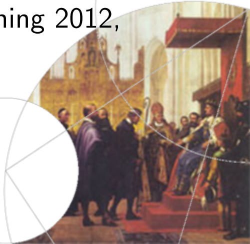
Parametric Compositional Data Types

Patrick Bahr Tom Hvitved

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

Mathematically Structured Functional Programming 2012,
Tallinn, Estonia, March 25th, 2012



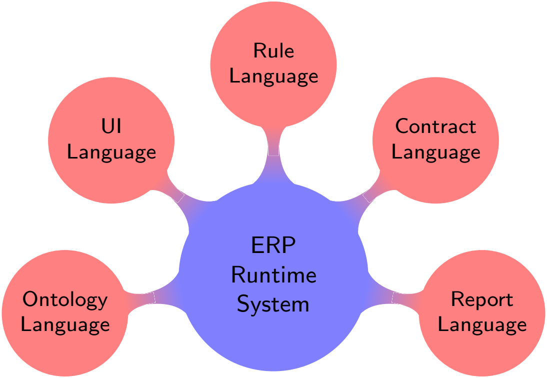
Outline

The Issue

Implementation/Prototyping of DSLs

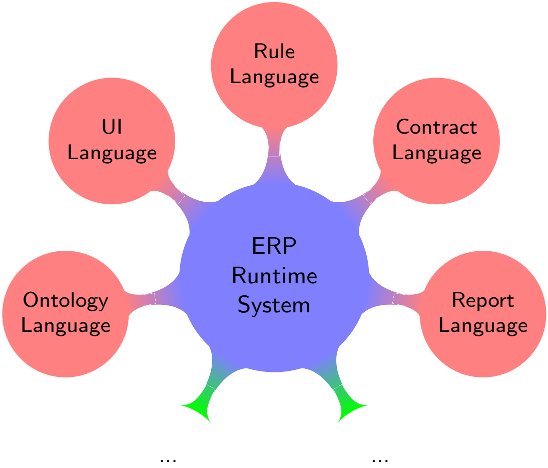
The Issue

Implementation/Prototyping of DSLs



The Issue

Implementation/Prototyping of DSLs



The Issue

Implementation/Prototyping of DSLs

The abstract picture

- We have a number of domain-specific languages.
- Each pair of DSLs shares some common sublanguage.
- All of them share a common language of values.
- We have the same situation on the type level!

The Issue

Implementation/Prototyping of DSLs

The abstract picture

- We have a number of domain-specific languages.
- Each pair of DSLs shares some common sublanguage.
- All of them share a common language of values.
- We have the same situation on the type level!

How do we implement this system without duplicating code?!

More General Application

Even with only one language to implement this issue appears!

More General Application

Even with only one language to implement this issue appears!

Different stages of a compiler work on different languages.

- Desugaring: $\mathit{FullExp} \rightarrow \mathit{CoreExp}$
- Evaluation: $\mathit{Exp} \rightarrow \mathit{Value}$

$\vdots$

More General Application

Even with only one language to implement this issue appears!

Different stages of a compiler work on different languages.

- Desugaring: $\mathit{FullExp} \to \mathit{CoreExp}$
- Evaluation: $\mathit{Exp} \to \mathit{Value}$

  $\vdots$

Manipulating/extending syntax trees

- annotating syntax trees
- adding/removing type annotations

Compositional Data Types

$$
\begin{aligned}
\textbf{data}\ \mathit{Exp} = {} & \mathit{Lit}\ \ \ \mathit{Int} \\
\mid {} & \mathit{Add}\ \ \mathit{Exp}\ \mathit{Exp} \\
\mid {} & \mathit{Mult}\ \mathit{Exp}\ \mathit{Exp}
\end{aligned}
$$

Compositional Data Types

```
data Exp = Lit  Int
         | Add  Exp Exp
         | Mult Exp Exp
```

decompose

recursion

```
data Term f =
  In (f (Term f))
```

signature

```
data Sig e = Lit  Int
           | Add  e e
           | Mult e e
```

