Compositional Data Types

Patrick Bahr Tom Hvitved

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
{paba,hvitved}@diku.dk

Abstract

Building on Wouter Swierstra's *Data types à la carte*, we present a comprehensive Haskell library of *compositional data types* suitable for practical applications. In this framework, data types and functions on them can be defined in a modular fashion. We extend the existing work by implementing a wide array of recursion schemes including monadic computations. Above all, we generalise recursive data types to *contexts*, which allow us to characterise a special yet frequent kind of catamorphisms. The thus established notion of *term homomorphisms* allows for flexible reuse and enables short-cut fusion style deforestation which yields considerable speedups. We demonstrate our framework in the setting of compiler construction, and moreover, we compare compositional data types with generic programming techniques and show that both are comparable in run-time performance and expressivity while our approach allows for stricter types. We substantiate this conclusion by lifting compositional data types to mutually recursive data types and generalised algebraic data types. Lastly, we compare the run-time performance of our techniques with traditional implementations over algebraic data types. The results are surprisingly good.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data Types and Structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and Recursion Schemes

General Terms Design, Languages, Performance

Keywords algebraic programming, reusability, deforestation, mutual recursion

1. Introduction

Static typing provides a valuable tool for expressing invariants of a program. Yet, all too often, this tool is not leveraged to its full extent because it is simply not practical. Vice versa, if we want to use the full power of a type system, we often find ourselves writing large chunks of boilerplate code or—even worse—duplicating code. For example, consider the type of non-empty lists. Even though having such a type at your disposal is quite useful, you would rarely find it in use since—in a practical type system such as Haskell's—it would require the duplication of functions which work both on general and non-empty lists.

The situation illustrated above is an ubiquitous issue in compiler construction: In a compiler, an abstract syntax tree (AST) is produced from a source file, which then goes through different transformation and analysis phases, and is finally transformed into the target code. As functional programmers, we want to reflect the changes of each transformation step in the type of the AST. For example, consider the desugaring phase of a compiler which reduces syntactic sugar to the core syntax of the object language. To properly reflect this structural change also in the types, we have to create and maintain a variant of the data type defining the AST for the core syntax. Then, however, functions have to be defined for both types independently, i.e. code cannot be readily reused for both types! If we add annotations in an analysis step of the compiler, the type of the AST has to be changed again. But some functions should ignore certain annotations while being aware of others. And it gets even worse if we allow extensions to the object language that can be turned on and off independently, or if we want to implement several domain-specific languages which share a common core. This quickly becomes a nightmare with the choice of either duplicating lots of code or giving up static type safety by using a huge AST data type that covers all cases.

The essence of this problem can be summarised as the *Expression Problem*, i.e. "the goal [...] to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety" [25]. Wouter Swierstra [20] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits invaluable simplicity and clarity, it lacks abilities necessary to apply it in a practical setting beyond the confined simplicity of the expression problem.

The goal of this paper is to extend Swierstra's work in order to enhance its flexibility, improve its performance and broaden its scope of applications. In concrete terms, our contributions are:

- We implement recursion schemes other than catamorphisms (Section 4.5) and also account for recursion schemes over *monadic computations* (Section 3.2).
- We show how *generic programming* techniques can be efficiently implemented on top of the compositional data types framework (Section 3.1), providing a performance competitive with top-performing dedicated generic programming libraries.
- By generalising terms—i.e. recursive data types—to contexts—i.e. recursive data types with holes—we are able to capture the notion of term homomorphisms (Section 4.4), a special but common case of term algebras. In contrast to general algebras, term homomorphisms can easily be lifted to different data types, readily reused, and composed (also with algebras). The latter allows us to perform optimisations via short-cut fusion rules that provide considerable speedups (Section 6.2).

WGP'11, September 18, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0861-8/11/09. . . $10.00

- We further extend the scope of applications by capturing compositional mutually recursive data types and GADTs via the construction of Johann and Ghani [6] (Section 5).
- Finally, we show the practical competitiveness of compositional data types by reducing their syntactic overhead using Template Haskell [18] (Section 6.1), and by comparing the run-time of typical functions with corresponding implementations over ordinary recursive data types (Section 6.2).

The framework of compositional data types that we present here is available from Hackage[1]. It contains the complete source code, numerous examples, and the benchmarks whose results we present in this paper. All code fragments presented throughout the paper are written in Haskell [9].

2. Data Types à la Carte

This section serves as an introduction to Swierstra's *Data types à la carte* [20] (from here on, *compositional data types*), using our slightly revised notation and terminology. We demonstrate the application of compositional data types to a setting consisting of a family of expression languages that pairwise share some sublanguage, and operations that provide transformations between some of them. We illustrate the merits of this method on two examples: expression evaluation and desugaring.

2.1 Evaluating Expressions

Consider a simple language of expressions over integers and tuples, together with an evaluation function:

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

data Value = VConst Int | VPair Value Value

eval :: Exp → Value
eval (Const n)  = VConst n
eval (Mult x y) = let VConst m = eval x
                      VConst n = eval y
                  in  VConst (m ∗ n)
eval (Pair x y) = VPair (eval x) (eval y)
eval (Fst x)    = let VPair v _ = eval x in v
eval (Snd x)    = let VPair _ v = eval x in v
```

In order to statically guarantee that the evaluation function produces values—a sublanguage of the expression language—we are forced to replicate parts of the expression structure in order to represent values. Consequently, we are also forced to duplicate common functionality such as pretty printing. Compositional data types provide a solution to this problem by relying on the well-known technique [11] of separating the recursive structure of terms from their signatures (functors). Recursive functions, in the form of catamorphisms, can then be specified by algebras on these signatures.

For our example, it suffices to define the following two signatures in order to separate values from general expressions:

```
data Val e = Const Int | Pair e e

data Op e = Mult e e | Fst e | Snd e
```

The novelty of compositional data types then is to combine signatures—and algebras defined on them—in a modular fashion, by means of a formal sum of functors:

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

It is easy to show that f :+: g is a functor whenever f and g are functors. We thus obtain the combined signature for expressions:

```
type Sig = Op :+: Val
```

Finally, the type of terms over a (potentially compound) signature f can be constructed as the fixed point of the signature f:

```
data Term f = Term {unTerm :: (f (Term f))}
```

We then have that $Term\ Sig \cong Exp$ and $Term\ Val \cong Value$.[2]

However, using compound signatures constructed by formal sums means that we have to explicitly tag constructors with the right injections. For instance, the term $1 * 2$ has to be written as

```
e :: Term Sig
e = Term $ Inl $ Mult (Term $ Inr $ Const 1)
                      (Term $ Inr $ Const 2)
```

Even worse, if we want to embed the term e into a type over an extended signature, say with syntactic sugar, then we have to add another level of injections *throughout* its definition. To overcome this problem, injections are derived using a type class:

```
class sub :≺: sup where
  inj :: sub a → sup a
  proj :: sup a → Maybe (sub a)
```

Using *overlapping instance* declarations, the sub-signature relation :≺: can be constructively defined. However, due to restrictions of the type class system, we have to restrict ourselves to instances of the form f :≺: g where f is atomic, i.e. not a sum, and g is a right-associative sum, e.g. g_1 :+: $(g_2$:+: $g_3)$ but not $(g_1$:+: $g_2)$:+: g_3.[3] Using the carefully defined instances for :≺:, we can then define injection and projection functions:

```
inject :: (g :≺: f) ⇒ g (Term f) → Term f
inject = Term . inj

project :: (g :≺: f) ⇒ Term f → Maybe (g (Term f))
project = proj . unTerm
```

Additionally, in order to reduce the syntactic overhead, we use smart constructors—which can be derived automatically, cf. Section 6.1—that already comprise the injection:

```
iMult :: (Op :≺: f) ⇒ Term f → Term f → Term f
iMult x y = inject $ Mult x y
```

The term $1 * 2$ can now be written without syntactic overhead

```
e :: Term Sig
e = iConst 1 `iMult` iConst 2
```

and we can even give e the *open* type $(Val$:≺: f, Op :≺: $f) \Rightarrow Term\ f$. That is, e can be used as a term over any signature containing at least values and operators.

Next, we want to define the evaluation function, i.e. a function of type $Term\ Sig \rightarrow Term\ Val$. To this end, we define the following *algebra class Eval*:

```
type Alg f a = f a → a

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

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

The instance declaration for sums is crucial, as it defines how to combine instances for the different signatures—yet the structure of its declaration is independent from the particular algebra class, and it can be automatically derived for any algebra. Thus, we will omit the instance declarations lifting algebras to sums from now on. The actual evaluation function can then be obtained from instances of this algebra class as a *catamorphism*. In order to perform the necessary recursion, we require the signature f to be an instance of *Functor* providing the method $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$:

```
cata :: Functor f ⇒ Alg f a → Term f → a
cata f = f . fmap (cata f) . unTerm

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

[1] See `http://hackage.haskell.org/package/compdata`.

[2] For clarity, we have omitted the strictness annotation to the constructor *Term* which is necessary in order to obtain the indicated isomorphisms.

[3] We encourage the reader to consult Swierstra's original paper [20] for the proper definition of the :≺: relation.

What remains is to define the algebra instances for *Val* and *Op*. One approach is to define instances *Eval Val Val* and *Eval Op Val*, however such definitions are problematic if we later want to add a signature to the language which also extends the signature for values, say with Boolean values. We could hope to achieve such extendability by defining the instance

