Parametric Compositional Data Types

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

In previous work we have illustrated the benefits that *compositional data types* (CDTs) offer for implementing languages and in general for dealing with abstract syntax trees (ASTs). Based on Swierstra's *data types à la carte*, CDTs are implemented as a Haskell library that enables the definition of recursive data types and functions on them in a modular and extendable fashion. Although CDTs provide a powerful tool for analysing and manipulating ASTs, they lack a convenient representation of variable binders. In this paper we remedy this deficiency by combining the framework of CDTs with Chlipala's parametric higher-order abstract syntax (PHOAS). We show how a generalisation from functors to difunctors enables us to capture PHOAS while still maintaining the features of the original implementation of CDTs, in particular its modularity. Unlike previous approaches, we avoid so-called *exotic terms* without resorting to abstract types: this is crucial when we want to perform *transformations* on CDTs that inspect the recursively computed CDTs, e.g. constant folding.

1 Introduction

When implementing domain-specific languages (DSLs)—either as embedded languages or stand-alone languages—the abstract syntax trees (ASTs) of programs are usually represented as elements of a recursive algebraic data type. These ASTs typically undergo various transformation steps, such as desugaring from a full language to a core language. But reflecting the invariants of these transformations in the type system of the host language can be problematic. For instance, in order to reflect a desugaring transformation in the type system, we must define a separate data type for ASTs of the core language. Unfortunately, this has the side effect that common functionality, such as pretty printing, has to be duplicated.

Wadler identified the essence of this issue 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" [24]. Swierstra [22] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits invaluable simplicity and clarity, it lacks features necessary to apply it in a practical setting beyond the confined simplicity of the expression problem. To this end, the framework of *compositional data types* (CDTs) [4] provides a rich library for implementing practical functionality on highly modular data types. This includes support of a wide array of recursion schemes in both pure and monadic forms, as well as mutually recursive data types and generalised algebraic data types (GADTs) [18].

What CDTs fail to address, however, is a transparent representation of variable binders that frees the programmer's mind from common issues like computations modulo α-equivalence and capture-avoiding substitutions. The work we present in this paper fills that gap by adopting (a restricted form of) higher-order abstract syntax (HOAS) [15], which uses the host language's variable binding mechanism to represent binders in the object language. Since implementing efficient recursion schemes in the presence of HOAS is challenging [8, 13, 19, 25], integrating this technique with CDTs is a non-trivial task.

Following a brief introduction to CDTs in Section 2, we describe how to achieve this integration as follows:

J. Chapman and P. B. Levy (Eds.): Fourth Workshop on
Mathematically Structured Functional Programming (MSFP 2012).
EPTCS 76, 2012, pp. 3–24, doi:10.4204/EPTCS.76.3

- We adopt parametric higher-order abstract syntax (PHOAS) [6], and we show how to capture this restricted form of HOAS via difunctors. The thus obtained *parametric compositional data types* (PCDTs) allow for the definition of modular catamorphisms à la Fegaras and Sheard [8] in the presence of binders. Unlike previous approaches, our technique does not rely on abstract types, which is crucial for modular computations that are also modular in their result type (Section 3).
- We illustrate why monadic computations constitute a challenge in the parametric setting and we show how monadic catamorphisms can still be defined for a restricted class of PCDTs (Section 4).
- We show how to transfer the restricted recursion scheme of *term homomorphisms* [4] to PCDTs. Term homomorphisms enable the same flexibility for reuse and opportunity for deforestation [23] that we know from CDTs (Section 5).
- We show how to represent mutually recursive data types and GADTs by generalising PCDTs in the style of Johann and Ghani [10] (Section 6).
- We illustrate the practical applicability of our framework by means of a complete library example, and we show how to automatically derive functionality for deciding equality (Section 7).

Parametric compositional data types are available as a Haskell library[1], including numerous examples that are not included in this paper. All code fragments presented throughout the paper are written in (literate) Haskell [11], and the library relies on several language extensions that are currently only known to be supported by the Glasgow Haskell Compiler (GHC).

2 Compositional Data Types

Based on Swierstra's *data types à la carte* [22], compositional data types (CDTs) [4] provide a framework for manipulating recursive data structures in a type-safe, modular manner. The prime application of CDTs is within language implementation and AST manipulation, and we present the basic concepts of CDTs in this section. More advanced concepts are introduced in Sections 4, 5, and 6.

2.1 Motivating Example

Consider an extension of the lambda calculus with integers, addition, let expressions, and error signalling:

$$e ::= \lambda x.e \mid x \mid e_1\, e_2 \mid n \mid e_1 + e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{error}$$

Our goal is to implement a pretty printer, a desugaring transformation, constant folding, and a call-by-value interpreter for the simple language above. The desugaring transformation will turn let expressions $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ into $(\lambda x.e_2)\, e_1$. Constant folding and evaluation will take place after desugaring, i.e. both computations are only defined for the core language without let expressions.

The standard approach to representing the language above is in terms of an algebraic data type:

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

$\textbf{data}\ \mathit{Exp} = \mathit{Lam}\ \mathit{Var}\ \mathit{Exp} \mid \mathit{Var}\ \mathit{Var} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lit}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Let}\ \mathit{Var}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Err}$

We may then straightforwardly define the pretty printer $\mathit{pretty} :: \mathit{Exp} \to \mathit{String}$. However, when we want to implement the desugaring transformation, we need a new algebraic data type:

$\textbf{data}\ \mathit{Exp}' = \mathit{Lam}'\ \mathit{Var}\ \mathit{Exp}' \mid \mathit{Var}'\ \mathit{Var} \mid \mathit{App}'\ \mathit{Exp}'\ \mathit{Exp}' \mid \mathit{Lit}'\ \mathit{Int} \mid \mathit{Plus}'\ \mathit{Exp}'\ \mathit{Exp}' \mid \mathit{Err}'$

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

That is, we need to replicate all constructors of *Exp*—except *Let*—into a new type Exp' of core expressions, in order to obtain a properly typed desugaring function $desug :: Exp \to Exp'$. Not only does this mean that we have to replicate the constructors, we also need to replicate common functionality, e.g. in order to obtain a pretty printer for Exp' we must either write a new function, or write an injection function $Exp' \to Exp$.

CDTs provide a solution that allows us to define the ASTs for (core) expressions without having to duplicate common constructors, and without having to give up on statically guaranteed invariants about the structure of the ASTs. CDTs take the viewpoint of data types as fixed points of functors [12], i.e. the definition of the AST data type is separated into non-recursive signatures (functors) on the one hand and the recursive structure on the other hand. For our example, we define the following signatures (omitting the straightforward *Functor* instance declarations):

```
data Lam a = Lam Var a      data Lit a   = Lit Int      data Let a = Let Var a a
data Var a  = Var Var       data Plus a = Plus a a     data Err a = Err
data App a  = App a a
```

Signatures can then be combined in a modular fashion by means of a formal sum of functors:

```
data (f :+: g) a = Inl (f a) | Inr (g a)
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr x) = Inr (fmap f x)
type Sig  = Lam :+: Var :+: App :+: Lit :+: Plus :+: Err :+: Let
type Sig' = Lam :+: Var :+: App :+: Lit :+: Plus :+: Err
```

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

```
data Term f = In {out :: f (Term f)}
```

Modulo strictness, *Term Sig* is isomorphic to *Exp*, and *Term Sig'* is isomorphic to Exp'.