Compositional Data Types

```
data Exp = Lit  Int
         | Add  Exp Exp
         | Mult Exp Exp
```

recursion

signature

decompose

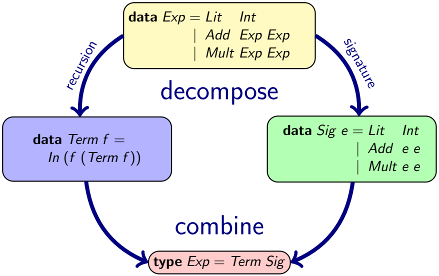
```
data Term f =
  In (f (Term f))
```

```
data Sig e = Lit  Int
           | Add  e e
           | Mult e e
```

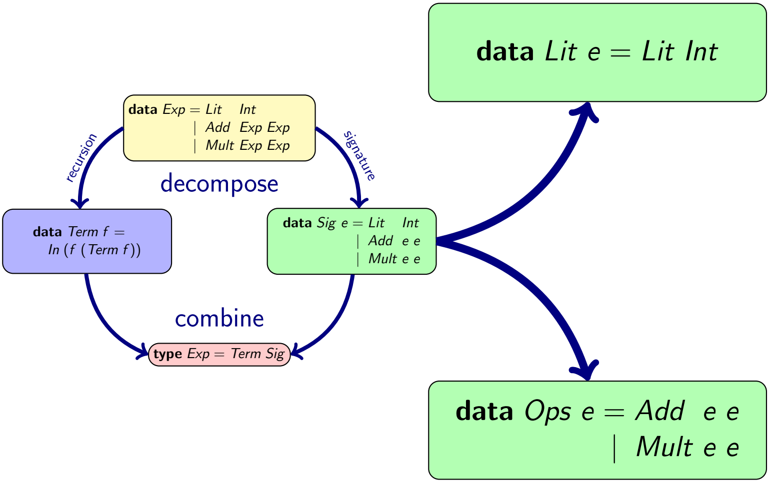
combine

```
type Exp = Term Sig
```

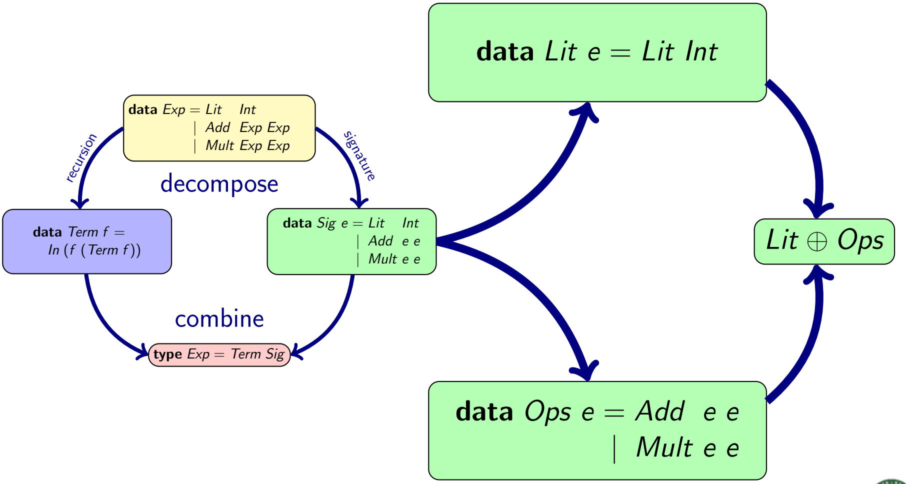
Compositional Data Types



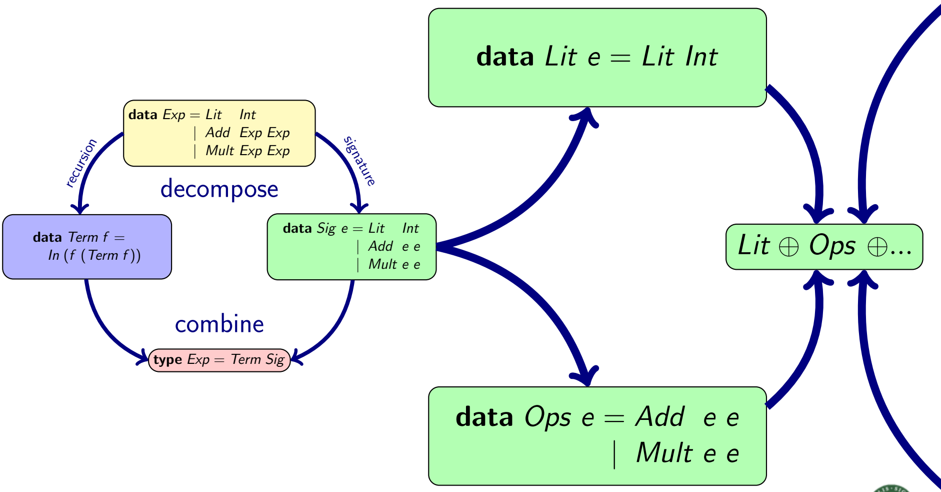
Compositional Data Types



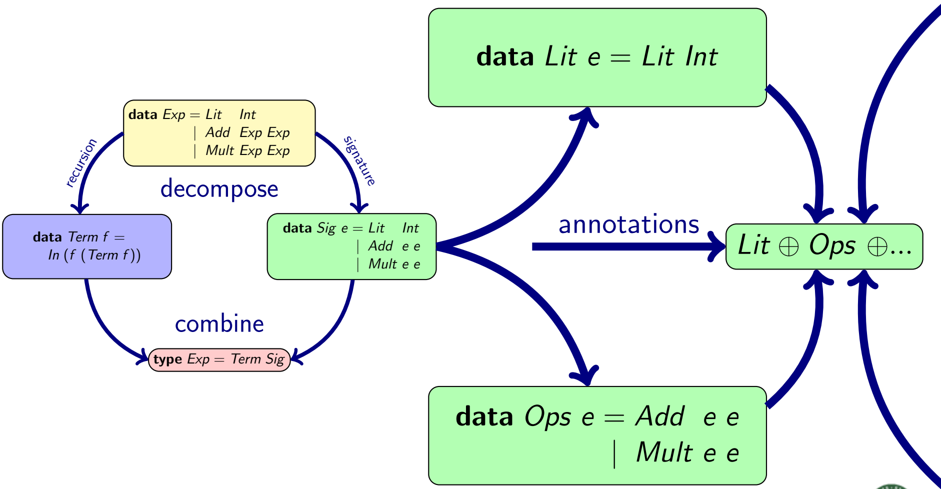
Compositional Data Types



Compositional Data Types



Compositional Data Types



Variable Binding

A straightforward solution

$$
\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String}\\
&\textbf{data}\ \mathit{Lam}\ e &&= \mathit{Lam}\ \mathit{Name}\ e\\
&\textbf{data}\ \mathit{Var}\ e &&= \mathit{Var}\ \mathit{Name}\\
&\textbf{data}\ \mathit{App}\ e &&= \mathit{App}\ e\ e
\end{aligned}
$$

Variable Binding

A straightforward solution