```
instance (Eval f v, v :≺: v′) ⇒ Eval f v′
```

but this is problematic for two reasons: First, the relation :≺: only works for atomic left-hand sides, and second, we can in fact not define this instance because the function $evalAlg :: f\ (Term\ v) \to Term\ v$ cannot be lifted to the type $f\ (Term\ v') \to Term\ v'$, as the type of the domain also changes. Instead, the correct approach is to leave the instance declarations open in the target signature:

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

instance (Val :≺: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst $ projC x ∗ projC y
  evalAlg (Fst x)    = fst $ projP x
  evalAlg (Snd x)    = snd $ projP x

projC :: (Val :≺: v) ⇒ Term v → Int
projC v = case project v of Just (Const n) → n

projP :: (Val :≺: v) ⇒ Term v → (Term v, Term v)
projP v = case project v of Just (Pair x y) → (x, y)
```

Notice how the constructors *Const* and *Pair* are treated with a single *inject*, as these are already part of the value signature.

2.2 Adding Sugar on Top

We now consider an extension of the expression language with *syntactic sugar*, exemplified via negation and swapping of pairs:

```
data Sug e = Neg e | Swap e

type Sig′  = Sug :+: Sig
```

Defining a desugaring function $Term\ Sig' \to Term\ Sig$ then amounts to instantiating the following algebra class:

```
class Desug f g where
  desugAlg :: Alg f (Term g)

desug :: (Functor f, Desug f g) ⇒ Term f → Term g
desug = cata desugAlg
```

Using overlapping instances, we can define a default translation for *Val* and *Op*, so we only have to write the "interesting" cases:

```
instance (f :≺: g) ⇒ Desug f g where
  desugAlg = inject

instance (Val :≺: f, Op :≺: f) ⇒ Desug Sug f where
  desugAlg (Neg x)  = iConst (−1) ‘iMult‘ x
  desugAlg (Swap x) = iSnd x ‘iPair‘ iFst x
```

Note how the context of the last instance reveals that desugaring of the extended syntax requires a target signature with at least base values, *Val* :≺: *f*, and operators, *Op* :≺: *f*. By composing *desug* and *eval*, we get an evaluation function for the extended language:

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

The definition above shows that there is a small price to pay for leaving the algebra instances open: We have to annotate the desugaring function in order to pin down the intermediate signature *Sig*.

3. Extensions

In this section, we introduce some rather straightforward extensions to the compositional data types framework: Generic programming combinators, monadic computations, and annotations.

3.1 Generic Programming

Most of the functions that are definable in the common generic programming frameworks [14] can be categorised as either query functions $d \to r$, which analyse a data structure of type d by extracting some relevant information of type r from parts of the input and compose them, or as transformation functions $d \to d$, which recursively apply some type preserving functions to parts of the input. The benefit that generic programming frameworks offer is that programmers only need to specify the "interesting" parts of the computation. We will show how we can easily reproduce this experience on top of compositional data types.

Applying a type-preserving function recursively throughout a term can be implemented easily. The function below applies a given function in a bottom-up manner:

```
trans :: Functor f ⇒ (Term f → Term f)
                   → (Term f → Term f)
trans f = cata (f . Term)
```

Other recursion schemes can be implemented just as easily.

In order to implement generic querying functions, we need a means to combine the result of querying a functorial value. The standard type class *Foldable* generalises folds over lists and thus provides us with exactly the interface we need:[4]

```
class Foldable f where
  foldl :: (a → b → a) → a → f b → a
```

For example, an appropriate instance for the functor *Val* can be defined like this:

```
instance Foldable Val where
  foldl _ a (Const _)  = a
  foldl f a (Pair x y) = (a ‘f‘ x) ‘f‘ y
```

With *Foldable*, a generic querying function can be implemented easily. It takes a function $q :: Term\ f \to r$ to query a single node of the term and a function $c :: r \to r \to r$ to combine two results:

```
query :: Foldable f ⇒ (Term f → r)
                     → (r → r → r) → Term f → r
query q c t =
  foldl (λr x → r ‘c‘ query q c x) (q t) (unTerm t)
```

We can instantiate this scheme, for example, to implement a generic size function:

```
gsize :: Foldable f ⇒ Term f → Int
gsize = query (const 1) (+)
```

A very convenient scheme of query functions introduced by Mitchell and Runciman [12], in the form of the *universe* combinator, simply returns a list of all subterms. Specific queries can then be written rather succinctly using list comprehensions. Such a combinator can be implemented easily via *query*:

```
subs :: Foldable f ⇒ Term f → [Term f]
subs = query (λx → [x]) (++)
```

However, in order to make the pattern matching in list comprehensions work, we need to project the terms to the functor that contains the constructor we want to match against:

```
subs′ :: (Foldable f, g :≺: f) ⇒ Term f → [g (Term f)]
subs′ = mapMaybe project . subs
```

With this in place we can for example easily sum up all integer literals in an expression:

```
sumInts :: (Val :≺: f) ⇒ Term f → Int
sumInts t = sum [i | Const i ← subs′ t]
```

This shows that we can obtain functionality similar to what dedicated generic programming frameworks offer. In contrast to generic programming, however, the compositional data type approach provides additional tools that allow us to define functions with a stricter type that reflects the underlying transformation. For example, we could have defined the desugaring function in terms of *trans*, but that would have resulted in the "weaker" type $Term\ Sig' \to Term\ Sig'$ instead of $Term\ Sig' \to Term\ Sig$. The latter type witnesses that indeed all syntactic sugar is removed!

[4] *Foldable* also has other fold functions, but they are derivable from *foldl* and are not relevant for our purposes.

Nevertheless, the examples show that at least the querying combinators *query* and *subs′* provide an added value to our framework. Moreover, by applying standard optimisation techniques we can obtain run-time performance comparable with top-performing generic programming libraries (cf. Section 6.2). In contrast to common generic programming libraries [14], we only considered combinators that work on a single recursive data type. This restriction is lifted in Section 5 when we move to mutually recursive data types.

3.2 Monadic Computations

We saw in Section 2 how to realise a modular evaluation function for a small expression language in terms of catamorphisms defined by algebras. In order to deal with type mismatches, we employed non-exhaustive case expressions. Clearly, it would be better to use a monad instead. However, a monadic carrier type *m a* would yield an algebra *f (m a) → m a* which means that we have to explicitly sequence the nested monadic values of the argument. What we would rather like to do is to write a *monadic algebra* [3]

```
type AlgM m f a = f a → m a
```

where the nested sequencing is done automatically and thus the monadic type only occurs in the codomain. Again we are looking for a function that we already know from lists:

```
sequence :: Monad m ⇒ [m a] → m [a]
```

The standard type class *Traversable* [10] provides the appropriate generalisation to functors:

```
class (Functor f, Foldable f) ⇒ Traversable f where
  sequence :: Monad m ⇒ f (m a) → m (f a)
  mapM     :: Monad m ⇒ (a → m b) → f a → m (f b)
```

Here, *mapM* is simply the composition of *sequence* and *fmap*.

The definition of a monadic variant of catamorphisms can then be derived by replacing *fmap* with *mapM* and function composition with monadic function composition ⋘:

```
cataM :: (Traversable f, Monad m) ⇒ AlgM m f a
                                    → Term f → m a
cataM f = f ⋘ mapM (cataM f) . unTerm
```

The following definitions illustrate how monadic catamorphisms can be used to define a *safe* version of the evaluation function from Section 2, which properly handles errors when applied to a *bad term* (using the *Maybe* monad for simplicity):

```
class EvalM f v where
  evalAlgM :: AlgM Maybe f (Term v)

evalM :: (Traversable f, EvalM f v) ⇒ Term f
                                     → Maybe (Term v)
evalM = cataM evalAlgM

instance (Val :≺: v) ⇒ EvalM Val v where
  evalAlgM = return . inject

instance (Val :≺: v) ⇒ EvalM Op v where
  evalAlgM (Mult x y) =
    liftM iConst $ liftM2 (∗) (projCM x) (projCM y)
  evalAlgM (Fst x)     = liftM fst $ projPM x
  evalAlgM (Snd x)     = liftM snd $ projPM x

projCM :: (Val :≺: v) ⇒ Term v → Maybe Int
projCM v = case project v of
  Just (Const n) → return n; _ → Nothing

projPM :: (Val :≺: v) ⇒ Term v
                      → Maybe (Term v, Term v)
projPM v = case project v of
  Just (Pair x y) → return (x, y); _ → Nothing
```

3.3 Products and Annotations

We have seen in Section 2 how the sum :+: can be used to combine signatures. This inevitably leads to the dual construction:

```
data (f :∗: g) a = f a :∗: g a
```

In its general form, the product :∗: seems of little use: Each constructor of *f* can be paired with each constructor of *g*. The special case, however, where *g* is a constant functor, is easy to comprehend yet immensely useful:

```
data (f :&: c) a = f a :&: c
```

Now, every value of type *(f :&: c) a* is value from *f a* annotated with a value in *c*. On the term level, this means that a term over *f :&: c* is a term over *f* in which each subterm is annotated with a value in *c*.

