Faculty of Science

Compositional Data Types

Patrick Bahr Tom Hvitved

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

7th ACM SIGPLAN
Workshop on Generic Programming,
Tokyo, Japan, September 18th, 2011

The Issue

Implementation/Prototyping of Languages

Our setting: Implementation of domain-specific languages

- We have a number of domain-specific languages.
- Each pair 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 Languages

Our setting: Implementation of domain-specific languages

- We have a number of domain-specific languages.
- Each pair 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?!

The Issue

Implementation/Prototyping of Languages

Our setting: Implementation of domain-specific languages

- We have a number of domain-specific languages.
- Each pair 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?!

Similar issues occur in general

- Evaluation: $\mathit{Exp} \to \mathit{Value}$ $\qquad \mathit{Value} \subseteq \mathit{Exp}$
- Desugaring: $\mathit{FullExp} \to \mathit{CoreExp}$ $\qquad \mathit{CoreExp} \subseteq \mathit{FullExp}$
- Annotating: $\mathit{Exp} \to \mathit{AnExp}$

  $\vdots$

Compositional Data Types

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.

Compositional Data Types

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!

Compositional Data Types

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

Compositional Data Types

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

Outline

1. Compositional Data Types
2. Extensions
3. Practical Considerations
4. Current Work

Outline

1. Compositional Data Types
2. Extensions
3. Practical Considerations
4. Current Work

Example: Evaluation Function

Example (A simple expression language)

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

Example: Evaluation Function

Example (A simple expression language)

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

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

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

Decoupling Signature and Term Construction

Remove recursion from data type definition

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

Decoupling Signature and Term Construction

Remove recursion from data type definition

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

$$\Downarrow$$

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

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

$$\Downarrow$$

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

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)

$Term\ Sig \cong Exp$ (modulo strictness)

Combining Signatures

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

Direct sum of signatures

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

$f \oplus g$ 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

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

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

Example

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

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

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

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

Combining Signatures

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

Direct sum of signatures

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

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

Example

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

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

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

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**
  $inj \;\; :: f\ a \to g\ a$
  $proj :: g\ a \to Maybe\ (f\ a)$

Subsignatures

Subsignature type class

class $f \prec g$ **where**
  $inj \;\; :: f \; a \rightarrow g \; a$
  $proj :: g \; a \rightarrow Maybe \; (f \; a)$

- $f \prec f$
- $f \prec (f \oplus g)$
- $(f \prec g) \Rightarrow f \prec (h \oplus g)$

Subsignatures

Subsignature type class

class $f \prec g$ **where**
  $inj \;\; :: f\ a \rightarrow g\ a$
  $proj :: g\ a \rightarrow Maybe\ (f\ a)$

- $f \prec f$
- $f \prec (f \oplus g)$
- $(f \prec g) \Rightarrow f \prec (h \oplus g)$

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

Subsignatures

Subsignature type class

class $f \prec g$ **where**
  $inj \;\; :: f\ a \to g\ a$
  $proj :: g\ a \to Maybe\ (f\ a)$

- $f \prec f$
- $f \prec (f \oplus g)$
- $(f \prec g) \Rightarrow f \prec (h \oplus g)$

Injection and projection functions lifted to terms

$inject :: (g \prec f) \Rightarrow g\ (Term\ f) \to Term\ f$
$inject = Term\ .\ inj$

$project :: (g \prec f) \Rightarrow Term\ f \to Maybe\ (g\ (Term\ f))$
$project\ (Term\ t) = proj\ t$

Subsignatures

Subsignature type class

class $f \prec g$ **where**
$\quad inj \;\; :: f\ a \rightarrow g\ a$
$\quad proj :: g\ a \rightarrow Maybe\ (f\ a)$

- $f \prec f$
- $f \prec (f \oplus g)$
- $(f \prec g) \Rightarrow f \prec (h \oplus g)$

Injection and projection functions lifted to terms

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

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

Smart Constructors

$Mult \quad \rightsquigarrow \quad iMult :: (Op \prec f) \Rightarrow Term\ f \rightarrow Term\ f \rightarrow Term\ f$
$iMult\ m\ n = inject\ (Mult\ m\ n)$