The use of formal sums entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the right tags automatically, 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 subsignature relation $:\prec:$ can be constructively defined [22]. However, due to the limitations of Haskell's type class system, instances are restricted to the form $f :\prec: g$ where f is atomic, i.e. not a sum, and g is a right-associated sum, e.g. $g_1 :+: (g_2 :+: g_3)$ but not $(g_1 :+: g_2) :+: g_3$. With the carefully defined instances for $:\prec:$, injection and projection functions for terms can then be defined as follows:

```
inject :: (g :≺: f) ⇒ g (Term f) → Term f
inject = In . inj
project :: (g :≺: f) ⇒ Term f → Maybe (g (Term f))
project = proj . out
```

Additionally, in order to reduce the syntactic overhead, the CDTs library can automatically derive

smart constructors that comprise the injections [4], e.g.

$$iPlus :: (Plus :\prec: f) \Rightarrow Term\, f \to Term\, f \to Term\, f$$
$$iPlus\ x\ y = inject\ (Plus\ x\ y)$$

Using the derived smart constructors, we can then write expressions such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ without syntactic overhead:

$$e :: Term\ Sig$$
$$e = iLet\ \texttt{"x"}\ (iLit\ 2)\ ((iLam\ \texttt{"y"}\ (Var\ \texttt{"y"}\ `iPlus`\ Var\ \texttt{"x"}))\ `iApp`\ iLit\ 3)$$

In fact, the principal type of e is the *open* type:

$$(Lam :\prec: f, Var :\prec: f, App :\prec: f, Lit :\prec: f, Plus :\prec: f, Let :\prec: f) \Rightarrow Term\, f$$

which means that e can be used as a term over any signature containing at least these six signatures!

Next, we want to define the pretty printer, i.e. a function of type $Term\ Sig \to String$. In order to make a recursive function definition modular too, it is defined as the catamorphism of an algebra [12]:

$$\textbf{type}\ Alg\, f\ a = f\ a \to a$$
$$cata :: Functor\, f \Rightarrow Alg\, f\ a \to Term\, f \to a$$
$$cata\ \phi = \phi\ .\ fmap\ (cata\ \phi)\ .\ out$$

The advantage of this approach is that algebras can be easily combined over formal sums. A modular algebra definition is obtained by an open family of algebras indexed by the signature and closed under forming formal sums. This is achieved as a type class:

$$\textbf{class}\ Pretty\, f\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}} :: Alg\, f\ String$$
$$\textbf{instance}\ (Pretty\, f, Pretty\, g) \Rightarrow Pretty\ (f :+: g)\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (Inl\ x) = \phi_{\text{Pretty}}\ x$$
$$\quad \phi_{\text{Pretty}}\ (Inr\ x) = \phi_{\text{Pretty}}\ x$$
$$pretty :: (Functor\, f, Pretty\, f) \Rightarrow Term\, f \to String$$
$$pretty = cata\ \phi_{\text{Pretty}}$$

The instance declaration that lifts *Pretty* instances to sums is crucial. Yet, the structure of its declaration is independent from the particular algebra class, and the CDTs library provides a mechanism for automatically deriving such instances [4]. What remains in order to implement the pretty printer is to define instances of the *Pretty* algebra class for the six signatures:

$$\textbf{instance}\ Pretty\ Lam\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (Lam\ x\ e) = \texttt{"(\\"} ++ x ++ \texttt{". "} ++ e ++ \texttt{")"}$$
$$\textbf{instance}\ Pretty\ Var\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (Var\ x) = x$$
$$\textbf{instance}\ Pretty\ App\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (App\ e_1\ e_2) = \texttt{"("} ++ e_1 ++ \texttt{" "} ++ e_2 ++ \texttt{")"}$$
$$\textbf{instance}\ Pretty\ Lit\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (Lit\ n) = show\ n$$
$$\textbf{instance}\ Pretty\ Plus\ \textbf{where}$$
$$\quad \phi_{\text{Pretty}}\ (Plus\ e_1\ e_2) = \texttt{"("} ++ e_1 ++ \texttt{" + "} ++ e_2 ++ \texttt{")"}$$

instance *Pretty Let* **where**
$\phi_{\text{Pretty}}\ (\mathit{Let}\ x\ e_1\ e_2) = \texttt{"(let "} +\!\!+ x +\!\!+ \texttt{" = "} +\!\!+ e_1 +\!\!+ \texttt{" in "} +\!\!+ e_2 +\!\!+ \texttt{")"}$
instance *Pretty Err* **where**
$\phi_{\text{Pretty}}\ \mathit{Err} = \texttt{"error"}$

With these definitions we then have that *pretty e* evaluates to the string `(let x = 2 in ((\y. (y + x)) 3))`. Moreover, we automatically obtain a pretty printer for the core language as well, cf. the type of *pretty*.

3 Parametric Compositional Data Types

In the previous section we considered a first-order encoding of the language, which means that we have to be careful to ensure that computations are invariant under α-equivalence, e.g. when implementing capture-avoiding substitutions. *Higher-order abstract syntax* (HOAS) [15] remedies this issue, by representing binders and variables of the object language in terms of those of the meta language.

3.1 Higher-Order Abstract Syntax

In a standard Haskell HOAS encoding we replace the signatures *Var* and *Lam* by a revised *Lam* signature:

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

Now, however, *Lam* is no longer an instance of *Functor*, because a occurs both in a contravariant position and a covariant position. We therefore need to generalise functors in order to allow for negative occurrences of the recursive parameter. *Difunctors* [13] provide such a generalisation:

class $\mathit{Difunctor}\ f$ **where**
$\mathit{dimap} :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d$
instance $\mathit{Difunctor}\ (\to)$ **where**
$\mathit{dimap}\ f\ g\ h = g\ .\ h\ .\ f$
instance $\mathit{Difunctor}\ f \Rightarrow \mathit{Functor}\ (f\ a)$ **where**
$\mathit{fmap} = \mathit{dimap}\ \mathit{id}$

A difunctor must preserve the identity function and distribute over function composition:

$$\mathit{dimap}\ \mathit{id}\ \mathit{id} = \mathit{id} \qquad \text{and} \qquad \mathit{dimap}\ (f\ .\ g)\ (h\ .\ i) = \mathit{dimap}\ g\ h\ .\ \mathit{dimap}\ f\ i$$

The derived *Functor* instance obtained by fixing the contravariant argument will hence satisfy the functor laws, provided that the difunctor laws are satisfied.