This addresses a common problem in compiler implementations: How to deal with annotations of AST nodes such as source positions or type information which have only a limited lifespan or are only of interest for some parts of the compiler?

Given the signature *Sig* for our simple expression language and a type *Pos* which represents source position information such as a file name and a line number, we can represent ASTs with source position annotations as *Term (Sig :&: Pos)* and write a parser that provides such annotations [22].

The resulting representation yields a clean separation between the actual data—the AST—and the annotation data—the source positions—which is purely supplemental for supplying better error messages. The separation allows us to write a generic function that strips off annotations when they are not needed:

```
remA :: (f :&: p) a → f a
remA (v :&: _) = v

stripA :: Functor f ⇒ Term (f :&: p) → Term f
stripA = cata (Term . remA)
```

With this in place, we can provide a generic combinator that lifts a function on terms to a function on terms with annotations

```
liftA :: Functor f ⇒ (Term f → t) → Term (f :&: p) → t
liftA f = f . stripA
```

which works for instance for the evaluation function:

```
liftA eval :: Term (Sig :&: Pos) → Term Val
```

But how do we actually define an algebra that uses the position annotations? We are faced with the problem that the product :&: is applied to a *sum*, viz. *Sig = Op :+: Val*. When defining the algebra for one of the summands, say *Val*, we do not have immediate access to the factor *Pos* which is outside of the sum.

We can solve this issue in two ways: (a) Propagating the annotation using a *Reader* monad or (b) providing operations that allow us to make use of the right-distributivity of :&: over :+:. For the first approach, we only need to move from algebras *Alg f a* to monadic algebras *AlgM (Reader p) f a*, for *p* the type of the annotations. Given an algebra class, e.g. for type inference

```
class Infer f where
  inferAlg :: AlgM (Reader Pos) f Type
```

we can lift it to annotated signatures:[5]

```
instance Infer f ⇒ Infer (f :&: Pos) where
  inferAlg (v :&: p) = local (const p) (inferAlg v)
```

When defining the other instances of the class, we can use the monadic function *ask :: Reader Pos Pos* to query the annotation of the current subterm. This provides a clean interface to the annotations. It requires, however, that we define a monadic algebra.

The alternative approach is to distribute the annotations over the sum, i.e. instead of *Sig :&: Pos* we use the type

```
type SigP = Op :&: Pos :+: Val :&: Pos
```

Now, we are able to define a direct instance of the form

```
instance Infer (Val :&: Pos) where
  inferAlg (v :&: p) = ...
```

where we have direct access to the position annotation *p*. However, now we have the dual problem: We do not have immediate access to the annotation at the outermost level of the sum. Hence, we

[5] The standard function *local :: (r → r) → Reader r a → Reader r a* updates the environment by the function given as first argument.

cannot use the function $liftA$ to lift functions to annotated terms. Yet, this direction—propagating annotations outwards—is easier to deal with. We have to generalise the function $remA$ to also deal with annotations distributed over sums. This is an easy exercise:

$\textbf{class}\ RemA\ f\ g \mid f \to g\ \textbf{where}$
$\quad remA :: f\ a \to g\ a$
$\textbf{instance}\ RemA\ (f :\&: p)\ f\ \textbf{where}$
$\quad remA\ (v :\&: \_) = v$
$\textbf{instance}\ RemA\ f\ f'$
$\quad \Rightarrow RemA\ (g :\&: p :+: f)\ (g :+: f')\ \textbf{where}$
$\quad remA\ (Inl\ (v :\&: \_)) = Inl\ v$
$\quad remA\ (Inr\ v) = Inr\ (remA\ v)$

Now the function $remA$ works as before, but it can also deal with signatures such as $SigP$, and the type of $liftA$ becomes:

$(Functor\ f, RemA\ f\ g) \Rightarrow (Term\ g \to t) \to Term\ f \to t$

Both approaches have their share of benefits and drawbacks. The monadic approach provides a cleaner interface but necessitates a monadic style. The explicit distribution is more flexible as it both allows us to access the annotations directly by pattern matching or to thread them through a monad if that is more convenient. On the other hand, it means that adding annotations is not straightforwardly compositional anymore. The annotation $:\&:A$ has to be added to each summand—just like compound signatures are not straightforwardly compositional, e.g. we have to write the sum $f :+: g$, for a signature $f = f_1 :+: f_2$, explicitly as $f_1 :+: f_2 :+: g$.

4. Context Matters

In this section, we will discuss two problems that arise when defining term algebras, i.e. algebras with a carrier of the form $Term\ f$. These problems occur when we want to lift term algebras to algebras on annotated terms, and when trying to compose term algebras. We will show how these problems can be addressed by *term homomorphisms*, a quite common special case of term algebras. In order to make this work, we shall generalise terms to contexts by using generalised algebraic data types (GADTs) [17].

4.1 Propagating Annotations

As we have seen in Section 3.3, it is easy to lift functions on terms to functions on annotated terms. It only amounts to removing all annotations before passing the term to the original function.

But what if we do not want to completely ignore the annotation but propagate it in a meaningful way to the output? Take for example the desugaring function $desug$ we have defined in Section 2 and which transforms terms over Sig' to terms over Sig. How do we lift this function easily to a function of type

$Term\ (Sig' :\&: Pos) \to Term\ (Sig :\&: Pos)$

which propagates the annotations such that each annotation of a subterm in the result is taken from the subterm it originated? For example, in the desugaring of a term $iSwap\ x$ to the term $iSnd\ x$ `$iPair$` $iFst\ x$, the top-most $Pair$-term, as well as the two terms $Snd\ x$ and $Fst\ x$ should get the same annotation as the original subterm $iSwap\ x$.

This propagation is independent of the transformation function. The same scheme can also be used for the type inference function in order to annotate the inferred type terms with the positions of the code that is responsible for each part of the type terms.

It is clear that we will not be able provide a combinator of type

$(Term\ f \to Term\ g) \to Term\ (f :\&: p) \to Term\ (g :\&: p)$

that lifts any function to one that propagates annotations meaningfully. We cannot tell from a plain function of type $Term\ f \to Term\ g$ where the subterms of the result term are originated in the input term. However, restricting ourselves to term algebras will not be sufficient either. That is, also a combinator of type

$Alg\ f\ (Term\ g) \to Alg\ (f :\&: p)\ (Term\ (g :\&: p))$

is out of reach. While we can tell from a term algebra, i.e. a function of type $f\ (Term\ g) \to Term\ g$, that some initial parts of the result term originate from the f-constructor at the root of the input, we do not know which parts. The term algebra only returns a *uniform* term of type $Term\ g$ which provides no information as to which parts were constructed from the f-part of the $f\ (Term\ g)$ argument and which were copied from the $(Term\ g)$-part.

Term algebras are still too general! We need to move to a function type that clearly states which parts are constructed from the "current" top-level symbol in f and which are copied from its arguments in $Term\ g$. In order to express that certain parts are just copied, we can make use of parametric polymorphism.

Instead of an algebra, we can define a function on terms also by a *natural transformation*, a function of type $\forall\ a\ .\ f\ a \to g\ a$. Such a function can only transform an f-constructor into a g-constructor and copy its arguments around. Since the copying is made explicit in the type, defining a function that propagates annotations through natural transformations is straightforward:

$prop :: (f\ a \to g\ a) \to (f :\&: p)\ a \to (g :\&: p)\ a$
$prop\ f\ (v :\&: p) = f\ v :\&: p$

Unfortunately, natural transformations are also quite limited. They only allow us to transform each constructor of the original term to exactly one constructor in the target term. This is for example not sufficient for the desugaring function, which translates a constructor application $iSwap\ x$ into three constructor applications $iSnd\ x$ `$iPair$` $iFst\ x$. In order to lift this restriction, we need to be able to define a function of type $\forall\ a\ .\ f\ a \to Context\ g\ a$ which transforms an f-constructor application to a *g-context* application, i.e. several nested applications of g-constructors potentially with some "holes" filled by values of type a.

We shall return to this idea in Section 4.4.

