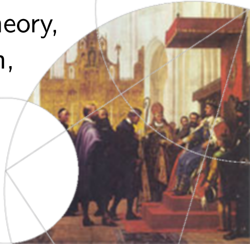
Evaluation à la Carte

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

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 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!

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)

$$\textbf{data}\ \mathit{Exp} = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Mult}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Fst}\ \mathit{Exp}$$

$$\textbf{data}\ \mathit{Value} = \mathit{VConst}\ \mathit{Int} \mid \mathit{VPair}\ \mathit{Value}\ \mathit{Value}$$

Example: Evaluation Function

Example (A simple expression language)

$$\textbf{data}\ Exp = Const\ Int \mid Pair\ Exp\ Exp \mid Mult\ Exp\ Exp \mid Fst\ Exp$$

$$\textbf{data}\ Value = VConst\ Int \mid VPair\ Value\ Value$$

$$
\begin{aligned}
& eval :: Exp \to Value \\
& eval\ (Const\ n) = VConst\ n \\
& eval\ (Pair\ x\ y) = VPair\ (eval\ x)\ (eval\ y) \\
& eval\ (Mult\ x\ y) = \textbf{let}\ VConst\ m = eval\ x \\
& \qquad\qquad\qquad\qquad\quad VConst\ n = eval\ y \\
& \qquad\qquad\qquad\quad \textbf{in}\ VConst\ (m * n) \\
& eval\ (Fst\ p) = \textbf{let}\ VPair\ x\ y = eval\ p\ \textbf{in}\ x
\end{aligned}
$$

Decoupling Signature and Term Construction

Remove recursion from data type definition

$\mathbf{data}\ \mathit{Exp} = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Mult}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Fst}\ \mathit{Exp}$

Decoupling Signature and Term Construction

Remove recursion from data type definition

data $Exp = Const\ Int \mid Pair\ Exp\ Exp \mid Mult\ Exp\ Exp \mid Fst\ Exp$

$$\Downarrow$$

data $Sig\ e = Const\ Int \mid Pair\ e\ e \mid Mult\ e\ e \mid Fst\ e$

Decoupling Signature and Term Construction

Remove recursion from data type definition

data *Exp* = *Const Int* | *Pair Exp Exp* | *Mult Exp Exp* | *Fst Exp*

⇓

data *Sig e* = *Const Int* | *Pair e e* | *Mult e e* | *Fst e*

Recursion can be added separately

data *Term f* = *Term* (*f* (*Term f*))

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

data *Exp* = *Const Int* | *Pair Exp Exp* | *Mult Exp Exp* | *Fst Exp*

$$\Downarrow$$

data *Sig e* = *Const Int* | *Pair e e* | *Mult e e* | *Fst e*

Recursion can be added separately

data *Term f* = *Term* (*f* (*Term f*))

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

Term Sig ≅ *Exp* (modulo strictness)

Combining Signatures

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

Direct sum of signatures

$\mathbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$

$f \oplus g$ 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

$\mathbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$

$f \oplus g$ is the sum of the signatures f and g

Example

$$
\begin{aligned}
\mathbf{data}\ \mathit{Sig}\ e = &\ \mathit{Const}\ \mathit{Int} \\
\mid &\ \mathit{Pair}\ e\ e \\
\mid &\ \mathit{Mult}\ e\ e \\
\mid &\ \mathit{Fst}\ e
\end{aligned}
$$

Combining Signatures

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

Direct sum of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$$

$f \oplus g$ is the sum of the signatures f and g

Example

$$\begin{aligned}\textbf{data}\ \mathit{Sig}\ e &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Pair}\ e\ e\\ &\mid \mathit{Mult}\ e\ e\\ &\mid \mathit{Fst}\ e\end{aligned} \quad \rightsquigarrow \quad \begin{aligned}\textbf{data}\ \mathit{Val}\ e &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Pair}\ e\ e\\ \textbf{data}\ \mathit{Op}\ e &= \mathit{Mult}\ e\ e\\ &\mid \mathit{Fst}\ e\end{aligned}$$