$$
\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String} \\
&\textbf{data}\ \mathit{Lam}\ e &&= \mathit{Lam}\ \mathit{Name}\ e \\
&\textbf{data}\ \mathit{Var}\ e &&= \mathit{Var}\ \mathit{Name} \\
&\textbf{data}\ \mathit{App}\ e &&= \mathit{App}\ e\ e
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \mathit{Sig} &&= \mathit{Lam} \oplus \mathit{Var} \oplus \mathit{App} \\
&\textbf{type}\ \mathit{Lambda} &&= \mathit{Term}\ \mathit{Lam}
\end{aligned}
$$

Variable Binding

A straightforward solution

$$\begin{array}{llcl} \textbf{type} & \mathit{Name} & = & \mathit{String} \\ \textbf{data} & \mathit{Lam}\ e & = & \mathit{Lam}\ \mathit{Name}\ e \\ \textbf{data} & \mathit{Var}\ e & = & \mathit{Var}\ \mathit{Name} \\ \textbf{data} & \mathit{App}\ e & = & \mathit{App}\ e\ e \end{array}$$

$$\begin{array}{llcl} \textbf{type} & \mathit{Sig} & = & \mathit{Lam} \oplus \mathit{Var} \oplus \mathit{App} \\ \textbf{type} & \mathit{Lambda} & = & \mathit{Term}\ \mathit{Lam} \end{array}$$