4.2 Composing Term Algebras

The benefit of having a desugaring function $desug :: Term\ Sig' \to Term\ Sig$, which is able to reduce terms over the richer signature Sig' to terms over the core signature Sig, is that it allows us to easily lift functions that are defined on terms over Sig—such as evaluation and type inference—to terms over Sig':

$eval' :: Term\ Sig' \to Term\ Val$
$eval' = eval\ .\ (desug :: Term\ Sig' \to Term\ Sig)$

However, looking at how $eval$ and $desug$ are defined, viz. as catamorphisms, we notice a familiar pattern:

$eval' = cata\ evalAlg\ .\ cata\ desugAlg$

This looks quite similar to the classic example of *short-cut fusion*:

$$map\ f\ .\ map\ g \quad \leadsto \quad map\ (f\ .\ g)$$

An expression that traverses a data structure twice is transformed into one that only does this once.

To replicate this on terms, we need an appropriately defined composition operator $\circledcirc$ on term algebras that allows us to perform a similar semantics-preserving transformation:

$$cata\ f\ .\ cata\ g \quad \leadsto \quad cata\ (f \circledcirc g)$$

As a result, the input term only needs to be traversed once instead of twice and the composition and decomposition of an intermediate term is avoided. The type of $\circledcirc$ should be

$Alg\ g\ (Term\ h) \to Alg\ f\ (Term\ g) \to Alg\ f\ (Term\ h)$

Since term algebras are functions, the only way to compose them is by first making them compatible and then performing function composition. Given two term algebras $a :: Alg\ g\ (Term\ h)$ and $b :: Alg\ f\ (Term\ g)$, we can turn them into compatible functions by lifting a to terms via $cata$. The problem now is that the composition $cata\ a\ .\ b$ has type $f\ (Term\ g) \to Term\ h$, which is only an algebra if $g = h$. This issue arises due to the simple fact that the carrier of an algebra occurs in both the domain and the codomain of the function! Instead of a term algebra of type $f\ (Term\ g) \to Term\ g$, we need a function type in which the domain is more independent from the codomain in order to allow composition. Again, a type of the form $\forall\ a\ .\ f\ a \to Context\ g\ a$ provides a solution.

4.3 From Terms to Contexts and back

We have seen in the two preceding sections that we need an appropriate notion of *contexts*, i.e. a term which can also contain "holes" filled with values of a certain type. Starting from the definition of terms, we can easily generalise it to contexts by simply adding an additional case:

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

Note that we can obtain a type isomorphic to the one above using summation: $Context\ f\ a \cong Term\ (f :+: K\ a)$ for a type

```
data K a b = K a
```

Since we will use contexts quite often, we will use the direct representation. Moreover, this allows us to tightly integrate contexts into our framework. Since contexts are terms with holes, we also want to go the other way around by defining terms as contexts *without holes*! This will allow us to lift functions defined on terms—catamorphisms, injections etc.—to functions on contexts that provide the original term-valued function as a special case.

The idea of defining terms as contexts without holes can be encoded in Haskell quite easily as a *generalised algebraic data type (GADT)* [17] with a *phantom type Hole*:

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

In this representation, we add an additional type argument that indicates whether the context might contain holes or not. A context that does have a hole must have a type of the form $Cxt\ Hole\ f\ a$. Our initial definition of contexts can be recovered by defining:

```
type Context = Cxt Hole
```

That is, contexts *may* contain holes. On the other hand, terms must not contain holes. This can be defined by:

```
type Term f = ∀ h a . Cxt h f a
```

While this is a natural representation of terms as a special case of the more general concept of contexts, this usually causes some difficulties because of the impredicative polymorphism. We therefore prefer an approximation of this type that will do fine in almost any relevant case. Instead of universal quantification, we use empty data types $NoHole$ and $Nothing$:

```
type Term f = Cxt NoHole f Nothing
```

In practice, this does not pose any restriction whatsoever. Both $NoHole$ and $Nothing$ are phantom types and do not contribute to the internal representation of values. For the former this is obvious, for the latter this follows from the fact that the phantom type $NoHole$ witnesses that the context has indeed no holes which would otherwise enforce the type $Nothing$. Hence, we can transform a term to any context type over any type of holes:

```
toCxt :: Functor f ⇒ Term f → ∀ h a . Cxt h f a
toCxt (Term t) = Term (fmap toCxt t)
```

In fact, $toCxt$ does not change the representation of the input term. Looking at its definition, $toCxt$ is operationally equivalent to the identity. Thus, we can safely use the function $unsafeCoerce :: a \to b$ in order to avoid run-time overhead:

```
toCxt :: Functor f ⇒ Term f → ∀ h a . Cxt h f a
toCxt = unsafeCoerce
```

This representation of contexts and terms allows us to uniformly define functions which work on both types. The function $inject$ can be defined as before, but now has the type

```
inject :: (g :≺: f) ⇒ g (Cxt h f a) → Cxt h f a
```

and thus works for both terms and proper contexts. The projection function has to be extended slightly to accommodate for holes:

```
project :: (g :≺: f) ⇒ Cxt h f a → Maybe (g (Cxt h f a))
project (Term t) = proj t
project (Hole _) = Nothing
```

The relation between terms and contexts can also be illustrated algebraically: If we ignore for a moment the ability to define infinite terms due to Haskell's non-strict semantics, the type $Term\ F$ represents the initial $\mathcal{F}$-algebra which has the carrier $\mathcal{T}(\mathcal{F})$, the terms over signature $\mathcal{F}$. The type of contexts $Context\ F\ X$ on the other hand represents the free $\mathcal{F}$-algebra generated by $\mathcal{X}$ which has the carrier $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the terms over signature $\mathcal{F}$ and variables $\mathcal{X}$.

Thus, for recursion schemes, we can move naturally from catamorphisms, i.e. initial algebra semantics, to free algebra semantics:

```
free :: Functor f ⇒ Alg f b → (a → b) → Cxt h f a → b
free alg v (Term t) = alg (fmap (free alg v) t)
free _   v (Hole x) = v x

freeM :: (Traversable f, Monad m) ⇒
          AlgM m f b → (a → m b) → Cxt h f a → m b
freeM alg v (Term t) = alg =≪ mapM (freeM alg v) t
freeM _   v (Hole x) = v x
```

This yields the central function for working with contexts:

```
appCxt :: Functor f ⇒ Context f (Cxt h f a)
                      → Cxt h f a
appCxt = free Term id
```

This function takes a context whose holes are terms (or contexts) and returns the term (respectively context) that is obtained by merging the two—essentially by removing each constructor $Hole$. Notice how the type variables h and a are propagated from the input context's holes to the return type. In this way, we can uniformly treat both terms and contexts.

4.4 Term Homomorphisms

The examples from Sections 4.1 and 4.2 have illustrated the need for defining functions on terms by functions of the form $\forall\ a . f\ a \to Context\ \ g\ \ a$. Such functions can then be transformed to term algebras via $appCxt$ and, thus, be lifted to terms:

```
termHom :: (Functor f, Functor g) ⇒
   (∀ a . f a → Context g a) → Term f → Term g
termHom f = cata (appCxt . f)
```

In fact, the polymorphism in the type $\forall\ a\ .\ f\ a \to Context\ g\ a$ guarantees that arguments of the functor f can only be copied—not inspected or modified. This restriction captures a well-known concept from tree automata theory:

Definition 1 (term homomorphisms[6] [2, 21])**.** Let $\mathcal{F}$ and $\mathcal{G}$ be two sets of function symbols, possibly not disjoint. For each $n > 0$, let $\mathcal{X}_n = \{x_1, \ldots, x_n\}$ be a set of variables disjoint from $\mathcal{F}$ and $\mathcal{G}$. Let $h_{\mathcal{F}}$ be a mapping which, with $f \in \mathcal{F}$ of arity n, associates a context $t_f \in \mathcal{T}(\mathcal{G}, \mathcal{X}_n)$. The *term homomorphism* $h\colon \mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{G})$ determined by $h_{\mathcal{F}}$ is defined as follows:

$$h(f(t_1, \ldots, t_n)) = t_f\,\{x_1 \mapsto h(t_1), \ldots, x_n \mapsto h(t_n)\}$$

The term homomorphism h is called *symbol-to-symbol* if, for each $f \in \mathcal{F}, t_f = g(y_1, \ldots, y_m)$ with $g \in \mathcal{G}, y_1, \ldots, y_m \in \mathcal{X}_n$, i.e. each t_f is a context of height 1. It is called *ε-free* if, for each $f \in \mathcal{F}, t_f \notin \mathcal{X}_n$, i.e. each t_f is a context of height at least 1.

