Evaluation à la Carte
Non-Strict Evaluation via Compositional Data Types

Patrick Bahr

University of Copenhagen, Department of Computer Science
paba@diku.dk

23rd Nordic Workshop on Programming Theory,
Mälardalen University, Västerås, Sweden,
October 26 - 28, 2011
Outline

1. Compositional Data Types
2. Monadic Catamorphisms & Thunks
3. Conclusions
Outline

1. Compositional Data Types
2. Monadic Catamorphisms & Thunks
3. Conclusions
A Solution to the Expression Problem

Expression Problem [Phil Wadler]

The goal is to define a data type 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.
A Solution to the Expression Problem

Expression Problem [Phil Wadler]

*The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompling existing code, and while retaining static type safety.*

“Data Types à la Carte” by Wouter Swierstra (2008)

A solution to the expression problem: **Decoupling + Composition!**
A Solution to the Expression Problem

Expression Problem [Phil Wadler]

The goal is to define a data type 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.

“Data Types à la Carte” by Wouter Swierstra (2008)

A solution to the expression problem: Decoupling + Composition!

- data types: decoupling of signature and term construction
- functions: decoupling of pattern matching and recursion
A Solution to the Expression Problem

Expression Problem [Phil Wadler]

The goal is to define a data type 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.

“Data Types à la Carte” by Wouter Swierstra (2008)

A solution to the expression problem: Decoupling + Composition!

- data types: decoupling of signature and term construction
- functions: decoupling of pattern matching and recursion
- signatures & functions defined on them can be composed
Example: Evaluation Function

Example (A simple expression language)

```haskell
data Exp = Const Int | Pair Exp Exp | Mult Exp Exp | Fst Exp

data Value = VConst Int | VPair Value Value
```

**Example: Evaluation Function**

**Example (A simple expression language)**

```haskell
data Exp = Const Int | Pair Exp Exp | Mult Exp Exp | Fst Exp

data Value = VConst Int | VPair Value Value

eval :: Exp → Value

eval (Const n) = VConst n

eval (Pair x y) = VPair (eval x) (eval y)

eval (Mult x y) = let VConst m = eval x
                      VConst n = eval y
                      in VConst (m * n)

eval (Fst p) = let VPair x y = eval p in x
```
Decoupling Signature and Term Construction

Remove recursion from data type definition

<table>
<thead>
<tr>
<th>data</th>
<th>Exp = Const Int</th>
<th>Pair Exp Exp</th>
<th>Mult Exp Exp</th>
<th>Fst Exp</th>
</tr>
</thead>
</table>

Term f is the initial f-algebra (a.k.a. term algebra over f)
Decoupling Signature and Term Construction

Remove recursion from data type definition

\[
\text{data } \exp = \text{Const } \text{Int} \mid \text{Pair } \exp \exp \mid \text{Mult } \exp \exp \mid \text{Fst } \exp
\]

\[
\downarrow
\]

\[
\text{data } \text{sig } e = \text{Const } \text{Int} \mid \text{Pair } e e \mid \text{Mult } e e \mid \text{Fst } e
\]
Decoupling Signature and Term Construction

Remove recursion from data type definition

\[
\text{data } \text{Exp} = \text{Const Int} \mid \text{Pair Exp Exp} \mid \text{Mult Exp Exp} \mid \text{Fst Exp}
\]

\[
\Downarrow
\]

\[
\text{data } \text{Sig } e = \text{Const Int} \mid \text{Pair } e \ e \mid \text{Mult } e \ e \mid \text{Fst } e
\]

Recursion can be added separately

\[
\text{data } \text{Term } f = \text{Term } (f \ (\text{Term } f))
\]

\text{Term } f \text{ is the initial } f\text{-algebra (a.k.a. term algebra over } f)
Decoupling Signature and Term Construction

Remove recursion from data type definition

\[
\text{data } \Exp = \text{Const Int} \mid \text{Pair Exp Exp} \mid \text{Mult Exp Exp} \mid \text{Fst Exp}
\]

\[
\Downarrow
\]

\[
\text{data } \Sig e = \text{Const Int} \mid \text{Pair e e} \mid \text{Mult e e} \mid \text{Fst e}
\]

Recursion can be added separately

\[
\text{data } \Term f = \Term (f (\Term f))
\]

\(\Term f\) is the initial \(f\)-algebra (a.k.a. term algebra over \(f\))

\[
\Term \Sig \cong \Exp \quad \text{(modulo strictness)}
\]
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