Issues

- Definitions modulo α-equivalence
- Capture-avoiding substitutions
- Implementing embedded languages

Variable Binding

A straightforward solution

$$\begin{aligned}
&\textbf{type}\ \mathit{Name} &&= \mathit{String} \\
&\textbf{data}\ \mathit{Lam}\ e &&= \mathit{Lam}\ \mathit{Name}\ e \\
&\textbf{data}\ \mathit{Var}\ e &&= \mathit{Var}\ \mathit{Name} \\
&\textbf{data}\ \mathit{App}\ e &&= \mathit{App}\ e\ e
\end{aligned}$$

$$\begin{aligned}
&\textbf{type}\ \mathit{Sig} &&= \mathit{Lam} \oplus \mathit{Var} \oplus \mathit{App} \\
&\textbf{type}\ \mathit{Lambda} &&= \mathit{Term}\ \mathit{Lam}
\end{aligned}$$

Issues

- Definitions modulo α-equivalence
- Capture-avoiding substitutions
- Implementing embedded languages

Goal

Use higher-order abstract syntax to leverage the variable binding mechanism of the host language.

Higher-Order Abstract Syntax

Explicit Variables

$$\textbf{type } \mathit{Name} = \mathit{String}$$

$$\begin{aligned}
&\textbf{data } \mathit{Lam}\ e = \mathit{Lam}\ \ \mathit{Name}\ e \\
&\textbf{data } \mathit{Var}\ e = \mathit{Var}\ \mathit{Name} \\
&\textbf{data } \mathit{App}\ e = \mathit{App}\ e\ e
\end{aligned}$$

Higher-Order Abstract Syntax

Explicit Variables

$\textbf{type}\ \mathit{Name} = \mathit{String}$

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ \mathit{Name}\ e$
$\textbf{data}\ \mathit{Var}\ e = \mathit{Var}\ \mathit{Name}$
$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

Higher-Order Abstract Syntax

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ (e \to e)$

$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

Higher-Order Abstract Syntax

Explicit Variables

$\textbf{type}\ \mathit{Name} = \mathit{String}$

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ \mathit{Name}\ e$

$\textbf{data}\ \mathit{Var}\ e = \mathit{Var}\ \mathit{Name}$

$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

Higher-Order Abstract Syntax

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ (e \to e)$

$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

Higher-Order Abstract Syntax

Explicit Variables

Lam `"x"` $(\ldots \mathit{Var}$ `"x"` $\ldots)$

$\textbf{type}\ \mathit{Name} = \mathit{String}$

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ \ \mathit{Name}\ e$

$\textbf{data}\ \mathit{Var}\ e = \mathit{Var}\ \mathit{Name}$

$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

er-Order Abst

$\mathit{Lam}\ (\lambda x \to \ldots x \ldots)$

$\textbf{data}\ \mathit{Lam}\ e = \mathit{Lam}\ \ (e \to e)$

$\textbf{data}\ \mathit{App}\ e = \mathit{App}\ e\ e$

Higher-Order Abstract Syntax

Explicit Variables

Lam "x" (…*Var* "x"…)

type *Name* $=$ *String*
data *Lam e* $=$ *Lam* *Name e*
data *Var e* $=$ *Var Name*
data *App e* $=$ *App e e*

er-Order Abst

Lam $(\lambda x \rightarrow \ldots x \ldots)$

data *Lam e* $=$ *Lam* $(e \rightarrow e)$

data *App e* $=$ *App e e*

Issues

- inefficient and cumbersome recursion schemes
  (catamorphism needs an algebra and the inverse coalgebra)
- Full function space allows for exotic terms

Higher-Order Abstract Syntax

Explicit Variables

Lam "x" (...*Var* "x"...)

type *Name* = *String*

data *Lam e* = *Lam* *Name e*

data *Var e* = *Var Name*

data *App e* = *App e e*

er-Order Abst

Lam ($\lambda x \rightarrow$ ...*x*...)