Applying the *placeholders-via-naturality* principle of Hasuo et al. [5], term homomorphisms are captured by the following type:

```
type TermHom f g = ∀ a . f a → Context g a
```

As we did for other functions on terms, we can generalise the application of term homomorphism uniformly to contexts:

[6] Actually, Thatcher [21] calls them "tree homomorphisms". But we prefer the notion "term" over "tree" in our context.

```
termHom :: (Functor f, Functor g)
    ⇒ TermHom f g → Cxt h f a → Cxt h g a
termHom f (Term t) =
    appCxt (f (fmap (termHom f) t))
termHom _ (Hole b) = Hole b
```

The use of explicit pattern matching in lieu of defining the function as a free algebra homomorphism *free* (*appCxt* . *f*) *Hole* is essential in order to obtain this general type. In particular, the use of the proper GADT constructor *Hole*, which has result type *Context g a*, makes this necessary.

Of course, the polymorphic type of term homomorphisms restricts the class of functions that can be defined in this way. It can be considered as a special form of term algebra: *appCxt* . *f* is the term algebra corresponding to the term homomorphism *f*. But not every catamorphism is also a term homomorphism. For certain term algebras we actually need to inspect the arguments of the functor instead of only shuffling them around. For example, we cannot hope to define the evaluation function *eval* as a term homomorphism.

Some catamorphisms, however, can be represented as term homomorphisms, e.g. the desugaring function *desug*:

```
class (Functor f, Functor g) ⇒ Desug f g where
    desugHom :: TermHom f g
```

Lifting term homomorphisms to sums is standard. The instances for the functors that do not need to be desugared can be implemented by turning a single functor application to a context of height 1, and using overlapping instances:

```
simpCxt :: Functor f ⇒ f a → Context f a
simpCxt = Term . fmap Hole

instance (f :≺: g, Functor g) ⇒ Desug f g where
    desugHom = simpCxt . inj
```

Turning to the instance for *Sug*, we can see why a term homomorphism suffices for implementing *desug*. In the original catamorphic definition, we had for example

```
desugAlg (Neg x) = iConst (−1) `iMult` x
```

Here we only need to copy the argument *x* of the constructor *Neg* and define the appropriate context around it. This definition can be copied almost verbatim for the term homomorphism:

```
desugHom (Neg x) = iConst (−1) `iMult` Hole x
```

We only need to embed the *x* as a hole. The same also applies to the other defining equation. In order to make the definitions more readable we add a convenience function to the class *Desug*:

```
class (Functor f, Functor g) ⇒ Desug f g where
    desugHom :: TermHom f g
    desugHom = desugHom′ . fmap Hole
    desugHom′ :: Alg f (Context g a)
    desugHom′ x = appCxt (desugHom x)
```

Now we can actually copy the catamorphic definition one-to-one:

```
instance (Op :≺: f, Val :≺: f, Functor f)
    ⇒ Desug Sug f where
    desugHom′ (Neg x)  = iConst (−1) `iMult` x
    desugHom′ (Swap x) = iSnd x `iPair` iFst x
```

In the next two sections, we will show what we actually gain by adopting the term homomorphism approach. We will reconsider and address the issues that we identified in Sections 4.1 and 4.2.

4.4.1 Propagating Annotations through Term Homomorphisms

The goal is now to take advantage of the structure of term homomorphisms in order to automatically propagate annotations. This boils down to transforming a function of type *TermHom f g* to a function of type *TermHom* (*f* :&: *p*) (*g* :&: *p*). In order to do this, we need a function that is able to annotate a context with a fixed annotation. Such a function is in fact itself a term homomorphism:

```
ann :: Functor f ⇒ p → Cxt h f a → Cxt h (f :&: p) a
ann p = termHom (simpCxt . (:&:p))
```

To be more precise, this function is a *symbol-to-symbol* term homomorphism—(:&:*p*) is of type ∀ *a* . *f a* → (*f* :&: *p*) *a*—that maps each constructor to exactly one constructor. The composition with *simpCxt* lifts it to the type of general term homomorphisms.

The propagation of annotations is now simple:

```
propAnn :: Functor g ⇒ TermHom f g
                     → TermHom (f :&: p) (g :&: p)
propAnn f (t :&: p) = ann p (f t)
```

The annotation of the current subterm is propagated to the context created by the original term homomorphism.

This definition can now be generalised—as we did in Section 3.3—such that it can also deal with annotations that have been distributed over a sum of signatures. Unfortunately, the type class *RemA* that we introduced for dealing with such distributed annotations is not enough for this setting as we need to extract and inject annotations now:

```
class DistAnn f p f′ | f′ → f, f′ → p where
    injectA :: p → f a → f′ a
    projectA :: f′ a → (f a, p)
```

An instance of *DistAnn f p f′* indicates that signature *f′* is a variant of *f* annotated with values of type *p*. The relevant instances are straightforward:

```
instance DistAnn f p (f :&: p) where
    injectA c v = v :&: c
    projectA (v :&: p) = (v, p)

instance DistAnn f p f′
    ⇒ DistAnn (g :+: f) p ((g :&: p) :+: f′) where
    injectA c (Inl v) = Inl (v :&: c)
    injectA c (Inr v) = Inr (injectA c v)

    projectA (Inl (v :&: p)) = (Inl v, p)
    projectA (Inr v)         = let (v′, p) = projectA v
                               in (Inr v′, p)
```

We can then make use of this infrastructure in the definition of *ann* and *propAnn*:

```
ann :: (DistAnn f p g, Functor f, Functor g)
    ⇒ p → Cxt h f a → Cxt h g a
ann p = termHom (simpCxt . injectA p)

propAnn :: (DistAnn f p f′, DistAnn g p g′, Functor g,
    Functor g′) ⇒ TermHom f g → TermHom f′ g′
propAnn f t′ = let (t, p) = projectA t′ in ann p (f t)
```

We can now use *propAnn* to propagate source position information from a full AST to its desugared version:

```
type SigP′ = Sug :&: Pos :+: SigP

desugHom′ :: TermHom SigP′ SigP
desugHom′ = propAnn desugHom
```

4.4.2 Composing Term Homomorphisms

Another benefit of the function type of term homomorphisms over term algebras is the simple fact that its domain *f a* is independent of the target signature *g*:

```
type TermHom f g = ∀ a . f a → Context g a
```

This enables us to compose term homomorphisms:

```
(⊚) :: (Functor g, Functor h) ⇒
    TermHom g h → TermHom f g → TermHom f h
f ⊚ g = termHom f . g
```

Here we make use of the fact that *termHom* also allows us to apply a term homomorphism to a proper context—*termHom f* has type ∀ *a* . *Context g a* → *Context h a*.

Although the occurrence of the target signature in the domain of term algebras prevents them from being composed with each other, the composition with a term homomorphism is still possible:

```
(⊡) :: Functor g ⇒ Alg g a → TermHom f g → Alg f a
alg ⊡ talg = free alg id . talg
```

The ability to compose term homomorphisms with term algebras or other term homomorphisms allows us to perform program transformations in the vein of short-cut fusion [4]. For an example, recall that we have extended the evaluation to terms over Sig' by precomposing the evaluation function with the desugaring function:

```
eval′ :: Term Sig′ → Term Val
eval′ = eval . desug
```

The same can be achieved by composing on the level of algebras respectively term homomorphisms instead of the level of functions:

```
eval′ :: Term Sig′ → Term Val
eval′ = cata (evalAlg ⊡ desugHom)
```

Using the rewrite mechanism of GHC [7], we can make this optimisation automatic, by including the following rewrite rule:

```
"cata/termHom" ∀ (a :: Alg g d) (h :: TermHom f g) x .
    cata a (termHom h x) = cata (a ⊡ h) x
```

One can easily show that this transformation is sound. Moreover, a similar rule can be devised for composing two term homomorphisms. The run-time benefits of these optimisation rules are considerable as we will see in Section 6.2.

4.4.3 Monadic Term Homomorphisms

Like catamorphisms, we can also easily lift term homomorphisms to monadic computations. We only need to lift the computations to a monadic type and use $mapM$ instead of $fmap$ for the recursion respectively use monadic function composition ⋘ instead of pure function composition:

```
type TermHomM m f g = ∀ a . f a → m (Context g a)

termHomM :: (Traversable f, Functor g, Monad m)
    ⇒ TermHomM m f g → Cxt h f a → m (Cxt h g a)
termHomM f (Term t) =
    liftM appCxt . f ⋘ mapM (termHomM f) t
termHomM _ (Hole b)  = return (Hole b)
```

The same strategy yields monadic variants of ⊚ and ⊡

```
(⊚̂) :: (Traversable g, Functor h, Monad m) ⇒
    TermHomM m g h → TermHomM m f g
                         → TermHomM m f h
f ⊚̂ g = termHomM f ⋘ g

(⊡̂) :: (Traversable g, Monad m) ⇒
    AlgM m g a → TermHomM m f g → AlgM m f a
alg ⊡̂ talg = freeM alg return ⋘ talg
```

In contrast to pure term homomorphisms, one has to be careful when applying these composition operators: The fusion equation

```
termHomM (f ⊚̂ g) = termHomM f ⋘ termHomM g
```

does not hold in general! However, Fokkinga [3] showed that for monads satisfying a certain distributivity law, the above equation indeed holds. An example of such a monad is the $Maybe$ monad. Furthermore, the equation is also true whenever one of the term homomorphisms is in fact pure, i.e. of the form $return\ .\ h$ for a non-monadic term homomorphism h. The same also applies to the fusion equation for ⊡̂. Nevertheless, it is still possible to devise rewrite rules that perform short-cut fusion under these restrictions.