**Direct sum of signatures**

\[
data (f \oplus g) \ e = \text{Inl} \ (f \ e) \mid \text{Inr} \ (g \ e)
\]

\( f \oplus g \) is the sum of the signatures \( f \) and \( g \)
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

Direct sum of signatures

\[
data (f \oplus g) e = Inl (f e) | Inr (g e)
\]

\(f \oplus g\) is the sum of the signatures \(f\) and \(g\)

Example

\[
data Sig e = Const Int
  | Pair e e
  | Mult e e
  | Fst e
\]
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

Direct sum of signatures

\[
\text{data } (f \oplus g) \ e = \text{Inl } (f \ e) \mid \text{Inr } (g \ e)
\]

\(f \oplus g\) is the sum of the signatures \(f\) and \(g\)

Example

\[
\begin{align*}
\text{data } \text{Sig } \ e &= \text{Const } \text{Int} \\
&\quad \mid \text{Pair } \ e \ e \\
&\quad \mid \text{Mult } \ e \ e \\
&\quad \mid \text{Fst } \ e \\
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Val } \ e &= \text{Const } \text{Int} \\
&\quad \mid \text{Pair } \ e \ e \\
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Op } \ e &= \text{Mult } \ e \ e \\
&\quad \mid \text{Fst } \ e
\end{align*}
\]
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

**Direct sum of signatures**

\[
\text{data } (f \oplus g) \ e = \text{Inl } (f \ e) \mid \text{Inr } (g \ e)
\]

\(f \oplus g\) is the sum of the signatures \(f\) and \(g\)

**Example**

\[
\begin{align*}
\text{data } \text{Sig } \ e &= \text{Const } \text{Int} \\
&\mid \text{Pair } \ e \ e \\
&\mid \text{Mult } \ e \ e \\
&\mid \text{Fst } \ e \\
\text{data } \text{Val } \ e &= \text{Const } \text{Int} \\
&\mid \text{Pair } \ e \ e \\
\text{data } \text{Op } \ e &= \text{Mult } \ e \ e \\
&\mid \text{Fst } \ e
\end{align*}
\]

\(\text{Val} \oplus \text{Op} \cong \text{Sig}\)
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

Direct sum of signatures

data \( (f \oplus g) \ e = \text{Inl} \ (f \ e) \ | \ \text{Inr} \ (g \ e) \)

\( f \oplus g \) is the sum of the signatures \( f \) and \( g \)

Example

\textbf{type} \ \textit{Sig} = \textit{Val} \oplus \textit{Op} \\
\textbf{data} \ \textit{Val} \ e = \text{Const Int} \\
| \ \text{Pair} \ e \ e \\
\textbf{data} \ \textit{Op} \ e = \text{Mult} \ e \ e \\
| \ \text{Fst} \ e
Combining Signatures

In order to extend expressions, we need a way to combine signatures.

Direct sum of signatures

\[
\text{data } (f \oplus g) \ e = \text{Inl } (f \ e) \mid \text{Inr } (g \ e)
\]

\(f \oplus g\) is the sum of the signatures \(f\) and \(g\)

Example

\[
\text{type } \text{Sig} = \text{Val} \oplus \text{Op}
\]

\[
\begin{align*}
\text{data Val } e &= \text{Const Int} \\
&\mid \text{Pair } e \ e \\
\text{data Op } e &= \text{Mult } e \ e \\
&\mid \text{Fst } e
\end{align*}
\]

\[
\text{Term } \text{Sig} \cong \text{Exp} \\
\text{Term } \text{Val} \cong \text{Value}
\]
Subsignatures

Subsignature type class

class \( f \prec g \) where

...
## Subsignatures

### Subsignature type class

```haskell
class f ≺ g where
    ...
```

- \( f ≺ g \) iff
  - \( g = g_1 \oplus g_2 \oplus \ldots \oplus g_n \) and
  - \( f = g_i, \quad 0 < i \leq n \)

For example:

- \( \text{Val} ≺ \text{Val} \oplus \text{Op} \)

**Injection and projection functions**

- `inject :: (g ≺ f) → \text{Term } f → \text{Term } f`
- `project :: (g ≺ f) → \text{Term } f → \text{Maybe } (g \text{ (Term } f))`
Subsignatures

Subsignature type class

Class $f \prec g$ where

... 