data *Lam e* = *Lam* $(e \rightarrow e)$

data *App e* = *App e e*

Issues

- inefficient and cumbersome recursion schemes (catamorphism needs an algebra and the inverse coalgebra)

  $\rightsquigarrow$ Fegaras & Sheard (1996): parametric functions space
- Full function space allows for exotic terms

Higher-Order Abstract Syntax

Explicit Variables

Lam "x" $(\ldots Var$ "x"$\ldots)$

type $Name = String$

data $Lam\ e = Lam\ Name\ e$

data $Var\ e = Var\ Name$

data $App\ e = App\ e\ e$

Higher-Order Abstract Syntax

$Lam\ (\lambda x \to \ldots x \ldots)$

data $Lam\ e = Lam\ (e \to e)$

data $App\ e = App\ e\ e$

Issues

- inefficient and cumbersome recursion schemes (catamorphism needs an algebra and the inverse coalgebra)

  $\rightsquigarrow$ Fegaras & Sheard (1996): parametric functions space

- Full function space allows for exotic terms

  $\rightsquigarrow$ Washburn & Weirich (2008): polymorphism & abstract type of terms

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  data $Lam\ a\ e = Lam\ (a \to e)$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  $\mathbf{data}\ Lam\ a\ e = Lam\ (a \rightarrow e)$

- Recursive construction of terms:

  $\mathbf{data}\ Trm\ f\ a \quad = In\ (f\ a\ (Trm\ f\ a))$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  data $Lam\ a\ e = Lam\ (a \to e)$

- Recursive construction of terms:

  data $Trm\ f\ a = In\ (f\ a\ (Trm\ f\ a))$

  type $Term\ f = \forall\ a\ .\ Trm\ f\ a$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  $$\textbf{data}\ Lam\ a\ e = Lam\ (a \to e)$$

- Recursive construction of terms:

  $$\begin{array}{llll} \textbf{data} & Trm\ f\ a & = & In\ (f\ a\ (Trm\ f\ a))\ \mid\ Var\ a \\ \textbf{type} & Term\ f & = & \forall\, a\,.\ Trm\ f\ a \end{array}$$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  data $Lam\ a\ e = Lam\ (a \to e)$

- Recursive construction of terms:

  data $Trm\ f\ a \quad = In\ (f\ a\ (Trm\ f\ a))\ \mid\ Var\ a$

  newtype $Term\ f \quad = Term\ (\forall\ a\ .\ Trm\ f\ a)$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  data $Lam\ a\ e = Lam\ (a \to e)$

- Recursive construction of terms:

  data $Trm\ f\ a = In\ (f\ a\ (Trm\ f\ a))\ \mid\ Var\ a$

  newtype $Term\ f = Term\ (\forall\ a\ .\ Trm\ f\ a)$

Example

data $Sig\ a\ e = Lam\ (a \to e)\ \mid\ App\ e\ e$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  $\textbf{data}\ \mathit{Lam}\ a\ e = \mathit{Lam}\ (a \to e)$

- Recursive construction of terms:

  $\textbf{data}\ \mathit{Trm}\ f\ a = \mathit{In}\ (f\ a\ (\mathit{Trm}\ f\ a)) \mid \mathit{Var}\ a$

  $\textbf{newtype}\ \mathit{Term}\ f = \mathit{Term}\ (\forall\ a\ .\ \mathit{Trm}\ f\ a)$

Example

$\textbf{data}\ \mathit{Sig}\ a\ e = \mathit{Lam}\ (a \to e) \mid \mathit{App}\ e\ e$

$e :: \mathit{Term}\ \mathit{Sig}$

$e = \mathit{Term}\ \$\ \mathit{Lam}\ (\lambda x \to \mathit{Var}\ x\ `\mathit{App}`\ \mathit{Var}\ x)$

Parametric Higher-Order Abstract Syntax

[Chlipala 2008]