Meijer and Hutton [13] showed that it is possible to perform recursion over difunctor terms:

$\mathbf{data}\ \mathit{Term}_{MH}\ f = \mathit{In}_{MH}\ \{\mathit{out}_{MH} :: f\ (\mathit{Term}_{MH}\ f)\ (\mathit{Term}_{MH}\ f)\}$
$\mathit{cata}_{MH} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to \mathit{Term}_{MH}\ f \to a$
$\mathit{cata}_{MH}\ \phi\ \psi = \phi\ .\ \mathit{dimap}\ (\mathit{ana}_{MH}\ \phi\ \psi)\ (\mathit{cata}_{MH}\ \phi\ \psi)\ .\ \mathit{out}_{MH}$
$\mathit{ana}_{MH} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to b \to \mathit{Term}_{MH}\ f$
$\mathit{ana}_{MH}\ \phi\ \psi = \mathit{In}_{MH}\ .\ \mathit{dimap}\ (\mathit{cata}_{MH}\ \phi\ \psi)\ (\mathit{ana}_{MH}\ \phi\ \psi)\ .\ \psi$

With Meijer and Hutton's approach, however, in order to lift an algebra $\phi :: f\ b\ a \to a$ to a catamorphism, we also need to supply the *inverse coalgebra* $\psi :: b \to f\ b\ a$. That is, in order to write a pretty printer we must supply a parser, which is not feasible—or perhaps even possible—in practice.

Fortunately, Fegaras and Sheard [8] realised that if the embedded functions within terms are *parametric*, then the inverse coalgebra is only used in order to *undo* computations performed by the algebra, since parametric functions can only "push around their arguments" without examining them. The solution proposed by Fegaras and Sheard is to add a *placeholder* to the structure of terms, which acts as a right-inverse of the catamorphism:[2]

$$\begin{aligned}
&\textbf{data}\ \mathit{Term}_{FS}\ f\ a = \mathit{In}_{FS}\ (f\ (\mathit{Term}_{FS}\ f\ a)\ (\mathit{Term}_{FS}\ f\ a)) \mid \mathit{Place}\ a\\
&\mathit{cata}_{FS} :: \mathit{Difunctor}\ f \Rightarrow (f\ a\ a \to a) \to \mathit{Term}_{FS}\ f\ a \to a\\
&\mathit{cata}_{FS}\ \phi\ (\mathit{In}_{FS}\ t) \quad = \phi\ (\mathit{dimap}\ \mathit{Place}\ (\mathit{cata}_{FS}\ \phi)\ t)\\
&\mathit{cata}_{FS}\ \phi\ (\mathit{Place}\ x) = x
\end{aligned}$$

We can then define e.g. a signature for lambda terms, and a function that calculates the number of bound variables occurring in a term, as follows (the example is adopted from Washburn and Weirich [25]):

$$\begin{aligned}
&\textbf{data}\ T\ a\ b = \mathit{Lam}\ (a \to b) \mid \mathit{App}\ b\ b \quad \text{-- } T \text{ is a difunctor, we omit the instance declaration}\\
&\phi :: T\ \mathit{Int}\ \mathit{Int} \to \mathit{Int}\\
&\phi\ (\mathit{Lam}\ f) \quad = f\ 1\\
&\phi\ (\mathit{App}\ x\ y) = x + y\\
&\mathit{countVar} :: \mathit{Term}_{FS}\ T\ \mathit{Int} \to \mathit{Int}\\
&\mathit{countVar} = \mathit{cata}_{FS}\ \phi
\end{aligned}$$

In the Term_{FS} encoding above, however, parametricity of the embedded functions is not guaranteed. More specifically, the type allows for three kinds of *exotic terms* [25], i.e. values in the meta language that do not correspond to terms in the object language:

$$\begin{aligned}
&\mathit{badPlace} :: \mathit{Term}_{FS}\ T\ \mathit{Bool}\\
&\mathit{badPlace} = \mathit{In}_{FS}\ (\mathit{Place}\ \mathit{True})\\
&\mathit{badCata} :: \mathit{Term}_{FS}\ T\ \mathit{Int}\\
&\mathit{badCata} = \mathit{In}_{FS}\ (\mathit{Lam}\ (\lambda x \to \textbf{if}\ \mathit{countVar}\ x \equiv 0\ \textbf{then}\ x\ \textbf{else}\ \mathit{Place}\ 0))\\
&\mathit{badCase} :: \mathit{Term}_{FS}\ T\ a\\
&\mathit{badCase} = \mathit{In}_{FS}\ (\mathit{Lam}\ (\lambda x \to \textbf{case}\ x\ \textbf{of}\ \mathit{Term}_{FS}\ (\mathit{App}\ _\ _) \to \mathit{Term}_{FS}\ (\mathit{App}\ x\ x); _ \to x))
\end{aligned}$$

Fegaras and Sheard showed how to avoid exotic terms by means of a custom type system. Washburn and Weirich [25] later showed that exotic terms can be avoided in a Haskell encoding via type parametricity and an abstract type of terms: terms are restricted to the type $\forall\ a\ .\ \mathit{Term}_{FS}\ f\ a$, and the constructors of Term_{FS} are hidden. Parametricity rules out *badPlace* and *badCata*, while the use of an abstract type rules out *badCase*.

3.2 Parametric Higher-Order Abstract Syntax

While the approach of Washburn and Weirich effectively rules out exotic terms in Haskell, we prefer a different encoding that relies on type parametricity only, and not an abstract type of terms. Our solution is inspired by Chlipala's *parametric higher-order abstract syntax* (PHOAS) [6]. PHOAS is similar to the restricted form of HOAS that we saw above; however, Chlipala makes the parametricity explicit in the definition of terms by distinguishing between the type of bound variables and the type of recursive terms. In Chlipala's approach, an algebraic data type encoding of lambda terms *LTerm* can effectively be defined via an auxiliary data type *LTrm* of "preterms" as follows:

[2] Actually, Fegaras and Sheard do not use difunctors, but the given definition corresponds to their encoding.

```
type LTerm  = ∀ a . LTrm a
data LTrm a = Lam (a → LTrm a) | Var a | App (LTrm a) (LTrm a)
```

The definition of *LTerm* guarantees that all functions embedded via *Lam* are parametric, and likewise that *Var*—Fegaras and Sheard's *Place*—can only be applied to variables bound by an embedded function. Atkey [2] showed that the encoding above adequately captures closed lambda terms modulo α-equivalence, assuming that there is no infinite data and that all embedded functions are total.

3.2.1 Parametric Terms

In order to transfer Chlipala's idea to non-recursive signatures and catamorphisms, we need to distinguish between covariant and contravariant uses of the recursive parameter. But this is exactly what difunctors do! We therefore arrive at the following definition of terms over difunctors:

```
newtype Term f = Term {unTerm :: ∀ a . Trm f a}
data Trm f a   = In (f a (Trm f a)) | Var a    -- "preterm"
```

Note the difference in *Trm* compared to $Term_{FS}$ (besides using the name *Var* rather than *Place*): the contravariant argument to the difunctor f is not the type of terms *Trm f a*, but rather a parametrised type a, which we quantify over at top-level to ensure parametricity. Hence, the only way to use a bound variable is to wrap it in a *Var* constructor—it is not possible to inspect the parameter. This representation more faithfully captures—we believe—the restricted form of HOAS than the representation of Washburn and Weirich: in our encoding it is explicit that bound variables are merely placeholders, and not the same as terms. Moreover, in some cases we actually *need* to inspect the structure of terms in order to define term transformations—we will see such an example in Section 3.2.3. With an abstract type of terms, this is not possible as Washburn and Weirich note [25].