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: $Term\ f \to a$

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

Algebras

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

Applying a function recursively to a term

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

Defining Algebras

On the singleton signatures

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

Defining Algebras

On the 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 iConst (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**
  $evalAlg\ (Mult\ x\ y) = \textbf{let}\ Just\ (Const\ m) = project\ x$
  $\qquad\qquad\qquad\qquad\quad Just\ (Const\ n) = project\ y$
  $\qquad\qquad\qquad\quad \textbf{in}\ iConst\ (m * n)$
  $evalAlg\ (Fst\ p) = \textbf{let}\ Just\ (Pair\ x\ y) = project\ p$
  $\qquad\qquad\qquad\quad \textbf{in}\ x$

Forming the catamorphism

$eval :: Term\ Sig \rightarrow Term\ Val$
$eval = cata\ evalAlg$

Defining Algebras

On the singleton signatures

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

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

Forming the catamorphism

$eval :: (Functor\ f, Eval\ f) \Rightarrow Term\ f \rightarrow Term\ Val$
$eval = cata\ evalAlg$

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 *iConst* ($m * n$)
  evalAlg (*Fst p*) = **let** *Just* (*Pair x y*) = *project p*
            in *x*

Forming the catamorphism

eval :: (*Functor f*, *Eval f v*) ⇒ *Term f* → *Term v*
eval = *cata evalAlg*

Outline

1. Compositional Data Types
2. Extensions
3. Practical Considerations
4. Current Work

Tree Homomorphisms

Term transformations: $\mathit{Term}\ f \rightarrow \mathit{Term}\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Tree Homomorphisms

Term transformations: $Term\ f \rightarrow Term\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Tree Homomorphisms

Term transformations: $Term\ f \rightarrow Term\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Signature transformations

$$\forall\, a\, .\, f\ a \rightarrow \qquad g\ a$$

Tree Homomorphisms

Term transformations: $Term\ f \rightarrow Term\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Tree Homomorphisms

$$\forall\, a\, .\, f\ a \rightarrow Context\ g\ a$$

Tree Homomorphisms

Term transformations: $Term\ f \rightarrow Term\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Tree Homomorphisms

type $Hom\ f\ g = \forall\ a\ .\ f\ a \rightarrow Context\ g\ a$

Tree Homomorphisms

Term transformations: $Term\ f \to Term\ g$

There are a lot of formalisms for term transformations

- tree transducers
- tree homomorphisms
- term rewriting
- ...

Tree Homomorphisms

type $Hom\ f\ g = \forall\ a\ .\ f\ a \to Context\ g\ a$

Contexts (a.k.a. free monads)

data $Context\ f\ a = Term\ (f\ (Context\ f))$
$\quad\quad\quad\quad\quad\quad\quad\quad \mid\ Hole\ a$

Applying Tree Homomorphisms

Applying Tree Homomorphisms

$appHom :: (Functor\ f, Functor\ g) \Rightarrow Hom\ f\ g \rightarrow Term\ f \rightarrow Term\ g$
$appHom = \ldots$

Applying Tree Homomorphisms

Applying Tree Homomorphisms

$appHom :: (Functor\ f, Functor\ g) \Rightarrow Hom\ f\ g \rightarrow Term\ f \rightarrow Term\ g$
$appHom = \ldots$

Example (Desugaring)

data $Sugar\ e = Neg\ e$
type $SigExt\ = Sugar \oplus Sig$

Applying Tree Homomorphisms

Applying Tree Homomorphisms

$$appHom :: (Functor\ f, Functor\ g) \Rightarrow Hom\ f\ g \rightarrow Term\ f \rightarrow Term\ g$$
$$appHom = \ldots$$

Example (Desugaring)

data $Sugar\ e = Neg\ e$
type $SigExt\ = Sugar \oplus Sig$

class $(Functor\ f, Functor\ g) \Rightarrow Desugar\ f\ g$ **where**
  $desugHom :: Hom\ f\ g$

$$desugar :: Desugar\ f\ g \Rightarrow Term\ f \rightarrow Term\ g$$
$$desugar = appHom\ desugHom$$

Implementing Desugaring

The trivial case

instance $(Functor\ f, Functor\ g, f \prec g) \Rightarrow Desugar\ f\ g$ **where**
  $desugHom = simpCxt\ .\ inj$

Implementing Desugaring

Simple contexts

$simpCxt :: Functor\ f \Rightarrow f\ a \rightarrow Context\ f\ a$
$simpCxt = Term\ .\ fmap\ Hole$

The trivial case

$\textbf{instance}\ (Functor\ f, Functor\ g, f \prec g) \Rightarrow Desugar\ f\ g\ \textbf{where}$
$\quad desugHom = simpCxt\ .\ inj$