An example of a monadic term homomorphism is the following function that recursively coerces a term to a sub-signature:

```
deepProject :: (Functor g, Traversable f, g :≺: f)
                ⇒ Term f → Maybe (Term g)
deepProject = termHomM (liftM simpCxt . proj)
```

As $proj$ is, in fact, a monadic *symbol-to-symbol* term homomorphism we have to compose it with $simpCxt$ to obtain a general monadic term homomorphism.

4.5 Beyond Catamorphisms

So far we have only considered (monadic) algebras and their (monadic) catamorphisms. It is straightforward to implement the machinery for programming in coalgebras and their anamorphisms:

```
type Coalg f a = a → f a

ana :: Functor f ⇒ Coalg f a → a → Term f
ana f x = Term (fmap (ana f) (f x))
```

In fact, also more advanced recursion schemes can be accounted for in our framework: This includes paramorphisms and histomorphisms as well as their dual notions of apomorphisms and futumorphisms [23]. Similarly, monadic variants of these recursion schemes can be derived using the type class $Traversable$.

As an example of the abovementioned recursion schemes, we want to single out *futumorphisms*, as they can be represented conveniently using contexts and in fact are more natural to program than run-of-the-mill anamorphisms. The algebraic counterpart of futumorphisms are *cv-coalgebras* [23]. In their original algebraic definition they look rather cumbersome (cf. [23, Ch. 4.3]). If we implement cv-coalgebras in Haskell using contexts, the computation they denote becomes clear immediately:

```
type CVCoalg f a = a → f (Context f a)
```

Anamorphisms only allow us to construct the target term one layer at a time. This can be plainly seen from the type $a \to f\ a$ of coalgebras. Futumorphisms on the other hand allow us to construct an arbitrary large part of the target term. Instead of only producing a single application of a constructor, cv-coalgebras produce a *non-empty* context, i.e. a context of height at least 1. The non-emptiness of the produced contexts guarantees that the resulting futumorphism is *productive*.

For the sake of brevity, we lift this restriction to non-empty contexts and consider *generalised cv-coalgebras*:

```
type CVCoalg f a = a → Context f a
```

Constructing the corresponding futumorphism is simple and almost the same as for anamorphisms:

```
futu :: Functor f ⇒ CVCoalg f a → a → Term f
futu f x = appCxt (fmap (futu f) (f x))
```

Generalised cv-coalgebras also occur when composing a coalgebra and a term homomorphism, which can be implemented by plain function composition:

```
compCoa :: TermHom f g → Coalg f a → CVCoalg g a
compCoa hom coa = hom . coa
```

This can then be lifted to the composition of a generalised cv-coalgebra and a term homomorphism, by running the term homomorphism:

```
compCVCoalg :: (Functor f, Functor g)
    ⇒ TermHom f g → CVCoalg f a → CVCoalg g a
compCVCoalg hom coa = termHom hom . coa
```

With generalised cv-coalgebras one has to be careful, though, as they might not be productive. However, the above constructions can be replicated with ordinary cv-coalgebras. Instead of general term homomorphisms, we have to restrict ourselves to *ε-free* term homomorphisms [2] which are captured by the type:

```
type TermHom′ f g = ∀ a . f a → g (Context g a)
```

This illustrates that with the help of contexts, (generalised) futumorphisms provide a much more natural coalgebraic programming model than anamorphisms.

5. Mutually Recursive Data Types and GADTs

Up to this point we have only considered the setting of a single recursively defined data type. We argue that this is the most common setting in the area we are targeting, viz. processing and analysing abstract syntax trees. Sometimes it is, however, convenient to encode certain invariants of the data structure, e.g. well-typing of ASTs, as mutually recursive data types or GADTs. In this section, we will show how this can be encoded as a family of compositional data types by transferring the construction of Johann and Ghani [6] to compositional data types.

Recall that the idea of representing recursive data types as fixed points of functors is to abstract from the recursive reference to the data type that should be defined. Instead of a recursive data type

```
data Exp   = ··· | Mult Exp Exp | Fst Exp
```

we define a functor

```
data Sig e = ··· | Mult e   e   | Fst e
```

The trick for defining mutually recursive data types is to use phantom types as labels that indicate which data type we are currently in. As an example, reconsider our simple expression language over integers and pairs. But now we define them in a family of two mutually recursive data types in order to encode the expected invariants of the expression language, e.g. the sum of two integers yields an integer:

```
data IExp = Const Int | Mult IExp IExp
          | Fst PExp | Snd PExp
data PExp = Pair IExp IExp
```

We can encode this on signatures by adding an additional type argument which indicates the data types we are expecting as arguments to the constructors:

```
data Pair
data ISig e l = Const Int | Mult (e Int) (e Int)
              | Fst (e Pair) | Snd (e Pair)
data PSig e l = Pair (e Int) (e Int)
```

Notice that the type variable e that is inserted in lieu of recursion is now of kind $* \to *$ as we consider a family of types. The "label type"—*Int* respectively *Pair*—then selects the desired type from this family. The definitions above, however, only indicate which data type we are expecting, e.g. *Mult* expects two integer expressions and *Swap* a pair expression. In order to also label the result type accordingly, we rather want to define *ISig* and *PSig* as

```
data ISig e Int  = ...
data PSig e Pair = ...
```

Using GADTs we can do this, although in a syntactically more verbose way:

```
data ISig e l where
  Const    ::               Int    → ISig e Int
  Mult     :: e Int → e Int    → ISig e Int
  Fst, Snd ::               e Pair → ISig e Int
data PSig e l where
  Pair :: e Int → e Int → PSig e Pair
```

Notice that signatures are not functors of kind $* \to *$ anymore. Instead, they have the kind $(* \to *) \to (* \to *)$, thus adding one level of indirection.

Following previous work [6, 16], we can formulate the actual recursive definition of terms as follows:

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

The first argument f is a signature, i.e. has the kind $(* \to *) \to (* \to *)$. The type constructor *Term* recursively applies the signature f while propagating the label l according to the signature. Note that *Term* f is of kind $* \to *$. A value of type *Term* f l is a mutually recursive data structure with topmost label l. In the recursive definition, *Term* f is applied to a signature f, i.e. in the case of f being *ISig* or *PSig* it instantiates the type variable e in their respective definitions. The type signatures of *ISig* and *PSig* can thus be read as propagation rules for the labels: For example, *Fst* takes a term with top-level labeling *Pair* and returns a term with top-level labeling *Int*.

5.1 Higher-Order Functors

It is important to realise that the transition to a family of mutually recursive data types amounts to nothing more than adding a layer of indirection. A signature, which has previously been a functor, is now a (generalised) *higher-order functor* [6]:

```
type f →̇ g = ∀ a . f a → g a

class HFunctor h where
  hfmap :: f →̇ g → h f →̇ h g

instance HFunctor ISig where
  hfmap _ (Const i)  = Const i
  hfmap f (Mult x y) = Mult (f x) (f y)
  hfmap f (Fst x)    = Fst (f x)
```

The function *hfmap* witnesses that a natural transformation $f \mathrel{\dot\to} g$ from functor f to functor g is mapped to a natural transformation $h\ f \mathrel{\dot\to} h\ g$.

Observe the simplicity of the pattern that we used to lift our representation of compositional data types to mutually recursive types: Replace functors with higher-order functors, and instead of the function space $\to$ consider the natural transformation space $\dot\to$. This simple pattern will turn out to be sufficient in order to lift most of the concepts of compositional data types to mutually recursive data types. Sums and injections can thus be represented as follows:

```
data (f :+: g) (a :: * → *) l = Inl (f a l) | Inr (g a l)

type NatM m f g = ∀ i . f i → m (g i)

class (sub :: (* → *) → * → *) :≺: sup where
  inj  :: sub a →̇ sup a
  proj :: NatM Maybe (sup a) (sub a)
```

Lifting *HFunctor* instances to sums works in the same way as we have seen for *Functor*. The same goes for instances of :≺:.

With the summation :+: in place we can define the family of data types that defines integer and pair expressions:

```
type Expr = Term (ISig :+: PSig)
```

This is indeed a family of types. We obtain the type of integer expressions with *Expr Int* and the type of pair expressions as *Expr Pair*.