For example: $Val \prec Val \oplus Op$

$Sig$

$f \prec g$ iff

$g = g_1 \oplus g_2 \oplus ... \oplus g_n$ and

$f = g_i, \quad 0 < i \leq n$
Subsignatures

Subsignature type class

```haskell
class f ≺ g where
    ...

For example: Val ≺ Val ⊕ Op

\[ f ≺ g \text{ iff } \]
\[ g = g_1 ⊕ g_2 ⊕ ... ⊕ g_n \text{ and } \]
\[ f = g_i, \quad 0 < i \leq n \]
```

Injection and projection functions

```haskell
inject :: (g ≺ f) ⇒ g (Term f) → Term f
project :: (g ≺ f) ⇒ Term f → Maybe (g (Term f))
```
Separating Function Definition from Recursion

Compositional function definitions as algebras

In the same way as we defined the types:

- **define** functions on the signatures (non-recursive): \( f \ a \rightarrow a \)
- **combine** functions using type classes
- **apply** the resulting function **recursively** on the term: \( \text{Term} \ f \rightarrow a \)
Separating Function Definition from Recursion

Compositional function definitions as algebras

In the same way as we defined the types:

- **define** functions on the signatures (non-recursive): \( f \ a \rightarrow a \)
- **combine** functions using type classes
- **apply** the resulting function **recursively** on the term: \( \text{Term } f \rightarrow a \)

Algebras

```
class Eval f where
  evalAlg :: f (Term Val) → Term Val
```
Separating Function Definition from Recursion

Compositional function definitions as algebras

In the same way as we defined the types:

- define functions on the signatures (non-recursive): \( f \ a \rightarrow a \)
- combine functions using type classes
- apply the resulting function recursively on the term: \( \text{Term } f \rightarrow a \)

Algebras

```haskell
class Eval f where
  evalAlg :: f (Term Val) \rightarrow Term Val
```

Applying a function recursively to a term

```haskell
cata :: Functor f \Rightarrow (f a \rightarrow a) \rightarrow Term f \rightarrow a
```

\[
cata f (\text{Term } t) = f (\text{fmap (cata } f \text{)} t)
\]
Defining Algebras

On the singleton signatures

\begin{verbatim}
instance Eval Val where
  evalAlg = inject
\end{verbatim}
Defining Algebras

On the singleton signatures

\[ \text{instance } \text{Eval Val where} \]
\[ \text{evalAlg} = \text{inject} \]
Defining Algebras

On the singleton signatures

\textbf{instance} \textit{Eval Val where}
\[\text{evalAlg} = \text{inject}\]

\textbf{instance} \textit{Eval Op where}
\[\text{evalAlg} (\text{Mult} \times y) = \text{let} \, \text{Just} (\text{Const} \, m) = \text{project} \, x \]
\[\text{Just} (\text{Const} \, n) = \text{project} \, y \]
\[\text{in} \, \text{inject} (\text{Const} \, (m \ast n))\]
\[\text{evalAlg} (\text{Fst} \, p) = \text{let} \, \text{Just} (\text{Pair} \times y) = \text{project} \, p \]
\[\text{in} \, x\]
Defining Algebras

On the singleton signatures

```haskell
instance Eval Val where
  evalAlg = inject

instance Eval Op where
  evalAlg (Mult x y) = let Just (Const m) = project x
                      Just (Const n) = project y
                      in inject (Const (m * n))
  evalAlg (Fst p) = let Just (Pair x y) = project p
                    in x
```

Forming the catamorphism

```haskell
eval :: Term Sig → Term Val
eval = cata evalAlg
```
Outline

1. Compositional Data Types
2. Monadic Catamorphisms & Thunks
3. Conclusions
Monadic Catamorphisms

Fear the bottoms!

\[
\text{instance } \text{Eval } \text{Op where}
\]
\[
\text{evalAlg } (\text{Mult } x \times y) = \text{let } \text{Just } (\text{Const } m) = \text{project } x
\]
\[
\text{Just } (\text{Const } n) = \text{project } y
\]
\[
in \text{inject } (\text{Const } (m \times n))
\]
\[
\text{evalAlg } (\text{Fst } p) = \text{let } \text{Just } (\text{Pair } x \times y) = \text{project } p
\]
\[
in x
\]
Monadic Catamorphisms

Fear the bottoms!