Implementing Desugaring

Simple contexts

$$simpCxt :: Functor\ f \Rightarrow f\ a \rightarrow Context\ f\ a$$
$$simpCxt = Term\ .\ fmap\ Hole$$

The trivial case

instance $(Functor\ f, Functor\ g, f \prec g) \Rightarrow Desugar\ f\ g$ **where**
$\quad desugHom = simpCxt\ .\ inj$

The interesting case

instance $(Functor\ f, Op \prec f, Val \prec f) \Rightarrow Desugar\ Sugar\ f$ **where**
$\quad desugHom\ (Neg\ x) = iConst\ (-1)\ \text{`}iMult\text{`}\ (Hole\ x)$

Composition of Homomorphisms

Composition operators

$(\circledcirc) :: (Functor\ g, Functor\ h) \Rightarrow Hom\ g\ h \rightarrow Hom\ f\ g \rightarrow Hom\ f\ h$

$(\boxdot) :: Functor\ g \Rightarrow (g\ a \rightarrow a) \rightarrow Hom\ f\ g \rightarrow (f\ a \rightarrow a)$

Composition of Homomorphisms

Composition operators

$$(\circledcirc) \qquad f \xrightarrow[\mathit{Hom}\ f\ g]{\phi} g \xrightarrow[\mathit{Hom}\ g\ h]{\psi} h, \qquad f \xrightarrow[\mathit{Hom}\ f\ h]{\phi\ \circledcirc\ \psi} h$$

Composition of Homomorphisms

Composition operators

$$(\boxdot) \qquad f \xrightarrow[\mathit{Hom}\ f\ g]{\phi} g \xrightarrow[g\ a \to a]{\psi} a, \qquad f \xrightarrow[f\ a \to a]{\phi \boxdot \psi} a$$

Composition of Homomorphisms

Composition operators

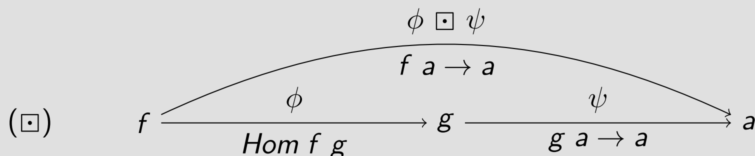
$$(\boxdot) \qquad f \xrightarrow[\text{Hom } f\ g]{\phi} g \xrightarrow[g\ a \to a]{\psi} a, \qquad f \xrightarrow[f\ a \to a]{\phi \boxdot \psi} a$$

Example

$evalDesug :: Term\ SigExt \to Term\ Val$
$evalDesug = eval\ .\ desugar$

Composition of Homomorphisms

Composition operators



Example

$evalDesug :: Term\ SigExt \rightarrow Term\ Val$
$evalDesug = cata\ (evalAlg \boxdot desugarHom)$

Composition of Homomorphisms

Composition operators

$(\boxdot)$

$$f \xrightarrow[\mathit{Hom}\ f\ g]{\phi} g \xrightarrow[g\ a \to a]{\psi} a \qquad \phi \boxdot \psi : f\ a \to a$$

Example

$$\mathit{evalDesug} :: \mathit{Term}\ \mathit{SigExt} \to \mathit{Term}\ \mathit{Val}$$
$$\mathit{evalDesug} = \mathit{cata}\ (\mathit{evalAlg} \boxdot \mathit{desugarHom})$$

Short-cut fusion!

This can be implemented as GHC rewrite rules!

$$\mathit{cata}\ f\ .\ \mathit{appHom}\ g \quad \rightsquigarrow \quad \mathit{cata}\ (f \boxdot g)$$
$$\mathit{appHom}\ f\ .\ \mathit{appHom}\ g \quad \rightsquigarrow \quad \mathit{appHom}\ (f \circledcirc g)$$
$$\vdots$$

Other Extensions

- monadic algebras: $f\ a \to m\ a$
- monadic tree homomorphisms: $f\ a \to m\ (Context\ g\ a)$
- coalgebras & monadic coalgebras
  - generating terms $\leadsto$ e.g. for QuickCheck
- generic functions
  - e.g. size, querying, unification, matching ...
  - using generalised $foldl :: (a \to b \to a) \to a \to [a] \to b$
- generic term rewriting
  - e.g. for performing program transformations
- mutually recursive data types [Yakushev et al. 2009]
  - by adding additional type argument to the signatures

Outline

1. Compositional Data Types
2. Extensions
3. Practical Considerations
4. Current Work

Runtime Comparison

This is all nice, but how does it affect runtime performance?