Combining Signatures

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

Direct sum of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)$$

$f \oplus g$ is the sum of the signatures f and g

Example

$$\begin{aligned}\textbf{data}\ \mathit{Sig}\ e &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Pair}\ e\ e\\ &\mid \mathit{Mult}\ e\ e\\ &\mid \mathit{Fst}\ e\end{aligned} \quad \rightsquigarrow \quad \begin{aligned}\textbf{data}\ \mathit{Val}\ e &= \mathit{Const}\ \mathit{Int}\\ &\mid \mathit{Pair}\ e\ e\\ \textbf{data}\ \mathit{Op}\ e &= \mathit{Mult}\ e\ e\\ &\mid \mathit{Fst}\ e\end{aligned}$$

$$\mathit{Val} \oplus \mathit{Op} \cong \mathit{Sig}$$

Combining Signatures

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

Direct sum of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)$$

$f \oplus g$ is the sum of the signatures f and g

Example

$$\textbf{type}\ \mathit{Sig} = \mathit{Val} \oplus \mathit{Op}$$

$$\begin{aligned} \textbf{data}\ \mathit{Val}\ e &= \mathit{Const}\ \mathit{Int} \\ &\mid \mathit{Pair}\ e\ e \\ \textbf{data}\ \mathit{Op}\ e &= \mathit{Mult}\ e\ e \\ &\mid \mathit{Fst}\ e \end{aligned}$$

Combining Signatures

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

Direct sum of signatures

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)$$

$f \oplus g$ is the sum of the signatures f and g

Example

$$\textbf{type}\ \mathit{Sig} = \mathit{Val} \oplus \mathit{Op}$$

$$\begin{aligned} \textbf{data}\ \mathit{Val}\ e &= \mathit{Const}\ \mathit{Int} \\ &\mid \mathit{Pair}\ e\ e \\ \textbf{data}\ \mathit{Op}\ e &= \mathit{Mult}\ e\ e \\ &\mid \mathit{Fst}\ e \end{aligned}$$

$$\mathit{Term}\ \mathit{Sig} \cong \mathit{Exp}$$
$$\mathit{Term}\ \mathit{Val} \cong \mathit{Value}$$

Subsignatures

Subsignature type class

class $f \prec g$ **where**

$\ldots$

Subsignatures

Subsignature type class

class $f \prec g$ **where**

...

$f \prec g$ iff

- $g = g_1 \oplus g_2 \oplus \ldots \oplus g_n$ and
- $f = g_i, \quad 0 < i \leq n$

Subsignatures

Subsignature type class

class $f \prec g$ **where**

$\cdots$

For example: $\underbrace{Val \prec Val \oplus Op}$ — $\underbrace{Val \oplus Op}_{Sig}$

$f \prec g$ iff

- $g = g_1 \oplus g_2 \oplus \ldots \oplus g_n$ and
- $f = g_i, \quad 0 < i \leq n$

Subsignatures

Subsignature type class

class $f \prec g$ **where**

…

For example: $Val \prec \underbrace{Val \oplus Op}_{Sig}$

$f \prec g$ iff

- $g = g_1 \oplus g_2 \oplus \ldots \oplus g_n$ and
- $f = g_i, \quad 0 < i \leq n$

Injection and projection functions

$inject :: (g \prec f) \Rightarrow g\ (Term\ f) \rightarrow Term\ f$

$project :: (g \prec f) \Rightarrow Term\ f \rightarrow Maybe\ (g\ (Term\ f))$

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: $Term\ f \rightarrow a$

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): $f\ a \to a$
- combine functions using type classes
- apply the resulting function recursively on the term: $Term\ f \to a$

Algebras

class *Eval f* **where**
  $evalAlg :: f\ (Term\ Val) \to Term\ Val$

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): $f\ a \to a$
- combine functions using type classes
- apply the resulting function recursively on the term: $Term\ f \to a$

Algebras

class *Eval f* **where**
  $evalAlg :: f\ (Term\ Val) \to Term\ Val$

Applying a function recursively to a term

$cata :: Functor\ f \Rightarrow (f\ a \to a) \to Term\ f \to a$
$cata\ f\ (Term\ t) = f\ (fmap\ (cata\ f)\ t)$

Defining Algebras

On the singleton signatures

instance *Eval Val* **where**
    evalAlg = *inject*

Defining Algebras

On the [obscured by callout]

$Val\ (Term\ Val) \rightarrow Term\ Val$

instance *Eval Val* **where**
    $evalAlg = inject$

Defining Algebras

On the singleton signatures

```
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
```