5.2 Representing GADTs

Before we continue with lifting recursion schemes such as catamorphisms to the higher-order setting, we reconsider our example of mutually recursive data types. In contrast to the representation using a single recursive data type, the definition of *IExp* and *PExp* does not allow nested pairs—pairs are always built from integer expressions. The same goes for *Expr Int* and *Expr Pair*, respectively. This restriction is easily lifted by using a GADT instead:

```
data SExp l where
  Const ::                 Int        → SExp Int
  Mult  :: SExp Int → SExp Int   → SExp Int
  Fst   ::                 SExp (s, t) → SExp s
  Snd   ::                 SExp (s, t) → SExp t
  Pair  :: SExp s   → SExp t     → SExp (s, t)
```

This standard GADT representation can be mapped directly to our signature definitions. However, instead of defining a single GADT, we proceed as we did with non-mutually recursive compositional data types. We split the signature into values and operations:

```
data Val e l where
  Const ::         Int → Val e Int
  Pair  :: e s → e t → Val e (s, t)
data Op e l where
  Mult :: e Int → e Int  → Op e Int
  Fst  ::         e (s, t) → Op e s
  Snd  ::         e (s, t) → Op e t
type Sig = Op :+: Val
```

Combining the above two signatures then yields the desired family of mutually recursive data types *Term Sig* ≅ *SExp*.

This shows that the transition to higher-order functors also allows us to naturally represent GADTs in a modular fashion.

5.3 Recursion Schemes

We shall continue to apply the pattern for shifting to mutually recursive data types: Replace *Functor* with *HFunctor* and function space → with the space of natural transformations $\dot{\to}$. Take, for example, algebras and catamorphisms:

```
type Alg f a = f a →̇ a

cata :: HFunctor f ⇒ Alg f a → Term f →̇ a
cata f (Term t) = f (hfmap (cata f) t)
```

Now, an algebra has a family of types $a :: * \to *$ as carrier. That is, we have to move from algebras to *many-sorted* algebras. Representing many-sorted algebras comes quite natural in most cases. For example, the evaluation algebra class can be recast as a many-sorted algebra class as follows:

```
class Eval e v where
  evalAlg :: Alg e (Term v)

eval :: (HFunctor e, Eval e v) ⇒ Term e →̇ Term v
eval = cata evalAlg
```

Here, we can make use of the fact that $Term\ v$ is in fact a family of types and can thus be used as a carrier of a many-sorted algebra.

Except for the slightly more precise type of *projC* and *projP*, the definition of *Eval* is syntactically equal to its non-mutually recursive original from Section 2.1:

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

instance (Val :≺: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst $ projC x ∗ projC y
  evalAlg (Fst x)     = fst $ projP x
  evalAlg (Snd x)     = snd $ projP x

projC :: (Val :≺: v) ⇒ Term v Int → Int
projC v = case project v of Just (Const n) → n

projP :: (Val :≺: v) ⇒ Term v (s, t)
                     → (Term v s, Term v t)
projP v = case project v of Just (Pair x y) → (x, y)
```

In some cases, it might be a bit more cumbersome to define and use the carrier of a many-sorted algebra. However, most cases are well-behaved and we can use the family of terms $Term\ f$ as above or alternatively the identity respectively the constant functor:

```
data I a   = I {unI :: a}
data K a b = K {unK :: a}
```

For example, a many-sorted algebra class to evaluate expressions directly into Haskell values of the corresponding types can be defined as follows:

```
class EvalI f where
  evalAlgI :: Alg f I

evalI :: (EvalI f, HFunctor f) ⇒ Term f t → t
evalI = unI . cata evalAlgI
```

The lifting of other recursion schemes whether algebraic or coalgebraic can be achieved in the same way as illustrated for catamorphisms above. The necessary changes are again quite simple. Similarly to the type class *HFunctor*, we can obtain lifted versions of *Foldable* and *Traversable* which can then be used to implement generic programming techniques and to perform monadic computations, respectively. The generalisation of terms to contexts and the corresponding notion of term homomorphisms is also straightforward. The same short-cut fusion rules that we have considered for simple compositional data types can be implemented without any surprises as well.

The only real issue worth mentioning is that the generic querying combinator *query* needs to produce result values of a fixed type as opposed to a family of types. The propagation of types defined by GADTs cannot be captured by the simple pattern of the querying combinator. Thus, the querying combinator is typed as follows:

```
query :: HFoldable f ⇒ (∀ i . Term f i → r)
         → (r → r → r) → Term f i → r
```

For the *subs* combinator, which produces a list of all subterms, the issue is similar: $Term\ f$ is a type family, thus $[\,Term\ f\,]$ is not a valid type. However, we can obtain the desired type of list of terms by existentially quantifying over the index type using the GADT

```
data A f = ∀ i . A (f i)
```

The type of *subs* can now be stated as follows:

```
subs :: HFoldable f ⇒ Term f i → [A (Term f)]
```

6. Practical Considerations

Besides showing the expressiveness and usefulness of the framework of compositional data types, we also want to showcase its practical applicability as a software development tool. To this end, we consider aspects of *usability* and *performance impacts* as well.

6.1 Generating Boilerplate Code

The implementation of recursion schemes depends on the signatures being instances of the type class *Functor*. For generic programming techniques and monadic computations, we rely on the type classes *Foldable* and *Traversable*, respectively. Additionally, higher-order functors necessitate a set of lifted variants of the abovementioned type classes. That is a lot of boilerplate code! Writing and maintaining this code would almost entirely defeat the advantage of using compositional data types in the first place.

Luckily, by leveraging Template Haskell [18], instance declarations of all generic type classes that we have mentioned in this paper can be generated automatically at compile time similar to Haskell's **deriving** mechanism. Even though some Haskell packages such as *derive* already provide automatically derived instances for some of the standard classes like *Functor*, *Foldable* and *Traversable*, we chose to implement the instance generators for these as well. The heavy use of the methods of these classes for implementing recursion schemes means that they contribute considerably to the computational overhead! Automatically deriving instance declarations with carefully optimised implementations of each of the class methods, have proven to yield substantial run-time improvements, especially for monadic computations.

We already mentioned that we assume with each constructor

$$Constr :: t_1 \to \cdots \to t_n \to f\ a$$

of a signature f, a smart constructor defined by

$$iConstr :: f :\prec: g \Rightarrow s_1 \to \cdots \to s_n \to Term\ g$$
$$iConstr\ x_1 \ldots x_n = inject\ \$\ Constr\ x_1 \ldots x_n$$

where the types s_i are the same as t_i except with occurrences of the type variable a replaced by $Term\ g$. These smart constructors can be easily generated automatically using Template Haskell.

Another issue is the declaration of instances of type classes *Eq*, *Ord* and *Show* for types of the form $Term\ f$. This can be achieved by lifting these type classes to functors, e.g. for *Eq*:

```
class EqF f where
  eqF :: Eq a ⇒ f a → f a → Bool
```

From instances of this class, corresponding instances of *Eq* for terms and contexts can be derived:

```
instance (EqF f, Eq a) ⇒ Eq (Cxt h f a) where
  (≡) (Term t1) (Term t2) = t1 `eqF` t2
  (≡) (Hole h1) (Hole h2) = h1 ≡ h2
  (≡) _         _         = False
```

Instances of *EqF*, *OrdF* and *ShowF* can be derived straightforwardly using Template Haskell which then yield corresponding instances of *Eq*, *Ord* and *Show* for terms and contexts. The thus obtained instances are equivalent to the ones obtained from Haskell's **deriving** mechanism on corresponding recursive data types.

Figure 1 demonstrates the *complete* source code needed in order to implement some of the earlier examples in our library.

```
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 | Snd e
data Sug e = Neg e | Swap e
type Sig   = Op :+: Val
type Sig'  = Sug :+: Sig

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

-- * Term Evaluation
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) = iConst $ projC x * projC y
  evalAlg (Fst x)    = fst $ projP x
  evalAlg (Snd x)    = snd $ projP x

projC :: (Val :<: v) ⇒ Term v → Int
projC v = case project v of Just (Const n) → n

projP :: (Val :<: v) ⇒ Term v → (Term v, Term v)
projP v = case project v of Just (Pair x y) → (x,y)

-- * Desugaring
instance (Op :<: f, Val :<: f, Functor f) ⇒ Desugar Sug f where
  desugHom' (Neg x)  = iConst (-1) `iMult` x
  desugHom' (Swap x) = iSnd x `iPair` iFst x

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

Figure 1. Example usage of the compositional data types library.

6.2 Performance Impact

In order to minimise the overhead of the recursion schemes, we applied some simple optimisations to the implementation of the recursion schemes themselves. For example, $cata$ is defined as

$$\begin{aligned}&cata :: \forall f\ a\ .\ Functor\ f \Rightarrow Alg\ f\ a \rightarrow Term\ f \rightarrow a\\&cata\ f = run\\&\quad \textbf{where}\ run :: Term\ f \rightarrow a\\&\qquad\quad run\ (Term\ t) = f\ (fmap\ run\ t)\end{aligned}$$

The biggest speedup, however, can be obtained by providing automatically generated, carefully optimised implementations for each method of the type classes *Foldable* and *Traversable*.