Runtime Comparison

This is all nice, but how does it affect runtime performance?

Slowdown factors compared to standard data types

Function	hand-written	random (10)	random (20)
inferDesug	1.11	1.52	0.82
inferDesugM	1.38	1.61	0.84
infer	2.39	2.29	2.65
inferM	1.06	1.30	1.68
evalDesug	2.64	1.79	0.89
evalDesugM	4.34	3.47	2.98
eval	2.58	1.84	1.64
evalDirect	6.10	3.96	3.62
evalM	3.41	4.78	7.52
evalDirectM	5.72	4.90	4.56

Runtime Comparison

This is all nice, but how does it affect runtime performance?

Slowdown factors compared to standard data types

Function	hand-written	random (10)	random (20)
inferDesug	(3.38) 1.11	(3.45) 1.52	(3.14) 0.82
inferDesugM	(2.68) 1.38	(2.87) 1.61	(2.79) 0.84
infer	2.39	2.29	2.65
inferM	1.06	1.30	1.68
evalDesug	(6.40) 2.64	(3.13) 1.79	(4.74) 0.89
evalDesugM	(7.32) 4.34	(6.22) 3.47	(9.69) 2.98
eval	2.58	1.84	1.64
evalDirect	6.10	3.96	3.62
evalM	3.41	4.78	7.52
evalDirectM	5.72	4.90	4.56

Runtime Comparison

This is all nice, but how does it affect runtime performance?

Slowdown factors compared to standard data types

Function	hand-written	random (10)	random (20)
inferDesug	(3.38) 1.11	(3.45) 1.52	(3.14) 0.82
inferDesugM	(2.68) 1.38	(2.87) 1.61	(2.79) 0.84
infer	2.39	2.29	2.65
inferM	1.06	1.30	1.68
evalDesug	(6.40) 2.64	(3.13) 1.79	(4.74) 0.89
evalDesugM	(7.32) 4.34	(6.22) 3.47	(9.69) 2.98
eval	2.58	1.84	1.64
evalDirect	6.10	3.96	3.62
evalM	3.41	4.78	7.52
evalDirectM	5.72	4.90	4.56
desugHom	$3.6 \cdot 10^{-1}$	$5.0 \cdot 10^{-3}$	$6.1 \cdot 10^{-6}$
desugCata	$1.8 \cdot 10^{-1}$	$4.41 \cdot 10^{-3}$	$5.3 \cdot 10^{-6}$

Outline

1. Compositional Data Types
2. Extensions
3. Practical Considerations
4. Current Work

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

- Support for binders via parametric higher-order abstract syntax
- algebras with nested monadic effect
  $f\ (Term\ v) \to m\ (Term\ v)$
- beyond tree homomorphisms:
  - attribute grammars
  - modular tree transducers

Current Work

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

Other extensions

- Support for binders via parametric higher-order abstract syntax
- algebras with nested monadic effect
  $f\ (Term\ v) \to m\ (Term\ v) \rightsquigarrow f\ (Term\ (m \oplus v)) \to Term\ (m \oplus v)$
- beyond tree homomorphisms:
  - attribute grammars
  - modular tree transducers

Current Work

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

Other extensions

- Support for binders via parametric higher-order abstract syntax
- algebras with nested monadic effect
  $f\ (Term\ v) \to m\ (Term\ v) \rightsquigarrow f\ (Term\ (m \oplus v)) \to Term\ (m \oplus v)$
- beyond tree homomorphisms:
  - attribute grammars
  - modular tree transducers

Try it yourself: `http://hackage.haskell.org/package/compdata`

References

Wouter Swierstra.
Data types à la carte.
Journal of Functional Programming, 2008.

A. R. Yakushev, S. Holdermans, A. Löh and J. Jeuring.
Generic programming with fixed points for mutually recursive datatypes.
Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009.