\[
\begin{align*}
\text{instance } \text{Eval } \text{Op} \text{ where} \\
\text{evalAlg } (\text{Mult } x \; y) &= \text{let } \text{Just } (\text{Const } m) = \text{project } x \\
& \quad \text{Just } (\text{Const } n) = \text{project } y \\
& \quad \text{in } \text{inject } (\text{Const } (m \times n)) \\
\text{evalAlg } (\text{Fst } p) &= \text{let } \text{Just } (\text{Pair } x \; y) = \text{project } p \\
& \quad \text{in } x
\end{align*}
\]

The case distinction is incomplete
Monadic Catamorphisms

Fear the bottoms!

```haskell
instance Eval Op where
  evalAlg (Mult x y) = let Just (Const m) = project x
                        Just (Const n) = project y
                        in inject (Const (m * n))
  evalAlg (Fst p)    = let Just (Pair x y) = project p
                        in x
```

Monadic Algebra

```haskell
instance Eval Op where
  evalAlg (Mult x y) = do Const m ← project x
                          Const n ← project y
                          return (inject (Const (m * n)))
  evalAlg (Fst p)    = do Pair x y ← project p
                          return x
```
Monadic Catamorphisms

Fear the bottoms!

\[
\text{instance } \text{Eval } \text{Op where}
\]
\[
\begin{align*}
\text{evalAlg} (\text{Mult } x \times y) &= \text{let } \text{Just} (\text{Const } m) = \text{project } x \\
& \quad \text{Just} (\text{Const } n) = \text{project } y \\
& \text{in } \text{inject} (\text{Const} (m \times n)) \\
\text{evalAlg} (\text{Fst } p) &= \text{let } \text{Just} (\text{Pair } x \times y) = \text{project } p \\
& \text{in } x
\end{align*}
\]

Monadic Algebra

\[
\text{instance } \text{Eval } \text{Op where}
\]
\[
\begin{align*}
\text{evalAlg} (\text{Mult } x \times y) &= \text{do } \text{Const } m \leftarrow \text{project } x \\
& \quad \text{Const } n \leftarrow \text{project } y \\
& \quad \text{return} (\text{inject} (\text{Const} (m \times n))) \\
\text{evalAlg} (\text{Fst } p) &= \text{do } \text{Pair } x \times y \leftarrow \text{project } p \\
& \quad \text{return } x
\end{align*}
\]
Monadic Catamorphisms

Fear the bottoms!

```haskell
instance Eval Op where
  evalAlg (Mult x y) = let Just (Const m) = project x
                        Just (Const n) = project y
                        in inject (Const (m * n))
  evalAlg (Fst p) = let Just (Pair x y) = project p
                    in x
```

Monadic Algebra

```haskell
instance Eval Op where
  evalAlg (Mult x y) = do
                        Const m ← project x
                        Const n ← project y
                        return (inject (Const (m * n)))
  evalAlg (Fst p) = do
                    Pair x y ← project p
                    return x
```

both $x$ and $y$ are evaluated
Monadic Catamorphisms

Fear the bottoms!

instance Eval Op where
  evalAlg (Mult x y) = let Just (Const m) = project x
                       Just (Const n) = project y
                       in inject (Const (m * n))
  evalAlg (Fst p) = let Just (Pair x y) = project p
                    in x

Monadic Algebra

instance Eval Op where
  evalAlg (Mult x y) = do Const m ← project x
                          Const n ← project y
                          return (inject (Const (m * n)))
  evalAlg (Fst p) = do Pair x y ← project p
                      return x

both x and y are evaluated

Stricter than non-monadic evaluation!
The Type of the Monadic Evaluation Function

\[ \text{eval} :: \text{Term Sig} \rightarrow m \ (\text{Term Val}) \]
The Type of the Monadic Evaluation Function

$m \ (\text{Term Val})$
The Type of the Monadic Evaluation Function

\[ m \ (\text{Term Val}) \]
The Type of the Monadic Evaluation Function

\[ \text{Term} \left( m \oplus \text{Val} \right) \]
Creating and Evaluating Thunks

Creating a thunk

\[
\text{thunk} :: m \ (\text{Term} \ (m \oplus f)) \rightarrow \text{Term} \ (m \oplus f)\\
\text{thunk} = \text{inject}
\]
Creating and Evaluating Thunks

Creating a thunk

\[
\text{thunk} :: m \ (\text{Term} \ (m \oplus f)) \rightarrow \text{Term} \ (m \oplus f)
\]
\[
\text{thunk} = \text{inject}
\]

Evaluation to weak head normal form

\[
\text{whnf} :: \text{Monad} \ m \Rightarrow \text{Term} \ (m \oplus f) \rightarrow m \ (f \ \text{Term} \ (m \oplus f))
\]
Creating and Evaluating Thunks