In order to gain speedup in the implementation of generic programming combinators, we applied the same techniques as Mitchell and Runciman [12] by leveraging short-cut fusion [4] via *build*. The *subs* combinator is thus defined as:

$$\begin{aligned}&subs :: \forall f\ .\ Foldable\ f \Rightarrow Term\ f \rightarrow [Term\ f]\\&subs\ t = build\ (f\ t)\ \textbf{where}\\&\quad f :: Term\ f \rightarrow (Term\ f \rightarrow b \rightarrow b) \rightarrow b \rightarrow b\\&\quad f\ t\ cons\ nil = t\ `cons`\\&\qquad foldl\ (\lambda u\ s \rightarrow f\ s\ cons\ u)\ nil\ (unTerm\ t)\end{aligned}$$

Instead of building the result list directly, we use the *build* combinator which then can be eliminated if combined with a consumer such as a fold or a list comprehension.

Table 1 shows the run-time performance of our framework for various functions dealing with ASTs: Desugaring (*desug*), type inference (*infer*), expression evaluation (*eval*), and listing respectively searching for free variables (*freeVars*, *contVar*). The *Hom* and *Cata* version of *desug* differ in that the former is defined as a term homomorphism, the latter as a catamorphism. For *eval* and *infer*, the suffix *Desug* indicates that the computation is prefixed by a desugaring phase (using *desugHom*), the suffix *M* indicates monadic variants (for error handling), and *Direct* indicates that the function was implemented not as a catamorphism but using explicit recursion. The numbers in the table are multiples of the run-time of an implementation using ordinary algebraic data types and recursion. The numbers in parentheses indicate the run-time factor if the automatic short-cut fusion described in Section 4.4.2 is disabled. Each function is tested on three different inputs of increasing size. The first is a hand-written "natural" expression consisting of 16 nodes. The other two expressions are randomly generated expressions of depth 10 and 20, respectively, which corresponds to approximately 800 respectively 200,000 nodes. This should reveal how the overhead of our framework scales. The benchmarks were performed with the *criterion* framework using GHC 7.0.2 with optimisation flag `-O2`.

Function	hand-written	random (10)	random (20)
desugHom	$3.6 \cdot 10^{-1}$	$5.0 \cdot 10^{-3}$	$6.1 \cdot 10^{-6}$
desugCata	$1.8 \cdot 10^{-1}$	$4.41 \cdot 10^{-3}$	$5.3 \cdot 10^{-6}$
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
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

Table 1. Run-time of functions on compositional data types (as multiples of the run-time of an implementation using ordinary algebraic data types).

As a pleasant surprise, we observe that the penalty of using compositional data types is comparatively low. It is in the same ballpark as for generic programming libraries [12, 15]. For some functions we even obtain a speedup! The biggest surprise is, however, the massive speedup gained by the desugaring function. In both its catamorphic and term-homomorphic version, it seems to perform asymptotically better than the classic implementation, yielding a speedup of over five orders of magnitude. We were also surprised to see that (except for one case) functions programmed as catamorphisms outperformed functions using explicit recursion! In fact, with GHC 6.12, the situation was reversed.

Moreover, we observe that the short-cut fusion rules implemented in our framework uniformly yield a considerable speedup of up to factor five. As a setback, however, we have to recognise that implementing desugaring as a term homomorphism yields a slowdown of factor up to two compared to its catamorphic version.

Finally, we compared our implementation of generic programming techniques with Uniplate [12], one of the top-performing generic programming libraries. In particular, we looked at its *universe* combinator which computes the list of all subexpressions. We have implemented this combinator in our framework as subs. In Table 1, our implementation is indicated by the suffix C, the Uniplate implementation, working on ordinary algebraic data types, is indicated by U. We can see that we are able to obtain comparable performance in all cases.

7. Discussion

Starting from Swierstra's *Data types à la carte* [20], we have constructed a framework for representing data types in a compositional fashion that is readily usable for practical applications. Our biggest contribution is the generalisation of terms to contexts which allow us to capture the notion of term homomorphisms. Term homomorphisms provide a rich structure that allows flexible reuse and enables simple but effective optimisation techniques. Moreover, term homomorphisms can be easily extended with a state. Depending on how the state is propagated, this yields bottom-up respectively top-down tree transducers [2]. The techniques for short-cut fusion and propagation of annotations can be easily adapted.

7.1 Related Work

The definition of monadic catamorphisms that we use goes back to Fokkinga [3]. He only considers monads satisfying a certain distributivity law. However, this distributivity is only needed for the fusion rules of Section 4.4.3 to be valid. Steenbergen et al. [22] use the same approach to implement catamorphisms with errors. In contrast, Visser and Löh [24] consider monadic catamorphism for which the monadic effect is part of the term structure.

The construction to add annotations to functors is also employed by Steenbergen et al. [22] to add detailed source position annotations to ASTs. However, since they are considering general catamorphisms, they are not able to provide a means to propagate annotations. Moreover, since Steenbergen et al. do not account for sums of functors, the distribution of annotations over sums is not an issue for them. Visser and Löh [24] consider a more general form of annotations via arbitrary *functor transformations*. Unfortunately, this generality prohibits the automatic propagation of annotations as well as their distribution over sums.

Methods to represent mutually recursive data types as fixed points of (regular) functors have been explored to some extent [1, 8, 16, 19]. All of these techniques are limited to mutually recursive data types in which the number of nested data types is limited up front and are thus not compositional. However, in the representation of Yakushev et al. [16], the restriction to mutually recursive data types with a closed set of constituent data types was implemented intentionally. Our representation simply removes these restrictions which would in fact add no benefit in our setting. The resulting notion of higher-order functors that we considered was also used by Johann and Ghani [6] in order to represent GADTs.

7.2 Future Work

There are a number of aspects that are still missing which should be the subject of future work: As we have indicated, the restriction of the subtyping class $:\prec:$ hinders full compositionality of signature summation $:+:$. A remedy could be provided with a richer type system as proposed by Yorgey [26]. This would also allow us to define the right-distributivity of annotations :&: over sums $:+:$ more directly by a type family. Alternatively, this issue can be addressed with type instance-chains as proposed by Morris and Jones [13]. Another issue of Swierstra's original work is the $\mathit{project}$ function which allows us to inspect terms ad-hoc. Unfortunately, it does not allow us to give a complete case analysis. In order to provide this, we need a function of type

$$\begin{aligned}(f :\prec: g) \Rightarrow \mathit{Term}\ g \rightarrow \mathit{Either}\ (&\ f\ (\mathit{Term}\ g))\\ &((g :-: f)\ (\mathit{Term}\ g))\end{aligned}$$

which allows us to match against the "remainder signature" $g :-: f$.

References

[1] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[2] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. Draft.

[3] M. Fokkinga. Monadic Maps and Folds for Arbitrary Datatypes. Technical report, Memoranda Informatica, University of Twente, 1994.

[4] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA '93*, pages 223–232. ACM, 1993.

[5] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical Views on Computations on Trees (Extended Abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *LNCS*, pages 619–630. Springer, 2007.

[6] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL '08*, pages 297–308. ACM, 2008.

[7] S. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop '01*, pages 203–233, 2001.

[8] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3):255 – 279, 1990.

[9] S. Marlow. Haskell 2010 Language Report, 2010.

[10] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

[11] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA '91*, pages 124–144. ACM, 1991.

[12] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Haskell Workshop '07*, pages 49–60. ACM, 2007.

[13] J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10*, pages 375–386, 2010.

[14] A. Rodriguez Yakushev, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell Workshop '08*, pages 111–122. ACM, 2008.

[15] A. Rodriguez Yakushev, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. Technical report, Department of Information and Computing Sciences, Utrecht University, 2008.

[16] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09*, pages 233–244. ACM, 2009.

[17] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP '09*, pages 341–352. ACM, 2009.

[18] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell Workshop '02*, pages 1–16. ACM, 2002.

[19] S. Swierstra, P. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In S. Swierstra, J. Oliveira, and P. Henriques, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer, 1999.

[20] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[21] J. W. Thatcher. *Tree automata: an informal survey*, chapter 4, pages 143–178. Prentice Hall, 1973.

[22] M. Van Steenbergen, J. P. Magalhães, and J. Jeuring. Generic selections of subexpressions. In *WGP '10*, pages 37–48. ACM, 2010.

[23] V. Vene. *Categorical programming with inductive and coinductive types*. Phd thesis, University of Tartu, Estonia, 2000.

[24] S. Visser and A. Löh. Generic storage in Haskell. In *WGP '10*, pages 25–36. ACM, 2010.

[25] P. Wadler. The Expression Problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[26] B. Yorgey. Typed type-level functional programming in GHC. Talk at Haskell Implementors Workshop, 2010.