Compositional Data Types

A Report from the Field

Patrick Bahr
paba@diku.dk

University of Copenhagen, Department of Computer Science

Fourth DIKU-IST Joint Workshop on
Foundations of Software,
January 10-14, 2011

joint work with Tom Hvitved and Morten Ib Nielsen

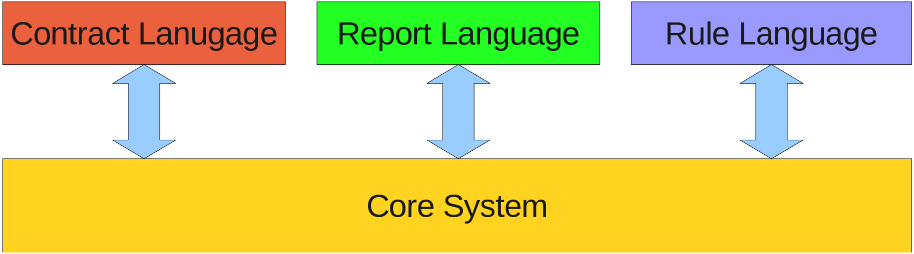
Outline

1. Compositional Data Types
2. A Toolbox for Prototyping Programming Languages
3. Conclusions

Outline

The Setting

Domain-Specific Languages in POETS



- 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!

How Can we Compose Data Structures?

. . . and Functions Defined on Them?

- This is easy on non-recursive data structures.
- Composition by sum or product.

For recursively defined data structures this is different.

Example (A simple expression language)

```
data Expr = Val Int
          | Add Expr Expr

eval :: Expr -> Int
eval (Val x ) = x
eval (Add x y) = eval x + eval y
```

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!

- data types: decoupling of signature and term construction
  - isolated signature (expression data type without recursion)
  - explicit recursive construction of terms over arbitrary signatures
- functions: decoupling of pattern matching and recursion
  - functions are defined on signatures
  - recursion is added separately
- signatures (+ functions defined on them) can be composed

Decoupling Signature and Term Construction

The data type contains both the signature of operations and the inductive definition of terms over them through recursion.

```
data Expr  = Val Int
           | Add Expr Expr
```

Remove recursion from the definition

```
data Sig e = Val Int
           | Add e e
```

Recursion can be added separately

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

Term Sig $\cong$ Expr

Combining Signatures

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

Direct sum of signatures

Type constructor `:+:` of kind `(* -> *) -> (* -> *) -> (* -> *)`:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

Example

```
data Sig e = Val Int
           | Add e e

data Val e = Val Int
data Add e = Add e e
```

`Val :+: Add` $\cong$ `Sig`

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 -> a`
- apply the resulting function recursively on the term: `Term f -> a`
- combine functions using type classes

Algebras

```
class Eval f where
  evalAlg :: f Int -> Int
```

Applying a function recursively to a term

```
algHom :: Functor f => (f a -> a) -> Term f -> a
algHom f (Term t) = f (fmap (algHom f ) t)
```

Defining Algebras

On the singleton signatures

```
instance Eval Val where
  evalAlg (Val x) = x
instance Eval Add where
  evalAlg (Add x y) = x + y
```

On sums of signatures

```
instance (Eval f , Eval g)
  => Eval (f :+: g) where
    evalAlg (Inl x) = evalAlg x
    evalAlg (Inr y) = evalAlg y
```

On sums of signatures

```
eval :: (Functor f, Eval f) => Term f -> Int
eval = algHom evalAlg
```

Outline

1. Compositional Data Types
2. A Toolbox for Prototyping Programming Languages
3. Conclusions

Using Compositional Data Types

Using Compositional Data Types in POETS

- Coarse-grained partition into only a few atomic signatures
  - one for base values
  - one for shared operations
  - operations for each individual language
  - syntactic sugar for each individual language
- similar on the type language

Now that we have this structure in place, can we make further use of it?

Products on Signatures

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

Type constructor `:*:` of kind `(* -> *) -> * -> (* -> *)`:

```
data (f :*: a) e = f e :*: a
```

Example

```
data Sig' e = Val Int SrcPos
            | Add e e SrcPos
```

`Sig'` ≅ `Sig :*: SrcPos`

Dealing with Annotations

Strip away annotations

```
stripP :: (s :*: p) a -> s a
stripP (v :*: _) = v

stripPTerm :: (Functor s)
           => Term (s:*:p) -> Term s
stripPTerm = algHom (Term . stripP)
```

Ignoring annotations

```
liftPTerm :: (Functor s)
       => (Term s -> t) -> (Term (s :*: p) -> t)
liftPTerm f = f . stripPTerm
```

This can be extended to annotations on signature built with sums.

Limitations

Propagation of annotations

How can we lift a function `Term f -> Term g`
to a function `Term (f :*: p) -> Term (g :*: p)`?

- Even if function is given as algebra `a :: f (Term g) -> Term g` this does not work:
  `a . fmap stripP` is of type `f (Term (g :*: p)) -> Term g`
- We could derive an algebra from that, but then result has uniformly the same annotation.