Creating a thunk

\[ \text{thunk} :: m \ (\text{Term} \ (m \oplus f)) \rightarrow \text{Term} \ (m \oplus f) \]
\[ \text{thunk} = \text{inject} \]

Evaluation to \textbf{weak head normal form}

\[ \text{whnf} :: \text{Monad} \ m \Rightarrow \text{Term} \ (m \oplus f) \rightarrow m \ (f \ (\text{Term} \ (m \oplus f))) \]
\[ \text{whnf} \ (\text{Term} \ (\text{Inl} \ m)) = m \gg \text{whnf} \]
\[ \text{whnf} \ (\text{Term} \ (\text{Inr} \ t)) = \text{return} \ t \]
Evaluation via Thunks

Algebra declaration & trivial instance

class \textit{EvalT} \; f \; \textbf{where}
\begin{align*}
\textit{evalAlgT} :: f \; (\text{Term} \; (\text{Maybe} \; \oplus \; \text{Val})) \rightarrow \text{Term} \; (\text{Maybe} \; \oplus \; \text{Val})
\end{align*}
Evaluation via Thunks

Algebra declaration & trivial instance

class EvalT f where
    evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val)
### Evaluation via Thunks

**Class Declaration**

```haskell
class EvalT f where
evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val)
```

**Evaluating Operators**

```haskell
instance EvalT Val where
evalAlgT = inject
```

```haskell
instance EvalT Op where
evalAlgT (Mult x y) = thunk $ do
    Const i ← whnf x
    Const j ← whnf y
    return (inject (Const (i ∗ j)))

evalAlgT (Fst v) = thunk $ do
    Pair x y ← whnf v
    return x
```
Evaluation via Thunks

Algebra declaration & trivial instance

```haskell
class EvalT f where
evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val)

instance EvalT Val where evalAlgT = inject
```

Evaluating operators

```haskell
instance EvalT Op where
  evalAlgT (Mult x y) = thunk $ do
    Const i ← whnf x
    Const j ← whnf y
    return (inject (Const (i * j)))

  evalAlgT (Fst v) = thunk $ do
    Pair x y ← whnf v
    return x
```
Evaluation via Thunks

Algebra declaration & trivial instance

```haskell
class EvalT f where
evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val))
```

```haskell
instance EvalT Val where evalAlgT = inject
```

Evaluating operators

```haskell
instance EvalT Op where
evalAlgT (Mult x y) = thunk $ do
  Const i ← whnf x
  Const j ← whnf y
  return (inject (Const (i * j)))
```

```haskell
evalAlgT (Fst v) = thunk $ do
  Pair x y ← whnf v
  return x
```
Evaluation via Thunks

Algebra declaration & trivial instance

```
class EvalT f where
evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val))

instance EvalT Val where evalAlgT = inject
```

Evaluating operators

```
instance EvalT Op where
evalAlgT (Mult x y) = thunk $ do
    Const i ← whnf x
    Const j ← whnf y
    return (inject (Const (i ∗ j)))

evalAlgT (Fst v) = thunk $ do
    Pair x y ← whnf v
    return x
```
## Evaluation via Thunks

### Algebra declaration & trivial instance

```haskell
class EvalT f where
evalAlgT :: f (Term (Maybe ⊕ Val)) → Term (Maybe ⊕ Val))
```

```haskell
instance EvalT Val where evalAlgT = inject
```

### Evaluating operators

```haskell
instance EvalT Op where
  evalAlgT (Mult x y) = thunk $ do
    Const i ← whnf x
    Const j ← whnf y
    return (inject (Const (i ∗ j)))

  evalAlgT (Fst v) = thunk $ do
    Pair x y ← whnf v
    return x
```
Obtaining the Evaluation Function

Forming the catamorphism

\[ \text{eval}^T :: \text{Term Sig} \rightarrow \text{Term} \ (\text{Maybe} \oplus \text{Val}) \]
\[ \text{eval}^T = \text{cata evalAlg}^T \]
Obtaining the Evaluation Function

Forming the catamorphism

\[ \text{eval}T :: \text{Term} \text{ Sig} \rightarrow \text{Term} (\text{Maybe} \oplus \text{Val}) \]
\[ \text{eval}T = \text{cata evalAlg}T \]