Idea

- Signature for lambda bindings:

  data $\mathit{Lam}\ a\ e = \mathit{Lam}\ (a \to e)$

- Recursive construction of terms:

  data $\mathit{Trm}\ f\ a = \mathit{In}\ (f\ a\ (\mathit{Trm}\ f\ a)) \mid \mathit{Var}\ a$

  newtype $\mathit{Term}\ f = \mathit{Term}\ (\forall\ a\ .\ \mathit{Trm}\ f\ a)$

Example

data $\mathit{Sig}\ a\ e = \mathit{Lam}\ (a \to e) \mid \mathit{App}\ e\ e$

$e :: \mathit{Term}\ \mathit{Sig}$

$e = \mathit{Term}\ \$\ \mathit{Lam}\ (\lambda x \to \mathit{Var}\ x\ \text{`}\mathit{App}\text{`}\ \mathit{Var}\ x)$

$e = \lambda x.x\ x$

Adding Compositionality

Coproducts

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

Adding Compositionality

Coproducts

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

Example

$\textbf{data}\ \mathit{Lam}\ a\ e = \mathit{Lam}\ (a \to e)$
$\textbf{data}\ \mathit{App}\ a\ e = \mathit{App}\ e\ e$

$\textbf{type}\ \mathit{Sig} = \mathit{Lam} \oplus \mathit{App}$

Recursion Schemes

Generalising functors

class *Functor f* **where**
  $fmap :: (a \to b) \to f\ a \to f\ b$

Recursion Schemes

Generalising functors to difunctors

class *Difunctor f* **where**

$dimap :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d$

Recursion Schemes

Generalising functors to difunctors

class *Difunctor f* **where**
  $dimap :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d$

instance *Difunctor* $(\to)$ **where**
  $dimap\ f\ g\ h = g\ .\ h\ .\ f$

Recursion Schemes

Generalising functors to difunctors

class *Difunctor f* **where**
  $dimap :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d$

instance *Difunctor* $(\to)$ **where**
  $dimap\ f\ g\ h = g\ .\ h\ .\ f$

Algebras

type $Alg\ f\ c = f\ c\ c \to c$

Recursion Schemes

Generalising functors to difunctors

class *Difunctor f* **where**
  $dimap :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow f\ b\ c \rightarrow f\ a\ d$
instance *Difunctor* $(\rightarrow)$ **where**
  $dimap\ f\ g\ h = g\ .\ h\ .\ f$

Algebras

type $Alg\ f\ c = f\ c\ c \rightarrow c$

Catamorphisms

$cata :: Difunctor\ f \Rightarrow Alg\ f\ c \rightarrow Term\ f \rightarrow c$
$cata\ \phi\ (Term\ t) = cat\ t$
where $cat :: Trm\ f\ c \rightarrow c$
  $cat\ (In\ t) \quad = \phi\ (dimap\ id\ cat\ t)$
  $cat\ (Var\ x) = x$

Example

Declaring a catamorphism

class *Count f* **where**
$\phi_{\mathrm{Count}} :: \mathit{Alg}\ f\ \mathit{Int}$

$\mathit{count} :: (\mathit{Difunctor}\ f, \mathit{Count}\ f) \Rightarrow \mathit{Term}\ f \to \mathit{Int}$
$\mathit{count} = \mathit{cata}\ \phi_{\mathrm{Count}}$

Example

Declaring a catamorphism

class *Count f* **where**
$\phi_{\text{Count}} :: \mathit{Alg}\ f\ \mathit{Int}$

$\mathit{count} :: (\mathit{Difunctor}\ f, \mathit{Count}\ f) \Rightarrow \mathit{Term}\ f \rightarrow \mathit{Int}$
$\mathit{count} = \mathit{cata}\ \phi_{\text{Count}}$

Instantiation

instance *Count Lam* **where**
$\phi_{\text{Count}}\ (\mathit{Lam}\ f) = f\ 1$

instance *Count App* **where**
$\phi_{\text{Count}}\ (\mathit{App}\ e_1\ e_2) = e_1 + e_2$

Extending the Signature

Let expressions

$$\textbf{data}\ \mathit{Let}\ a\ e = \mathit{Let}\ e\ (a \to e)$$
$$\textbf{type}\ \mathit{Sig}' = \mathit{Sig} \oplus \mathit{Let}$$