Composition of algebras

Given two algebras `a :: f (Term g) -> Term g` and `b :: g B -> B`, how do we compose them to an algebra `f B -> B`?

- Straightforward composition `homAlg b . a` is of type `f (Term g) -> A`

An Example

Example (Syntactic Sugar)

```
type Exp = Core :+: Sugar

desugarAlg :: Exp (Term Core) -> Term Core

desugar :: Term Exp -> Term Core
desugar = algHom desugarAlg
```

Specialising Algebras

Problem

```
desugarAlg :: Exp (Term Core) -> Term Core
```

- Algebras are too general!
- We have to employ the fact that the domain consists of terms!
- We need something more polymorphic!

First attempt: Signature Transformation

```
desugarAlg :: Exp a -> Core a
```

- This is often too restrictive!
- Each "layer" of a term over `Exp` has to be transformed into exactly one "layer" of a term over `Core`.
  - $x > y \quad \rightsquigarrow \quad y < x$ ✔
  - $x - y \quad \rightsquigarrow \quad x + (-y)$ ✘

Contexts and Term Homomorphisms

Generalise terms to contexts

```
data Context f a = Term (f (Term f))
                 | Hole a
```

From signature transformations to term homomorphisms

```
desugarAlg :: Exp a -> Context Core a
```

Term homomorphisms

- `type TermHom f g = forall a . f a -> Context g a`
- Term homomorphisms (a.k.a. tree homomorphisms) are the term algebras that are defined uniformly. Hence, the polymorphism!

Applying term homomorphisms

```
termHom :: (Functor f, Functor g)
  => TermHom f g -> Term f -> Term g
```

Propagating Annotations

Propagating Annotations

```
constP :: (Functor f)
       => p -> Context f a -> Context (f :*: p) a
constP p (Hole a) = Hole a
constP p (Term t) = Term (fmap (constP p) t :*: p)

liftPTermAlg :: (Functor g)
     => TermHom f g -> TermHom (f :*: p) (g :*: p)
liftPTermAlg f (v :*: p) = constP p (f v)
```

composing term homomorphisms (and algebras)

```
compTermHom :: (Functor g, Functor h) =>
   TermHom g h -> TermHom f g -> TermHom f h
compAlg :: (Functor g) =>
   (g a -> a) -> TermHom f g -> (f a -> a)
```

Terms as Contexts without Holes

Contexts with GADTs

```
data Cxt :: * -> (* -> *) -> * -> * where
            Term :: f (Cxt h f a) -> Cxt h f a
            Hole :: a -> Cxt Hole f a

type Context = Cxt Hole
type Term f = Cxt NoHole f Nothing

data Hole
data NoHole
data Nothing
```

- ⇝ Generalise initial algebra semantics to free algebra semantics.
- ⇝ Terms & initial algebras are a special case.

Other Extensions

- monadic algebras
  - using generalised `sequence :: [m a] -> m [a]`
- (monadic) coalgebras
  - generating terms $\rightsquigarrow$ e.g. for QuickCheck
- generic functions
  - e.g. size, querying, unification, matching . . .
  - using generalised `foldl :: (a -> b -> a) -> a -> [a] -> 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
  - can be extended to rational sets of trees (by bottom-up tree automata on the type level)

Performance Impact

- Composable data types simplify function definitions, provide flexibility, reduce boilerplate code and avoid code duplication!
- But how does it affect runtime performance?

The setting

- Three signatures:
  - values: integers, Booleans, pairs
  - core language operations: $+$, $*$, if, $=$, $<$, $\wedge$, $\neg$, projections
  - syntactic sugar: negation, $-$, $>$, $\vee$, $\Rightarrow$
- a number of different typical functions:
  - type inference
  - evaluation to normal form,
  - “desugaring” (reduce syntactic sugar to the core language)
  - computing free variables
- We compare this to an ordinary implementation using standard data types and recursive functions.

Runtime Comparison

slowdown factors compared to standard data types

function	n=16	n=63	n=1290	n=111,279
desugarType	4.8	5.2	5.3	4.1
desugarType'	4.2	4.9	5.0	2.5
typeSugar	3.2	3.7	3.7	4.6
desugarEval	15	11	11	15
desugarEval'	13	10	9.8	8.8
evalSugar	12	9.4	7.4	18
desugarEvalPure	11	7.1	6.4	11
desugarEvalPure'	6.5	4.4	4.0	3.8
evalSugarPure	7.3	7.0	4.0	3.6
freeVars	1.3	1.6	1.4	1.6
desugar	0.33	0.08	$1.2 \cdot 10^{-3}$	$1.5 \cdot 10^{-5}$

- monadic functions are in blue
- underlined variants use composition of algebras

Outline

Applications for Compositional Data Types

Drawbacks

- not as straightforward as ordinary data types
- type errors are sometimes hard to decypher
- memory and runtime overhead

Benefits

- minimises code duplication
- functions on shared structures can be shared as well
- it is often more convenient to define functions
- more flexible (algebras can be easily modified / lifted)
- only little runtime overhead
- sometimes asymptotically faster that ordinary recursive functions on recursive data types

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.