Defining Algebras

On the singleton signatures

```
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

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

Outline

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 Catamorphisms

Fear the bottoms!

The case distinction is incomplete

instance *Eval Op* **where**

$$evalAlg\ (Mult\ x\ y) = \mathbf{let}\ Just\ (Const\ m) = project\ x$$
$$Just\ (Const\ n) = project\ y$$
$$\mathbf{in}\ inject\ (Const\ (m * n))$$
$$evalAlg\ (Fst\ p) = \mathbf{let}\ Just\ (Pair\ x\ y) = project\ p$$
$$\mathbf{in}\ x$$

Monadic Catamorphisms

Fear the bottoms!

$$
\begin{aligned}
&\textbf{instance } \mathit{Eval}\ \mathit{Op}\ \textbf{where}\\
&\quad \mathit{evalAlg}\ (\mathit{Mult}\ x\ y) = \textbf{let } \mathit{Just}\ (\mathit{Const}\ m) = \mathit{project}\ x\\
&\quad\qquad\qquad\qquad\qquad\quad\ \ \mathit{Just}\ (\mathit{Const}\ n)\ = \mathit{project}\ y\\
&\quad\qquad\qquad\qquad\qquad\ \textbf{in } \mathit{inject}\ (\mathit{Const}\ (m * n))\\
&\quad \mathit{evalAlg}\ (\mathit{Fst}\ p)\qquad = \textbf{let } \mathit{Just}\ (\mathit{Pair}\ x\ y) = \mathit{project}\ p\\
&\quad\qquad\qquad\qquad\qquad\ \textbf{in } x
\end{aligned}
$$

Monadic Algebra

$$
\begin{aligned}
&\textbf{instance } \mathit{Eval}\ \mathit{Op}\ \textbf{where}\\
&\quad \mathit{evalAlg}\ (\mathit{Mult}\ x\ y) = \textbf{do } \mathit{Const}\ m \leftarrow \mathit{project}\ x\\
&\quad\qquad\qquad\qquad\qquad\quad\ \ \mathit{Const}\ n\ \leftarrow \mathit{project}\ y\\
&\quad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{inject}\ (\mathit{Const}\ (m * n)))\\
&\quad \mathit{evalAlg}\ (\mathit{Fst}\ p)\qquad = \textbf{do } \mathit{Pair}\ x\ y \leftarrow \mathit{project}\ p\\
&\quad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ x
\end{aligned}
$$

Monadic Catamorphisms

Fear the bottoms!

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Eval}\ \mathit{Op}\ \textbf{where}\\
&\quad \mathit{evalAlg}\ (\mathit{Mult}\ x\ y) = \textbf{let}\ \mathit{Just}\ (\mathit{Const}\ m) = \mathit{project}\ x\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (\mathit{Const}\ n)\ \ = \mathit{project}\ y\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \mathit{inject}\ (\mathit{Const}\ (m * n))\\
&\quad \mathit{evalAlg}\ (\mathit{Fst}\ p) \qquad = \textbf{let}\ \mathit{Just}\ (\mathit{Pair}\ x\ y) = \mathit{project}\ p\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ x
\end{aligned}
$$

Monadic Algebra

$\mathit{Op}\ (\mathit{Term}\ \mathit{Val}) \to \mathit{Maybe}\ (\mathit{Term}\ \mathit{Val})$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Eval}\ \mathit{Op}\ \textbf{where}\\
&\quad \mathit{evalAlg}\ (\mathit{Mult}\ x\ y) = \textbf{do}\ \mathit{Const}\ m \leftarrow \mathit{project}\ x\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Const}\ n\ \leftarrow \mathit{project}\ y\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{inject}\ (\mathit{Const}\ (m * n)))\\
&\quad \mathit{evalAlg}\ (\mathit{Fst}\ p) \qquad = \textbf{do}\ \mathit{Pair}\ x\ y \leftarrow \mathit{project}\ p\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ x
\end{aligned}
$$

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 Cons
                          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 Cons
                          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

$$eval :: Term\ Sig \rightarrow \ m \ (Term\ Val)$$

The Type of the Monadic Evaluation Function

$$m \;\; (Term \;\; Val)$$

The Type of the Monadic Evaluation Function

$$m\ (Term\ Val)$$

The Type of the Monadic Evaluation Function

$$Term \; (m \oplus Val)$$

Creating and Evaluating Thunks

Creating a thunk