Runtime Comparison – Generic Programming

slowdown factors compared to standard data types

Function	hand-written	random (10)	random (20)
contVar	1.92	1.97	3.22
freeVars	1.23	1.26	1.41

Runtime Comparison – Generic Programming

slowdown factors compared to standard data types

Function	hand-written	random (10)	random (20)
contVar	1.92	1.97	3.22
freeVars	1.23	1.26	1.41
contVarC	10.05	7.01	11.68
contVarU	8.24	5.64	11.21
freeVarsC	2.34	2.04	1.68
freeVarsU	2.03	1.75	1.58

Annotating Trees

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Annotating Trees

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Constant Products on Signatures

$\textbf{data}\ (f \mathbin{\&} a)\ e = f\ e \mathbin{\&} a$

Annotating Trees

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Constant Products on Signatures

data $(f \mathbin{\&} a)\ e = f\ e \mathbin{\&} a$

Example

data $Val\ e = Const\ Int$
$\quad\quad\quad\quad\quad | \ Pair\ e\ e$

Annotating Trees

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Constant Products on Signatures

data $(f \,\&\, a)\; e = f\; e \,\&\, a$

Example

data $Val'e = Const\; Int\; SrcPos$
$\quad\quad\quad\quad\quad\mid Pair\; e\; e\; SrcPos$

Annotating Trees

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Constant Products on Signatures

$\mathbf{data}\ (f \mathbin{\&} a)\ e = f\ e \mathbin{\&} a$

Example

$$\begin{aligned}\mathbf{data}\ Val'\, e = {} & Const\ Int\ SrcPos \\ \mid {} & Pair\ e\ e\ SrcPos\end{aligned}$$

$$Val' \cong Val \mathbin{\&} SrcPos$$

Propagating Annotations

Annotations are easily propagated through homomorphisms

$propAnn :: Functor\ g \Rightarrow Hom\ f\ g \rightarrow Hom\ (f\ \&\ a)\ (g\ \&\ a)$

Propagating Annotations

Annotations are easily propagated through homomorphisms

$propAnn :: Functor\ g \Rightarrow Hom\ f\ g \rightarrow Hom\ (f\ \&\ a)\ (g\ \&\ a)$

Example

$desugar :: Term\ SigExt \rightarrow Term\ Sig$

Propagating Annotations

Annotations are easily propagated through homomorphisms

$propAnn :: Functor\ g \Rightarrow Hom\ f\ g \rightarrow Hom\ (f \,\&\, a)\ (g \,\&\, a)$

Example

$desugar :: Term\ SigExt \rightarrow Term\ Sig$

$desugar' :: Term\ (SigExt \,\&\, SrcPos) \rightarrow Term\ (Sig \,\&\, SrcPos)$
$desugar' = propAnn\ desugar$

Eliminate Boilerplate Code

Template Haskell

We use Template Haskell to eliminate boilerplate code:

- instance declarations for *Functor*, *Foldable* etc.
- smart constructors: *iConst*, *iMult* etc.
- propagating algebras & homomorphisms to compound signatures.

Eliminate Boilerplate Code

Template Haskell

We use Template Haskell to eliminate boilerplate code:

- instance declarations for *Functor*, *Foldable* etc.
- smart constructors: *iConst*, *iMult* etc.
- propagating algebras & homomorphisms to compound signatures.

Example

```
$(derive [makeFunctor, makeTraversable, makeFoldable,
          makeEqF, makeShowF, smartConstructors]
         [''Val, ''Op, ''Sugar])
$(derive [liftSum] [''Eval])
```

Library Example

```
import Data.Comp
import Data.Comp.Derive
import Data.Comp.Show ()
import Data.Comp.Desugar

data Val e    = Const Int | Pair e e
data Op e     = Mult e e | Fst e
data Sugar e  = Neg e
type Sig      = Op ⊕ Val
type SigExt   = Sugar ⊕ Sig

$(derive [makeFunctor, makeFoldable, makeTraversable, makeShowF, smartConstructors] [''Val, ''Op, ''Sugar])

class Eval f v where
  evalAlg :: Alg f (Term v)

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

eval :: (Functor f, Eval f v) ⇒ Term f → Term v
eval = cata evalAlg

instance (Val ≺ v) ⇒ Eval Val v where
  evalAlg = inject

instance (Val ≺ v) ⇒ Eval Op v where
  evalAlg (Mult x y) = case (project x, project y) of (Just (Const n), Just (Const m)) → iConst $ n ∗ m
  evalAlg (Fst x)    = case project x of Just (Pair v _) → v

instance (Op ≺ f, Val ≺ f, Functor f) ⇒ Desugar Sugar f where
  desugHom′ (Neg x) = iConst (−1) ‘iMult‘ x

eval′ :: Term SigExt → Term Val
eval′ = eval . (desugar :: Term SigExt → Term Sig)
```