$
$$\mathsf{cAlgE} = \mathsf{widthAlgE2} \oplus \mathsf{depthAlgE}$$

- Retrieve individual interpretation:

$$\mathsf{gwidth} :: (\mathsf{Width2} :\prec: \mathsf{e}) \Rightarrow \mathsf{e} \to \mathsf{Int}$$
$$\mathsf{gwidth} = \mathsf{width} \circ \mathsf{inter}$$

$$\mathsf{gdepth} :: (\mathsf{Depth2} :\prec: \mathsf{e}) \Rightarrow \mathsf{e} \to \mathsf{Int}$$
$$\mathsf{gdepth} = \mathsf{depth} \circ \mathsf{inter}$$

- Example algebras:

$$\mathbf{newtype}\ \mathsf{Width} = \mathsf{Width}\ \{\mathsf{width} :: \mathsf{Int}\}$$

$$\begin{aligned}&\mathsf{widthAlgE2} :: \mathsf{CircuitAlgE\ Width}\\&\mathsf{widthAlgE2}\ (\mathsf{AboveF}\ x\ y) = \mathsf{Width}\ (\mathsf{width}\ x)\\&\mathsf{widthAlgE2}\ (\mathsf{StretchF}\ ws\ x) = \mathsf{Width}\ (\mathsf{sum}\ ws)\end{aligned}$$

$$\mathbf{newtype}\ \mathsf{Depth} = \mathsf{Depth}\ \{\mathsf{depth} :: \mathsf{Int}\}$$

$$\begin{aligned}&\mathsf{depthAlgE} :: \mathsf{CircuitAlgE\ Depth}\\&\mathsf{depthAlgE}\ (\mathsf{AboveF}\ x\ y) = \mathsf{Depth}\ (\mathsf{depth}\ x + \mathsf{depth}\ y)\\&\mathsf{depthAlgE}\ (\mathsf{StretchF}\ ws\ x) = \mathsf{Depth}\ (\mathsf{sum}\ ws)\end{aligned}$$

# Dependencies

- Composition operator:

```
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)))
```

# Dependent Interpretation

- Individual interpretations:

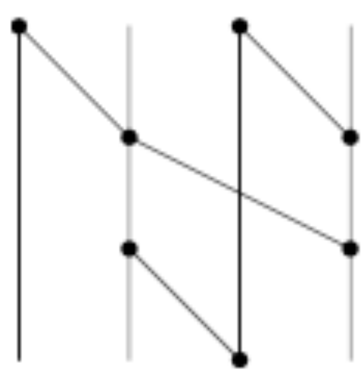
$$\begin{aligned}&\mathsf{width2} :: \mathsf{Circuit} \to \mathsf{Int}\\&\mathsf{width2} = \mathsf{gwidth} \circ \mathsf{eval}\end{aligned}$$

$$\begin{aligned}&\mathsf{wellSized2} :: \mathsf{Circuit} \to \mathsf{Bool}\\&\mathsf{wellSized2} = \mathsf{gwellSized} \circ \mathsf{eval}\end{aligned}$$

# Dependent Interpretation

- Example circuit
  (constructed with smart constructors):

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



- Tests:

```
test1 = width2 circuit1
test2 = wellSized2 circuit1
```

# Fold

- Overview:
  - *fold* Consume recursive data structures, produce a value

# Fold

- The Functor type class (member function: fmap):

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to (f\ a \to f\ b)$$

- Make CircuitF an instance of the Functor type class:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{CircuitF}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{IdentityF}\ w) &&= \mathit{IdentityF}\ w\\
&\quad \mathit{fmap}\ f\ (\mathit{FanF}\ w) &&= \mathit{FanF}\ w\\
&\quad \mathit{fmap}\ f\ (\mathit{AboveF}\ x_1\ x_2) &&= \mathit{AboveF}\ (f\ x_1)\ (f\ x_2)\\
&\quad \mathit{fmap}\ f\ (\mathit{BesideF}\ x_1\ x_2) &&= \mathit{BesideF}\ (f\ x_1)\ (f\ x_2)\\
&\quad \mathit{fmap}\ f\ (\mathit{StretchF}\ ws\ x) &&= \mathit{StretchF}\ ws\ (f\ x)
\end{aligned}$$

- Fold:

$$\mathsf{fold} :: \mathsf{CircuitAlg\ a} \to \mathsf{Circuit} \to \mathsf{a}$$
$$\mathsf{fold\ alg\ (In\ x) = alg\ (fmap\ (fold\ alg)\ x)}$$