$thunk :: m\ (Term\ (m \oplus f)) \rightarrow Term\ (m \oplus f)$
$thunk = inject$

Creating and Evaluating Thunks

Creating a thunk

$thunk :: m\ (Term\ (m \oplus f)) \rightarrow Term\ (m \oplus f)$
$thunk = inject$

Evaluation to **weak head normal form**

$whnf :: Monad\ m \Rightarrow Term\ (m \oplus f) \rightarrow m\ (f\ (Term\ (m \oplus f)))$

Creating and Evaluating Thunks

Creating a thunk

$$thunk :: m\,(Term\,(m \oplus f)) \rightarrow Term\,(m \oplus f)$$
$$thunk = inject$$

Evaluation to **weak head normal form**

$$whnf :: Monad\ m \Rightarrow Term\,(m \oplus f) \rightarrow m\,(f\,(Term\,(m \oplus f)))$$

$$whnf\ (Term\,(Inl\ m)) = m \gg\!= whnf$$
$$whnf\ (Term\,(Inr\ t)) = return\ t$$

Evaluation via Thunks

Algebra declaration & trivial instance

class *EvalT f* **where**

$\quad evalAlgT :: f\ (Term\ (Maybe \oplus Val)) \rightarrow Term\ (Maybe \oplus Val))$

Evaluation via Thunks

Algebra declaration & trivial instance

class *EvalT f* **where**

$evalAlgT :: f\ (Term\ (Maybe \oplus Val)) \rightarrow Term\ (Maybe \oplus Val)$

Evaluation via Thunks

$evalAlg :: f\ (Term\ Val) \rightarrow Maybe\ (Term\ Val)$

Algebra dec…

class *EvalT f* **where**

$evalAlgT :: f\ (Term\ (Maybe \oplus Val)) \rightarrow Term\ (Maybe \oplus Val))$

Evaluation via Thunks

Algebra declaration & trivial instance

class $EvalT\ f$ **where**
$evalAlgT :: f\ (Term\ (Maybe \oplus Val)) \to Term\ (Maybe \oplus Val))$

instance $EvalT\ Val$ **where** $evalAlgT = inject$

Evaluation via Thunks

Algebra declaration & trivial instance

class *EvalT f* **where**
$$evalAlgT :: f\ (Term\ (Maybe \oplus Val)) \rightarrow Term\ (Maybe \oplus Val))$$

instance *EvalT Val* **where** $evalAlgT = inject$

Evaluating operators

instance *EvalT Op* **where**

$$\begin{aligned} evalAlgT\ (Mult\ x\ y) &= thunk\ \$\ \textbf{do} \\ &\quad Const\ i \leftarrow whnf\ x \\ &\quad Const\ j \leftarrow whnf\ y \\ &\quad return\ (inject\ (Const\ (i * j))) \\ evalAlgT\ (Fst\ v) &= thunk\ \$\ \textbf{do} \\ &\quad Pair\ x\ y \leftarrow whnf\ v \\ &\quad return\ x \end{aligned}$$

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

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

Obtaining the Evaluation Function

Forming the catamorphism

$$evalT :: Term\ Sig \rightarrow Term\ (Maybe \oplus Val)$$
$$evalT = cata\ evalAlgT$$

Obtaining the Evaluation Function

Forming the catamorphism

$$evalT :: Term\ Sig \rightarrow Term\ (Maybe \oplus Val)$$
$$evalT = cata\ evalAlgT$$

Evaluating to normal form

$$nf :: (Monad\ m, Traversable\ f) \Rightarrow Term\ (m \oplus f) \rightarrow m\ (Term\ f)$$
$$nf = liftM\ Term\ .\ mapM\ nf \lll whnf$$

Obtaining the Evaluation Function

Forming the catamorphism

$$evalT :: Term\ Sig \rightarrow Term\ (Maybe \oplus Val)$$
$$evalT = cata\ evalAlgT$$

Evaluating to normal form

$$nf :: (Monad\ m, Traversable\ f) \Rightarrow Term\ (m \oplus f) \rightarrow m\ (Term\ f)$$
$$nf = liftM\ Term\ .\ mapM\ nf \lll whnf$$

The evaluation function

$$eval :: Term\ Sig \rightarrow Maybe\ (Term\ Value)$$
$$eval = nf\ .\ evalT$$

Adding Strictness

Value constructors are non-strict

instance *EvalT Val* **where** *evalAlgT* = *inject*

Adding Strictness

Value constructors are non-strict

instance $EvalT\ Val$ **where** $evalAlgT = inject$

Making constructors strict

$$strict :: (f \prec g, Traversable\ f, Monad\ m) \Rightarrow$$
$$\qquad f\ (Term\ (m \oplus g)) \to Term\ (m \oplus g)$$
$$strict = thunk\ .\ liftM\ inject\ .\ mapM\ (liftM\ inject\ .\ whnf)$$

Adding Strictness

Value constructors are non-strict

instance *EvalT Val* **where** *evalAlgT* = *inject*

Making constructors strict

$$