Evaluating to normal form

\[ \text{nf} :: (\text{Monad} m, \text{Traversable} f) \Rightarrow \text{Term} (m \oplus f) \rightarrow m (\text{Term} f) \]
\[ \text{nf} = \text{liftM} \text{ Term} \cdot \text{mapM} \text{ nf} \Leftarrow \text{whnf} \]
Obtaining the Evaluation Function

Forming the catamorphism

\[
evalT :: \text{Term Sig} \rightarrow \text{Term (Maybe} \oplus \text{Val)}
\]

\[
evalT = \text{cata evalAlgT}
\]

Evaluating to normal form

\[
f :: (\text{Monad} m, \text{Traversable} f) \Rightarrow \text{Term (} m \oplus f \text{)} \rightarrow m (\text{Term} f)
\]

\[
f = \text{liftM Term . mapM nf} \Leftarrow \text{whnf}
\]

The evaluation function

\[
eval :: \text{Term Sig} \rightarrow \text{Maybe (Term Value)}
\]

\[
eval = nf \cdot evalT
\]
## Adding Strictness

Value constructors are non-strict

```
instance EvalT Val where evalAlgT = inject
```

Adding Strictness

Value constructors are non-strict

```haskell
instance EvalT Val where evalAlgT = inject
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) ⇒
        f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```
Adding Strictness

Value constructors are non-strict

```haskell
instance EvalT Val where evalAlgT = inject
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) ⇒
        f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```
Adding Strictness

Making value constructors strict

```haskell
instance EvalT Val where evalAlgT = strict
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) ⇒
       f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . wnhf)
```
Adding Strictness

Making value constructors strict

```haskell
instance EvalT Val where evalAlgT = strictAt spec
  where spec (Pair a b) = [b]
        spec _ = []
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) ⇒
  f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```
## Adding Strictness

### Making value constructors strict

```haskell
instance EvalT Val where evalAlgT = strictAt spec
  where spec (Pair a b) = [b]
    spec _     = []
```

*spec can be derived from
Haskell strictness annotations*

### Making constructors strict

```
strict :: (f ≺ g, Traversable f, Monad m) ⇒
        f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```
Adding Strictness

Making value constructors strict

\[
\text{instance } \text{EvalT } \text{Val} \text{ where } \text{evalAlgT } = \text{strictAt spec}
\]
\[
\text{where spec (Pair a b)} = [b]
\]
\[
\text{spec _ } = []
\]

Making constructors strict

\[
\text{strict :: (f \prec g, Traversable f, Monad m) } \Rightarrow
\]
\[
f \left( \text{Term (m} \oplus g) \right) \rightarrow \text{Term (m} \oplus g)
\]
\[
\text{strict } = \text{thunk . liftM inject . mapM (liftM inject . whnf)}
\]

Strictness annotations

\[
\text{data Val a } = \text{Const Int}
\]
\[
\mid \text{Pair a a}
\]
Adding Strictness

Making value constructors strict

```haskell
instance EvalT Val where evalAlgT = \spec \rightarrow strictAt spec

    where spec (Pair a b) = [b]
          spec _ = []
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) \Rightarrow
        f (Term (m ⊕ g)) \rightarrow Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```

Strictness annotations

```haskell
data Val a = Const Int
            | Pair a a
```
Adding Strictness

Making value constructors strict

```haskell
instance EvalT Val where evalAlgT = haskellStrict
```

Making constructors strict

```haskell
strict :: (f ≺ g, Traversable f, Monad m) ⇒
f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject . mapM (liftM inject . whnf)
```

Strictness annotations

```haskell
data Val a = Const Int
          | Pair a ! a
```
Outline

1. Compositional Data Types
2. Monadic Catamorphisms & Thunks
3. Conclusions
The Last Slide

What have we gained?

Monadic computations with the same strictness as pure computations!
The Last Slide

What have we gained?
Monadic computations with the same strictness as pure computations!

Other settings
- (parametric) higher-order abstract syntax
- mutually recursive data types

Easy to use
we use it ourselves for implementing DSLs
try it:
cabal install compdata
The Last Slide

What have we gained?
Monadic computations with the same strictness as pure computations!

Other settings
- (parametric) higher-order abstract syntax
- mutually recursive data types

Easy to use
- we use it ourselves for implementing DSLs

Try it: cabal install compdata
What have we gained?

Monadic computations with the same strictness as pure computations!

Other settings

- (parametric) higher-order abstract syntax
- mutually recursive data types

Easy to use

- we use it ourselves for implementing DSLs
- try it: cabal install compdata