Extending the Signature

Let expressions

data $\mathit{Let}\ a\ e = \mathit{Let}\ e\ (a \to e)$
type $\mathit{Sig}' = \mathit{Sig} \oplus \mathit{Let}$

Note: $\mathit{Sig} \prec \mathit{Sig}'$

Extending the Signature

Let expressions

$\mathbf{data}\ \mathit{Let}\ a\ e = \mathit{Let}\ e\ (a \to e)$
$\mathbf{type}\ \mathit{Sig}' = \mathit{Sig} \oplus \mathit{Let}$

Note: $\mathit{Sig} \prec \mathit{Sig}'$

Example

$e\ \ :: \mathit{Term}\ \mathit{Sig}$
$e = \mathit{Term}\ \$\ \mathit{iLam}\ (\lambda x \to x\ \text{`}\mathit{iApp}\text{`}\ x)$

Extending the Signature

Let expressions

$\textbf{data}\ \mathit{Let}\ a\ e = \mathit{Let}\ e\ (a \to e)$
$\textbf{type}\ \mathit{Sig}' = \mathit{Sig} \oplus \mathit{Let}$

Note: $\mathit{Sig} \prec \mathit{Sig}'$

Example

$e :: \mathit{Term}\ \mathit{Sig}'$
$e = \mathit{Term}\ \$\ \mathit{iLam}\ (\lambda x \to x\ `\mathit{iApp}`\ x)$

Extending the Signature

Let expressions

$$\textbf{data}\ Let\ a\ e = Let\ e\ (a \rightarrow e)$$
$$\textbf{type}\ Sig' = Sig \oplus Let$$

Note: $Sig \prec Sig'$

Example

$$e, e' :: Term\ Sig'$$
$$e = Term\ \$\ iLam\ (\lambda x \rightarrow x\ `iApp`\ x)$$
$$e' = Term\ \$\ iLet\ (iLam\ (\lambda x \rightarrow x\ `iApp`\ x))\ (\lambda y \rightarrow y\ `iApp`\ y)$$

Extending the Signature

Let expressions

$\textbf{data}\ \mathit{Let}\ a\ e = \mathit{Let}\ e\ (a \to e)$
$\textbf{type}\ \mathit{Sig}' = \mathit{Sig} \oplus \mathit{Let}$

Note: $\mathit{Sig} \prec \mathit{Sig}'$

Example

let $y = \lambda x.x\,x$ in $y\,y$

$e, e' :: \mathit{Term}\ \mathit{Sig}'$
$e = \mathit{Term}\ \$\ \mathit{iLam}\ (\lambda x \to x\ `\mathit{iApp}`\ x)$
$e' = \mathit{Term}\ \$\ \mathit{iLet}\ (\mathit{iLam}\ (\lambda x \to x\ `\mathit{iApp}`\ x))\ (\lambda y \to y\ `\mathit{iApp}`\ y)$

Extending the Signature

Let expressions

$$\textbf{data}\ Let\ a\ e = Let\ e\ (a \to e)$$
$$\textbf{type}\ Sig' = Sig \oplus Let$$

Note: $Sig \prec Sig'$

Example

$$e, e' :: Term\ Sig'$$
$$e = Term\ \$\ iLam\ (\lambda x \to x\ \text{`}iApp\text{`}\ x)$$
$$e' = Term\ \$\ iLet\ (iLam\ (\lambda x \to x\ \text{`}iApp\text{`}\ x))\ (\lambda y \to y\ \text{`}iApp\text{`}\ y)$$

Extending the variable counter

$$\textbf{instance}\ Count\ Let\ \textbf{where}$$
$$\quad \phi_{\mathrm{Count}}\ (Let\ e\ f) = e + f\ 1$$

But Wait, There is More!

Term transformations

- functions of type $\mathit{Term}\ f \rightarrow \mathit{Term}\ g$
- e.g. desugaring, constant folding, type inference, annotations
- efficient recursion schemes derived from tree automata

But Wait, There is More!

Term transformations

- functions of type $\mathit{Term}\ f \rightarrow \mathit{Term}\ g$
- e.g. desugaring, constant folding, type inference, annotations
- efficient recursion schemes derived from tree automata

Monadic computations

- functions of type $\mathit{Term}\ f \rightarrow m\ r$
- e.g. for pretty printing, evaluation with side effects

But Wait, There is More!