\begin{aligned}
&\mathit{strict} :: (f \prec g, \mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \\
&\qquad f\ (\mathit{Term}\ (m \oplus g)) \rightarrow \mathit{Term}\ (m \oplus g) \\
&\mathit{strict} = \mathit{thunk}\ .\ \mathit{liftM}\ \mathit{inject}\ .\ \mathit{mapM}\ (\mathit{liftM}\ \mathit{inject}\ .\ \mathit{whnf})
\end{aligned}
$$

Adding Strictness

Making value constructors strict

instance $EvalT\ Val$ **where** $evalAlgT = strict$

Making constructors strict

$$\begin{aligned} strict &:: (f \prec g, Traversable\ f, Monad\ m) \Rightarrow \\ &\quad f\ (Term\ (m \oplus g)) \to Term\ (m \oplus g) \\ strict &= thunk\ .\ liftM\ inject\ .\ mapM\ (liftM\ inject\ .\ whnf) \end{aligned}$$

Adding Strictness

Making value constructors strict

instance $EvalT\ Val$ **where** $evalAlgT = strictAt\ spec$

  where $spec\ (Pair\ a\ b) = [b]$

  $spec\ \_ = []$

Making constructors strict

$$strict :: (f \prec g, Traversable\ f, Monad\ m) \Rightarrow f\ (Term\ (m \oplus g)) \rightarrow Term\ (m \oplus g)$$

$$strict = thunk\ .\ liftM\ inject\ .\ mapM\ (liftM\ inject\ .\ whnf)$$

Adding Strictness

Making value constructors strict

spec can be derived from Haskell strictness annotations

$$\textbf{instance}\ \mathit{EvalT}\ \mathit{Val}\ \textbf{where}\ \mathit{evalAlgT} = \mathit{strictAt}\ \mathit{spec}$$

$$\begin{aligned} \textbf{where}\ \mathit{spec}\ (\mathit{Pair}\ a\ b) &= [b] \\ \mathit{spec}\ \_ &= [\,] \end{aligned}$$

Making constructors strict

$$\begin{aligned} &\mathit{strict} :: (f \prec g, \mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \\ &\qquad f\ (\mathit{Term}\ (m \oplus g)) \to \mathit{Term}\ (m \oplus g) \\ &\mathit{strict} = \mathit{thunk}\ .\ \mathit{liftM}\ \mathit{inject}\ .\ \mathit{mapM}\ (\mathit{liftM}\ \mathit{inject}\ .\ \mathit{whnf}) \end{aligned}$$

Adding Strictness

Making value constructors strict

spec can be derived from Haskell strictness annotations

$$\textbf{instance } \mathit{EvalT}\ \mathit{Val}\ \textbf{where } \mathit{evalAlgT} = \mathit{strictAt}\ \mathit{spec}$$

$$\begin{aligned} \textbf{where } \mathit{spec}\ (\mathit{Pair}\ a\ b) &= [b] \\ \mathit{spec}\ \_ &= [\,] \end{aligned}$$

Making constructors strict

$$\begin{aligned} &\mathit{strict} :: (f \prec g, \mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \\ &\qquad f\ (\mathit{Term}\ (m \oplus g)) \rightarrow \mathit{Term}\ (m \oplus g) \\ &\mathit{strict} = \mathit{thunk}\ .\ \mathit{liftM}\ \mathit{inject}\ .\ \mathit{mapM}\ (\mathit{liftM}\ \mathit{inject}\ .\ \mathit{whnf}) \end{aligned}$$

Strictness annotations

$$\begin{aligned} \textbf{data } \mathit{Val}\ a &= \mathit{Const}\ \mathit{Int} \\ &\mid \mathit{Pair}\ a\ \ a \end{aligned}$$

Adding Strictness

Making value constructors strict

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

$strict :: (f \prec g, Traversable\ f, Monad\ m) \Rightarrow$

  $f\ (Term\ (m \oplus g)) \rightarrow Term\ (m \oplus g)$

$strict = thunk\ .\ liftM\ inject\ .\ mapM\ (liftM\ inject\ .\ whnf)$

Strictness annotations

data $Val\ a = Const\ Int$

  $\mid Pair\ a\ !\ a$

Adding Strictness

Making value constructors strict

instance *EvalT Val* **where** $evalAlgT = haskellStrict$

Making constructors strict

$$strict :: (f \prec g, Traversable\ f, Monad\ m) \Rightarrow f\ (Term\ (m \oplus g)) \rightarrow Term\ (m \oplus g)$$
$$strict = thunk\ .\ liftM\ inject\ .\ mapM\ (liftM\ inject\ .\ whnf)$$

Strictness annotations

data *Val a* = *Const Int*
| *Pair a* ! *a*

Outline

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

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

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`