Term transformations

- functions of type $\mathit{Term}\ f \to \mathit{Term}\ g$
- e.g. desugaring, constant folding, type inference, annotations
- efficient recursion schemes derived from tree automata

Monadic computations

- functions of type $\mathit{Term}\ f \to m\ r$
- e.g. for pretty printing, evaluation with side effects

Generalised Algebraic Data Types

- difunctors $\rightsquigarrow$ indexed difunctors
- algebras $\rightsquigarrow$ many-sorted algebras

But Wait, There is More!

Term transformations

- functions of type $\mathit{Term}\ f \to \mathit{Term}\ g$
- e.g. desugaring, constant folding, type inference, annotations
- efficient recursion schemes derived from tree automata

Monadic computations

- functions of type $\mathit{Term}\ f \to m\ r$
- e.g. for pretty printing, evaluation with side effects

Generalised Algebraic Data Types

- difunctors $\rightsquigarrow$ indexed difunctors
- algebras $\rightsquigarrow$ many-sorted algebras
- mutually recursive data types (with binders)
- for simple type systems (e.g. simply typed lambda calculus)

Current Work

We use our library constantly. $\rightsquigarrow$ We extend it constantly.

Current Work

We use our library constantly. $\rightsquigarrow$ We extend it constantly.

Other extensions

- algebras with nested monadic effect
- tree homomorphisms
- tree transducers
- attribute grammars

Current Work

We use our library constantly. $\rightsquigarrow$ We extend it constantly.

Other extensions

- algebras with nested monadic effect
- tree homomorphisms
- tree transducers
- attribute grammars

Try it yourself

- `http://hackage.haskell.org/package/compdata`
- `cabal install compdata`

An example – Language Definition & Desugaring

```
data Lam a b  = Lam (a → b)
data App a b  = App b b
data Lit a b  = Lit Int
data Plus a b = Plus b b
data Let a b  = Let b (a → b)
data Err a b  = Err

$(derive [smartConstructors, makeDifunctor, makeShowD, makeEqD, makeOrdD]
         [''Lam, ''App, ''Lit, ''Plus, ''Let, ''Err])

e :: Term (Lam :+: App :+: Lit :+: Plus :+: Let :+: Err)
e = Term (iLet (iLit 2) (λx → (iLam (λy → y `iPlus` x) `iApp` iLit 3)))

-- * Desugaring
class Desug f g where
  desugHom :: Hom f g

$(derive [liftSum] [''Desug]) -- lift Desug to coproducts

desug :: (Difunctor f, Difunctor g, Desug f g) ⇒ Term f → Term g
desug (Term t) = Term (appHom desugHom t)

instance (Difunctor f, Difunctor g, f :<: g) ⇒ Desug f g where
  desugHom = In . fmap Hole . inj -- default instance for core signatures

instance (App :<: f, Lam :<: f) ⇒ Desug Let f where
  desugHom (Let e1 e2) = inject (Lam (Hole . e2)) `iApp` Hole e1
```

An example – Call-By-Value Evaluation

```
data Sem m = Fun (Sem m → m (Sem m)) | Int Int

class Monad m ⇒ Eval m f where
  evalAlg :: Alg f (m (Sem m))

$(derive [liftSum] [''Eval]) -- lift Eval to coproducts

eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata evalAlg

instance Monad m ⇒ Eval m Lam where
  evalAlg (Lam f) = return (Fun (f . return))

instance MonadError String m ⇒ Eval m App where
  evalAlg (App mx my) = do x ← mx
                           case x of Fun f → my >>= f
                                     _     → throwError "stuck"

instance Monad m ⇒ Eval m Lit where
  evalAlg (Lit n) = return (Int n)

instance MonadError String m ⇒ Eval m Plus where
  evalAlg (Plus mx my) = do x ← mx
                            y ← my
                            case (x,y) of (Int n,Int m) → return (Int (n + m))
                                          _             → throwError "stuck"

instance MonadError String m ⇒ Eval m Err where
  evalAlg Err = throwError "error"
```