Before we define algebras and catamorphisms, we lift the ideas underlying CDTs to *parametric compositional data types* (PCDTs), namely coproducts and implicit injections. Fortunately, the constructions of Section 2 are straightforwardly generalised (the instance declarations for :≺: are exactly as in *data types à la carte* [22], so we omit them here):

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

instance (Difunctor f, Difunctor g) ⇒ Difunctor (f :+: g) where
  dimap f g (Inl x) = Inl (dimap f g x)
  dimap f g (Inr x) = Inr (dimap f g x)

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

inject :: (g :≺: f) ⇒ g a (Trm f a) → Trm f a
inject = In . inj

project :: (g :≺: f) ⇒ Trm f a → Maybe (g a (Trm f a))
project (Term t) = proj t
project (Var _)  = Nothing
```

We can then recast our previous signatures from Section 2.1 as difunctors:

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

$$\textbf{type } Sig \quad = Lam :+: App :+: Lit :+: Plus :+: Err :+: Let$$
$$\textbf{type } Sig' \quad = Lam :+: App :+: Lit :+: Plus :+: Err$$

Finally, we can automatically derive instance declarations for *Difunctor* as well as smart constructor definitions that comprise the injections as for CDTs [4]. However, in order to avoid the explicit *Var* constructor, we insert *dimap Var id* into the declarations, e.g.

```
iLam :: (Lam :≺: f) ⇒ (Trm f a → Trm f a) → Trm f a
iLam f = inject (dimap Var id (Lam f))    -- (= inject (Lam (f . Var)))
```

Using *iLam* we then need to be aware, though, that even if it takes a function $Trm\ f\ a \to Trm\ f\ a$ as argument, the input to that function will always be of the form *Var x by construction*. We can now again represent terms such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ compactly as follows:

```
e :: Term Sig
e = Term (iLet (iLit 2) (λx → (iLam (λy → y `iPlus` x) `iApp` iLit 3)))
```

3.2.2 Algebras and Catamorphisms

Given the representation of terms as fixed points of difunctors, we can now define algebras and catamorphisms:

```
type Alg f a = f a a → a
cata :: Difunctor f ⇒ Alg f a → Term f → a
cata φ (Term t) = cat t
  where cat (In t)  = φ (fmap cat t)    -- recall: fmap = dimap id
        cat (Var x) = x
```

The definition of *cata* above is essentially the same as $cata_{FS}$. The only difference is that bound variables within terms are already wrapped in a *Var* constructor. Therefore, the contravariant argument to *dimap* is the identity function, and we consequently use the derived function *fmap* instead.

With these definitions in place, we can now recast the modular pretty printer from Section 2.1 to the new difunctor signatures. However, since we now use a higher-order encoding, we need to generate variable names for printing. We therefore arrive at the following definition (the example is adopted from Washburn and Weirich [25], but we use streams rather than lists to represent the sequence of available variable names):

```
data Stream a = Cons a (Stream a)
class Pretty f where
  φPretty :: Alg f (Stream String → String)
  -- instance declaration that lifts Pretty to coproducts omitted
pretty :: (Difunctor f, Pretty f) ⇒ Term f → String
pretty t = cata φPretty t (names 1)
  where names n = Cons ('x' : show n) (names (n + 1))
instance Pretty Lam where
  φPretty (Lam f) (Cons x xs) = "(\\" ++ x ++ ". " ++ f (const x) xs ++ ")"
instance Pretty App where
  φPretty (App e1 e2) xs = "(" ++ e1 xs ++ " " ++ e2 xs ++ ")"
```

instance *Pretty Lit* **where**
$\phi_{\text{Pretty}}\ (Lit\ n)\ \_ = show\ n$
instance *Pretty Plus* **where**
$\phi_{\text{Pretty}}\ (Plus\ e_1\ e_2)\ xs = \texttt{"("} +\!\!+ e_1\ xs +\!\!+ \texttt{" + "} +\!\!+ e_2\ xs +\!\!+ \texttt{")"}$
instance *Pretty Let* **where**
$\phi_{\text{Pretty}}\ (Let\ e_1\ e_2)\ (Cons\ x\ xs) = \texttt{"(let "} +\!\!+ x +\!\!+ \texttt{" = "} +\!\!+ e_1\ xs +\!\!+ \texttt{" in "} +\!\!+ e_2\ (const\ x)\ xs +\!\!+ \texttt{")"}$
instance *Pretty Err* **where**
$\phi_{\text{Pretty}}\ Err\ \_ = \texttt{"error"}$

With this implementation of *pretty* we then have that *pretty e* evaluates to the string `(let x1 = 2 in ((\x2. (x2 + x1)) 3))`.

3.2.3 Term Transformations

The pretty printer is an example of a modular computation over a PCDT. However, we also want to define computations over PCDTs that *construct* PCDTs, e.g. the desugaring transformation. That is, we want to construct functions of type $Term\ f \to Term\ g$, which means that we must construct functions of type $(\forall\ a\ .\ Trm\ f\ a) \to (\forall\ a\ .\ Trm\ g\ a)$. Following the approach of Section 3.2.2, we construct such functions by forming the catamorphisms of algebras of type $Alg\ f\ (\forall\ a\ .\ Trm\ g\ a)$, i.e. functions of type $f\ (\forall\ a\ .\ Trm\ g\ a) \to \forall\ a\ .\ Trm\ g\ a$. However, in order to avoid the nested quantifiers, we instead use *parametric term algebras* of type $\forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$. From such algebras we then obtain functions of the type $\forall\ a\ .\ (Trm\ f\ a \to Trm\ g\ a)$ as catamorphisms, which finally yield the desired functions of type $(\forall\ a\ .\ Trm\ f\ a) \to (\forall\ a\ .\ Trm\ g\ a)$. With these considerations in mind, we arrive at the following definition of the desugaring algebra type class:

class *Desug f g* **where**
$\phi_{\text{Desug}} :: \forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$ -- not $Alg\ f\ (Term\ g)$!
-- instance declaration that lifts *Desug* to coproducts omitted
$desug :: (Difunctor\ f, Desug\ f\ g) \Rightarrow Term\ f \to Term\ g$
$desug\ t = Term\ (cata\ \phi_{\text{Desug}}\ t)$

The algebra type class above is a *multi-parameter type class*: it is parametrised both by the domain signature f and the codomain signature g. We do this in order to obtain a desugaring function that is also modular in the codomain, similar to the evaluation function for vanilla CDTs [4].

We can now define the instances of *Desug* for the six signatures in order to obtain the desugaring function. However, by utilising overlapping instances we can make do with just two instance declarations:

instance $(Difunctor\ f, f :\prec: g) \Rightarrow Desug\ f\ g$ **where**
$\phi_{\text{Desug}} = inject\ .\ dimap\ Var\ id$ -- default instance for core signatures
instance $(App :\prec: f, Lam :\prec: f) \Rightarrow Desug\ Let\ f$ **where**
$\phi_{\text{Desug}}\ (Let\ e_1\ e_2) = iLam\ e_2\ `iApp`\ e_1$

Given a term $e :: Term\ Sig$, we then have that $desug\ e :: Term\ Sig'$, i.e. the type shows that indeed all syntactic sugar has been removed.

Whereas the desugaring transformation shows that we can construct PCDTs from PCDTs in a mod-

ular fashion, we did not make use of the fact that PCDTs can be inspected. That is, the desugaring transformation does not inspect the recursively computed values, cf. the instance declaration for *Let*. However, in order to implement the constant folding transformation, we actually need to inspect recursively computed PCDTs. We again utilise overlapping instances:

```
class Constf f g where
  φConstf :: ∀ a . Alg f (Trm g a)
  -- instance declaration that lifts Constf to coproducts omitted
constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata φConstf t)
instance (Difunctor f, f :≺: g) ⇒ Constf f g where
  φConstf = inject . dimap Var id    -- default instance
instance (Plus :≺: f, Lit :≺: f) ⇒ Constf Plus f where
  φConstf (Plus e1 e2) = case (project e1, project e2) of
                           (Just (Lit n), Just (Lit m)) → iLit (n + m)
                           _                            → e1 `iPlus` e2
```

Since we provide a default instance, we not only obtain constant folding for the core language, but also for the full language, i.e. *constf* has both the types $Term\ Sig' \to Term\ Sig'$ and $Term\ Sig \to Term\ Sig$.

4 Monadic Computations

In the last section we demonstrated how to extend CDTs with parametric higher-order abstract syntax, and how to perform modular, recursive computations over terms containing binders. In this section we investigate monadic computations over PCDTs.

4.1 Monadic Interpretation

While the previous examples of modular computations did not require effects, the call-by-value interpreter prompts the need for monadic computations: both in order to handle errors as well as controlling the evaluation order. Ultimately, we want to obtain a function of the type $Term\ Sig' \to m\ (Sem\ m)$, where the semantic domain *Sem* is defined as follows (we use an ordinary algebraic data type for simplicity):

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

Note that the monad only occurs in the codomain of *Fun*—if we want call-by-name semantics rather than call-by-value semantics, we simply add *m* also to the domain.

We can now implement the modular call-by-value interpreter similar to the previous modular computations, but using the monadic algebra carrier $m\ (Sem\ m)$:

```
class Monad m ⇒ Eval m f where
  φEval :: Alg f (m (Sem m))
  -- instance declaration that lifts Eval to coproducts omitted
eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata φEval
instance Monad m ⇒ Eval m Lam where
  φEval (Lam f) = return (Fun (f . return))
```

```
instance MonadError String m ⇒ Eval m App where
  ϕEval (App mx my) = do x ← mx
                         case x of Fun f → my >>= f
                                   _     → throwError "stuck"
instance Monad m ⇒ Eval m Lit where
  ϕEval (Lit n) = return (Int n)
instance MonadError String m ⇒ Eval m Plus where
  ϕEval (Plus mx my) = do x ← mx
                          y ← my
                          case (x, y) of (Int n, Int m) → return (Int (n + m))
                                         _              → throwError "stuck"
instance MonadError String m ⇒ Eval m Err where
  ϕEval Err = throwError "error"
```

In order to indicate errors in the course of the evaluation, we require the monad to provide a method to throw an error. To this end, we use the type class *MonadError*. Note how the modular design allows us to require the stricter constraint *MonadError String m* only for the cases where it is needed. This modularity of effects will become quite useful when we will rule out `"stuck"` errors in Section 6.

With the interpreter definition above we have that *eval* (*desug e*) evaluates to the value *Right* (*Int* 5) as expected, where *e* is as of page 10 and *m* is the *Either String* monad. Moreover, we also have that $0 + \mathbf{error}$ and $0 + \lambda x.x$ evaluate to *Left* `"error"` and *Left* `"stuck"`, respectively.

4.2 Monadic Computations with Implicit Sequencing

In the example above we use a monadic algebra carrier for monadic computations. For vanilla CDTs [4], however, we have previously shown how to perform monadic computations with *implicit sequencing*, by utilising the standard type class *Traversable*[3]:

```
type AlgM m f a = f a → m a
class Functor f ⇒ Traversable f where
  sequence :: Monad m ⇒ f (m a) → m (f a)
cataM :: (Traversable f, Monad m) ⇒ AlgM m f a → Term f → m a
cataM ϕ = ϕ <=< sequence . fmap (cataM ϕ) . out
```

AlgM m f a represents the type of monadic algebras [9] over *f* and *m*, with carrier *a*, which is different from *Alg f* (*m a*) since the monad only occurs in the codomain of the monadic algebra. *cataM* is obtained from *cata* in Section 2 by performing *sequence* after applying *fmap* and replacing function composition with monadic function composition $<=<$. That is, the recursion scheme takes care of sequencing the monadic subcomputations. Monadic algebras are useful for instance if we want to recursively project a term over a compound signature to a smaller signature:

```
deepProject :: (Traversable g, f :≺: g) ⇒ Term f → Maybe (Term g)
deepProject = cataM (liftM In . proj)
```

Moreover, in a call-by-value setting we may use a monadic algebra *Alg f m a* rather than an ordinary algebra with a monadic carrier *Alg f* (*m a*) in order to avoid the explicit sequencing of effects.

[3] We have omitted methods from the definition of *Traversable* that are not necessary for our purposes.

Turning back to parametric terms, we can apply the same idea to difunctors yielding the following definition of monadic algebras:

$$\textbf{type}\ \mathit{AlgM}\ m\ f\ a = f\ a\ a \to m\ a$$

Similarly, we can easily generalise *Traversable* and *cataM* to difunctors:

$$\begin{array}{l}\textbf{class}\ \mathit{Difunctor}\ f \Rightarrow \mathit{Ditraversable}\ f\ \textbf{where}\\ \quad \mathit{disequence} :: \mathit{Monad}\ m \Rightarrow f\ a\ (m\ b) \to m\ (f\ a\ b)\\ \mathit{cataM} :: (\mathit{Ditraversable}\ f, \mathit{Monad}\ m) \Rightarrow \mathit{AlgM}\ m\ f\ a \to \mathit{Term}\ f \to m\ a\\ \mathit{cataM}\ \phi\ (\mathit{Term}\ t) = \mathit{cat}\ t\ \textbf{where}\ \mathit{cat}\ (\mathit{In}\ t) \quad = \mathit{disequence}\ (\mathit{fmap}\ \mathit{cat}\ t) \gg\!= \phi\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{cat}\ (\mathit{Var}\ x) = \mathit{return}\ x\end{array}$$

Unfortunately, *cataM* only works for difunctors that do not use the contravariant argument. To see why this is the case, reconsider the *Lam* constructor; in order to define an instance of *Ditraversable* for *Lam* we must write a function of the type:

$$\mathit{disequence} :: \mathit{Monad}\ m \Rightarrow \mathit{Lam}\ a\ (m\ b) \to m\ (\mathit{Lam}\ a\ b)$$

Since *Lam* is isomorphic to the function type constructor $\to$, this is equivalent to a function of the type:

$$\forall\ a\ b\ m\ .\ \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to m\ (a \to b)$$

We cannot hope to be able to construct a meaningful combinator of that type. Intuitively, in a function of type $a \to m\ b$, the monadic effect of the result can depend on the input of type a. The monadic effect of a monadic value of type $m\ (a \to b)$ is not dependent on such input. For example, think of a state transformer monad *ST* with state *S* and its put function $\mathit{put} :: S \to \mathit{ST}\ ()$. What would be the corresponding transformation to a monadic value of type $\mathit{ST}\ (S \to ())$?

Hence, *cataM* does not extend to terms with binders, but it still works for terms without binders as in vanilla CDTs [4]. In particular, we cannot use *cataM* to define the call-by-value interpreter from Section 4.1.

5 Contexts and Term Homomorphisms

While the generality of catamorphisms makes them a powerful tool for modular function definitions, their generality at the same time inhibits flexibility and reusability. However, the full generality of catamorphisms is not always needed in the case of term transformations, which we discussed in Section 3.2.3. To this end, we have previously studied term homomorphisms [4] as a restricted form of term algebras. In this section we redevelop term homomorphisms for PCDTs.

5.1 From Terms to Contexts and back

The crucial idea behind term homomorphisms is to generalise terms to *contexts*, i.e. terms with *holes*. Following previous work [4] we define the generalisation of terms with holes as a *generalised algebraic data type (GADT)* [18] with *phantom types Hole* and *NoHole*:

$$\begin{array}{l}\textbf{data}\ \mathit{Cxt} :: * \to (* \to * \to *) \to * \to * \to * \to *\ \textbf{where}\\ \quad \begin{array}{lll}\mathit{In} & :: f\ a\ (\mathit{Cxt}\ h\ f\ a\ b) \to \mathit{Cxt}\ h & f\ a\ b\\ \mathit{Var} & :: a \to \mathit{Cxt}\ h & f\ a\ b\\ \mathit{Hole} & :: b \to \mathit{Cxt}\ \mathit{Hole}\ f\ a\ b\end{array}\end{array}$$

data *Hole*
data *NoHole*

The first argument to *Cxt* is a phantom type indicating whether the term contains holes or not. A context can thus be defined as:

type *Context* = *Cxt Hole*

That is, contexts *may* contain holes. On the other hand, terms must not contain holes, so we can recover our previous definition of preterms *Trm* as follows:

type *Trm f a* = *Cxt NoHole f a* ()

The definition of *Term* remains unchanged. This representation of contexts and preterms allows us to uniformly define functions that work on both types. For example, the function *inject* now has the type:

$$inject :: (g :\prec: f) \Rightarrow g\ a\ (Cxt\ h\ f\ a\ b) \to Cxt\ h\ f\ a\ b$$

5.2 Term Homomorphisms

In Section 3.2.3 we have shown that term transformations, i.e. functions of type *Term f* → *Term g*, are obtained as catamorphisms of parametric term algebras of type $\forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$. Spelling out the definition of *Alg*, such algebras are functions of type:

$$\forall\ a\ .f\ (Trm\ g\ a)\ (Trm\ g\ a) \to Trm\ g\ a$$

As we have argued previously [4], the fact that the target signature g occurs in both the domain and codomain in the above type prevents us from making use of the structure of the algebra's carrier type *Trm g a*. In particular, the constructions that we show in Section 5.3 are not possible with the above type.

In order to circumvent this restriction, we remove the occurrences of the algebra's carrier type *Trm g a* in the domain by replacing them with type variables:

$$\forall\ a\ b\ .f\ a\ b \to Trm\ g\ a$$

However, since we introduce a fresh variable b, functions of the above type are not able to use the corresponding parts of the argument for constructing the result. A value of type b cannot be injected into the type *Trm g a*.

This is where contexts come into the picture: we enable the use of values of type b in the result by replacing the codomain type *Trm g a* with *Context g a b*. The result is the following type of *term homomorphisms*:

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

A function $\rho :: Hom\ f\ g$ is a transformation of constructors from f into a context over g, i.e. a term over g that may embed values taken from the arguments of the f-constructor. The parametric polymorphism of the type guarantees that the arguments of the f-constructor cannot be inspected but only embedded into the result context. In order to apply term homomorphisms to terms, we need an auxiliary function that merges nested contexts:

$$\begin{aligned}
&appCxt :: Difunctor\ f \Rightarrow Context\ f\ a\ (Cxt\ h\ f\ a\ b) \to Cxt\ h\ f\ a\ b\\
&appCxt\ (In\ t) \quad\ = In\ (fmap\ appCxt\ t)\\
&appCxt\ (Var\ x) \quad = Var\ x\\
&appCxt\ (Hole\ h) = h
\end{aligned}$$

Given a context that has terms embedded in its holes, we obtain a term as a result; given a context with embedded contexts, the result is again a context.

Using the combinator above we can now apply a term homomorphism to a preterm—or more generally, to a context:

$$\begin{aligned}
&\mathit{appHom} :: (\mathit{Difunctor}\, f, \mathit{Difunctor}\, g) \Rightarrow \mathit{Hom}\, f\ g \to \mathit{Cxt}\, h\, f\ a\ b \to \mathit{Cxt}\, h\ g\ a\ b \\
&\mathit{appHom}\ \rho\ (\mathit{In}\ t) \quad = \mathit{appCxt}\ (\rho\ (\mathit{fmap}\ (\mathit{appHom}\ \rho)\ t)) \\
&\mathit{appHom}\ \rho\ (\mathit{Var}\ x) \quad = \mathit{Var}\ x \\
&\mathit{appHom}\ \rho\ (\mathit{Hole}\ h) = \mathit{Hole}\ h
\end{aligned}$$

From *appHom* we can then obtain the actual transformation on terms as follows:

$$\begin{aligned}
&\mathit{appTHom} :: (\mathit{Difunctor}\, f, \mathit{Difunctor}\, g) \Rightarrow \mathit{Hom}\, f\ g \to \mathit{Term}\, f \to \mathit{Term}\, g \\
&\mathit{appTHom}\ \rho\ (\mathit{Term}\ t) = \mathit{Term}\ (\mathit{appHom}\ \rho\ t)
\end{aligned}$$

Before we describe the benefits of term homomorphisms over term algebras, we reconsider the desugaring transformation from Section 3.2.3, but as a term homomorphism rather than a term algebra:

```
class Desug f g where
  ρDesug :: Hom f g
  -- instance declaration that lifts Desug to coproducts omitted
desug :: (Difunctor f, Difunctor g, Desug f g) ⇒ Term f → Term g
desug = appTHom ρDesug
instance (Difunctor f, Difunctor g, f :≺: g) ⇒ Desug f g where
  ρDesug = In . fmap Hole . inj    -- default instance for core signatures
instance (App :≺: f, Lam :≺: f) ⇒ Desug Let f where
  ρDesug (Let e1 e2) = inject (Lam (Hole . e2)) `iApp` Hole e1
```

Note how, in the instance declaration for *Let*, the constructor *Hole* is used to embed arguments of the constructor *Let*, viz. e_1 and e_2, into the context that is constructed as the result.

As for the desugaring function in Section 3.2.3, we utilise overlapping instances to provide a default translation for the signatures that need not be translated. The definitions above yield the desired desugaring function $\mathit{desug} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Term}\ \mathit{Sig}'$.

5.3 Transforming and Combining Term Homomorphisms

In the following we shall shortly describe what we actually gain by adopting the term homomorphism approach. First, term homomorphisms enable automatic propagation of annotations, where annotations are added via a restricted difunctor product, namely a product of a difunctor f and a constant c:

$$\mathbf{data}\ (f :\&: c)\ a\ b = f\ a\ b :\&: c$$

For instance, the type of ASTs of our language where each node is annotated with source positions is captured by the type $\mathit{Term}\ (\mathit{Sig} :\&: \mathit{SrcPos})$. With a term homomorphism $\mathit{Hom}\, f\ g$ we automatically get a lifted version $\mathit{Hom}\ (f :\&: c)\ (g :\&: c)$, which propagates annotations from the input to the output. Hence, from our desugaring function in the previous section we automatically get a lifted function on parse trees $\mathit{Term}\ (\mathit{Sig} :\&: \mathit{SrcPos}) \to \mathit{Term}\ (\mathit{Sig}' :\&: \mathit{SrcPos})$, which propagates source positions from the syntactic sugar to the core constructs. We omit the details here, but note that the constructions for CDTs [4] carry over straightforwardly to PCDTs.

The second motivation for introducing term homomorphisms is deforestation [23]. As we have shown previously [4], it is not possible to fuse two term algebras in order to traverse the term only once. That is, we do not find a composition operator $\circledcirc$ on algebras that satisfies the following equation:

$$cata\ \phi_1 \,.\, cata\ \phi_2 = cata\ (\phi_1 \circledcirc \phi_2) \quad \text{for all } \phi_1 :: Alg\ g\ a \text{ and } \phi_2 :: \forall\, a \,.\, Alg\ f\ (Trm\ g\ a)$$

With term homomorphism, however, we do have such a composition operator $\circledcirc$:

$$(\circledcirc) :: (Difunctor\ g, Difunctor\ h) \Rightarrow Hom\ g\ h \to Hom\ f\ g \to Hom\ f\ h$$
$$\rho_1 \circledcirc \rho_2 = appHom\ \rho_1 \,.\, \rho_2$$

For this composition, we then obtain the desired equation:

$$appHom\ \rho_1 \,.\, appHom\ \rho_2 = appHom\ (\rho_1 \circledcirc \rho_2) \quad \text{for all } \rho_1 :: Hom\ g\ h \text{ and } \rho_2 :: Hom\ f\ g$$

In fact, we can also compose an arbitrary algebra with a term homomorphism:

$$(\boxdot) :: Difunctor\ g \Rightarrow Alg\ g\ a \to Hom\ f\ g \to Alg\ f\ a$$
$$\phi \boxdot \rho = free\ \phi\ id \,.\, \rho$$

where

$$\begin{aligned} &free :: Difunctor\ f \Rightarrow Alg\ f\ a \to (b \to a) \to Cxt\ h\ f\ a\ b \to a \\ &free\ \phi\ f\ (In\ t) \quad = \phi\ (fmap\ (free\ \phi\ f)\ t) \\ &free\ \_\ \_\ (Var\ x) \quad = x \\ &free\ \_\ f\ (Hole\ h) = f\ h \end{aligned}$$

The composition of algebras and homomorphisms satisfies the following equation:

$$cata\ \phi \,.\, appHom\ \rho = cata\ (\phi \boxdot \rho) \quad \text{for all } \phi :: Alg\ g\ a \text{ and } \rho :: Hom\ f\ g$$

For example, in order to evaluate a term with syntactic sugar, rather than composing *eval* and *desug*, we can use the function $cata\ (\phi_{\text{Eval}} \boxdot \rho_{\text{Desug}})$, which only traverses the term once. This transformation can be automated using GHC's rewrite mechanism [14] and our experimental results for CDTs show that the thus obtained speedup is significant [4].

6 Generalised Parametric Compositional Data Types

In this section we briefly describe how to lift the construction of mutually recursive data types and—more generally—GADTs from CDTs to PCDTs. The construction is based on the work of Johann and Ghani [10]. For CDTs the generalisation, roughly speaking, amounts to lifting functors to (generalised) *higher-order functors* [10], and functions on terms to *natural transformations*, as shown earlier [4]:

$$\textbf{type}\ a \xrightarrow{\bullet} b = \forall\, i \,.\, a\ i \to b\ i$$
$$\textbf{class}\ HFunctor\ f\ \textbf{where}$$
$$\quad hfmap :: a \xrightarrow{\bullet} b \to f\ a \xrightarrow{\bullet} f\ b$$

Now, signatures are of the kind $(* \to *) \to * \to *$, rather than $* \to *$, which reflects the fact that signatures are now *indexed types*, and so are terms (or contexts in general). Consequently, the carrier of an algebra is a type constructor of kind $* \to *$:

$$\textbf{type}\ Alg\ f\ a = f\ a \xrightarrow{\bullet} a$$

Since signatures will be defined as GADTs, we effectively deal with *many-sorted algebras*. If a subterm has the type index i, then the value computed recursively by a catamorphism will have the type $a\ i$. The

coproduct $:+:$ and the automatic injections $:\prec:$ carry over straightforwardly from functors to higher-order functors [4].

In order to lift the ideas from CDTs to PCDTs, we need to consider indexed difunctors. This prompts the notion of *higher-order difunctors*:

```
class HDifunctor f where
  hdimap :: (a -•> b) → (c -•> d) → f b c -•> f a d
instance HDifunctor f ⇒ HFunctor (f a) where
  hfmap = hdimap id
```

Note the familiar pattern from ordinary PCDTs: a higher-order difunctor gives rise to a higher-order functor when the contravariant argument is fixed.

To illustrate higher-order difunctors, consider a modular GADT encoding of our core language:

```
data TArrow i j
data TInt
data Lam :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Lam :: (a i → b j) → Lam a b (i `TArrow` j)
data App :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  App :: b (i `TArrow` j) → b i → App a b j
data Lit :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Lit :: Int → Lit a b TInt
data Plus :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Plus :: b TInt → b TInt → Plus a b TInt
data Err :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Err :: Err a b i
type Sig' = Lam :+: App :+: Lit :+: Plus :+: Err
```

Note, in particular, the type of *Lam*: now the bound variable is typed!

We use *TArrow* and *TInt* as type indices for the GADT definitions above. The preference of these fresh types over Haskell's $\rightarrow$ and *Int* is meant to emphasise that these phantom types are only labels that represent the type constructors of our object language.

We use the coproduct $:+:$ of higher-order difunctors above to combine signatures, which is easily defined, and as for CDTs it is straightforward to lift instances of *HDifunctor* for f and g to an instance for $f :+: g$. Similarly, we can generalise the relation $:\prec:$ from difunctors to higher-order difunctors, so we omit its definition here.

The type of generalised parametric (pre)terms can now be constructed as an indexed type:

```
newtype Term f i = Term {unTerm :: ∀ a . Trm f a i}
data Trm f a i   = In (f a (Trm f a) i) | Var (a i)
```

Moreover, we use smart constructors as for PCDTs to compactly construct terms, for instance:

```
e :: Term Sig' TInt
e = Term (iLam (λx → x `iPlus` x) `iApp` iLit 2)
```

Finally, we can lift algebras and their induced catamorphisms by lifting the definitions in Section 3.2.2 via natural transformations and higher-order difunctors:

```
type Alg f a = f a a -•> a
cata :: HDifunctor f => Alg f a -> Term f -•> a
cata φ (Term t) = cat t
  where cat (In t)    = φ (hfmap cat t)    -- recall: hfmap = hdimap id
        cat (Var x)   = x
```

With the definitions above we can now define a call-by-value interpreter for our typed example language. To this end, we must provide a type-level function that, for a given object language type constructed from *TArrow* and *TInt*, selects the corresponding subset of the semantic domain *Sem m* from Section 4.1. This can be achieved via Haskell's *type families* [17]:

```
type family Sem (m :: * -> *) i
type instance Sem m (i `TArrow` j) = Sem m i -> m (Sem m j)
type instance Sem m TInt           = Int
```

The type *Sem m t* is obtained from an object language type t by replacing each function type t_1 `` `TArrow` `` t_2 occurring in t with *Sem m* $t_1 \to m$ (*Sem m* t_2) and each *TInt* with *Int*.

In order to make *Sem* into a proper type—as opposed to a mere type synonym—and simultaneously add the monad m at the top level, we define a **newtype** M:

```
newtype M m i = M { unM :: m (Sem m i) }
class Monad m => Eval m f where
  φEval :: f (M m) (M m) i -> m (Sem m i)    -- M . φEval :: Alg f (M m) is the actual algebra
eval :: (Monad m, HDifunctor f, Eval m f) => Term f i -> m (Sem m i)
eval = unM . cata (M . φEval)
```

We can then provide the instance declarations for the signatures of the core language, and effectively obtain a tagless, modular, and extendable monadic interpreter:

```
instance Monad m => Eval m Lam where
  φEval (Lam f) = return (unM . f . M . return)
instance Monad m => Eval m App where
  φEval (App (M mf) (M mx)) = do f <- mf
                                 mx >>= f
instance Monad m => Eval m Lit where
  φEval (Lit n) = return n
instance Monad m => Eval m Plus where
  φEval (Plus (M mx) (M my)) = do x <- mx
                                  y <- my
                                  return (x + y)
instance MonadError String m => Eval m Err where
  φEval Err = throwError "error"
```

With the above definition of *eval* we have, for instance, that *eval e* :: *Either String Int* evaluates to the value *Right* 4. Due to the fact that we now have a typed language, the *Err* constructor is the only source of an erroneous computation—the interpreter cannot get stuck. Moreover, since the modular specification of the interpreter only enforces the constraint *MonadError String m* for the signature *Err*, the term e can in fact be interpreted in the identity monad, rather than the *Either String* monad, as it does not contain

error. Consequently, we know statically that the evaluation of *e* cannot fail!

Note that computations over generalised PCDTs are not limited to the tagless approach that we have illustrated above. We could have easily reformulated the semantic domain *Sem m* from Section 4.1 as a GADT to use it as the carrier of a many-sorted algebra. Other natural carriers for many-sorted algebras are the type families of terms *Term f*, of course.

Other concepts that we have introduced for vanilla PCDTs before can be transferred straightforwardly to generalised PCDTs in the same fashion. This includes contexts and term homomorphisms.

7 Practical Considerations

The motivation for introducing CDTs was to make Swierstra's *data types à la carte* [22] readily useful in practice. Besides extending *data types à la carte* with various aspects, such as monadic computations and term homomorphisms, the CDTs library provides all the generic functionality as well as automatic derivation of boilerplate code. With (generalised) PCDTs we have followed that path. Our library provides Template Haskell [20] code to automatically derive instances of the required type classes, such as *Difunctor* and *Ditraversable*, as well as smart constructors and lifting of algebra type classes to coproducts. Moreover, our library supports automatic derivation of standard type classes *Show*, *Eq*, and *Ord* for terms, similar to Haskell's **deriving** mechanism. We show how to derive instances of *Eq* in the following subsection. *Ord* follows in the same fashion, and *Show* follows an approach similar to the pretty printer in Section 3.2.2, but using the monad *FreshM* that is also used to determine equality, as we shall see below.

Figure 1 provides the complete source code needed to implement our example language from Section 2.1. Note that we have derived *Show*, *Eq*, and *Ord* instances for terms of the language—in particular the term *e* is printed as `Let (Lit 2) (\a -> App (Lam (\b -> Plus b a)) (Lit 3))`.

7.1 Equality

A common pattern when programming in Haskell is to derive instances of the type class *Eq*, for instance in order to test the desugaring transformation in Section 3.2.3. While the use of PHOAS ensures that all functions are invariant under α-renaming, we still have to devise an algorithm that decides α-equivalence. To this end, we will turn the rather elusive representation of bound variables via functions into a concrete form.

In order to obtain concrete representations of bound variables, we provide a method for generating fresh variable names. This is achieved via a monad *FreshM* offering the following operations:

$$\mathit{withName} :: (\mathit{Name} \to \mathit{FreshM}\ a) \to \mathit{FreshM}\ a$$
$$\mathit{evalFreshM} :: \mathit{FreshM}\ a \to a$$

FreshM is an abstraction of an infinite sequence of fresh names. The function *withName* provides a fresh name. Names are represented by the abstract type *Name*, which implements instances of *Show*, *Eq*, and *Ord*.

We first introduce a variant of the type class *Eq* that uses the *FreshM* monad:

class *PEq a* **where**
  $\mathit{peq} :: a \to a \to \mathit{FreshM}\ \mathit{Bool}$

This type class is used to define the type class *EqD* of equatable difunctors, which lifts to coproducts:

```
class EqD f where
  eqD :: PEq a ⇒ f Name a → f Name a → FreshM Bool
instance (EqD f, EqD g) ⇒ EqD (f :+: g) where
  eqD (Inl x) (Inl y) = x `eqD` y
  eqD (Inr x) (Inr y) = x `eqD` y
  eqD _       _       = return False
```

We then obtain equality of terms as follows (we do not consider contexts here for simplicity):

```
instance EqD f ⇒ PEq (Trm f Name) where
  peq (In t1) (In t2)   = t1 `eqD` t2
  peq (Var x1) (Var x2) = return (x1 ≡ x2)
  peq _        _        = return False
instance (Difunctor f, EqD f) ⇒ Eq (Term f) where
  (≡) (Term x) (Term y) = evalFreshM ((x :: Trm f Name) `peq` y)
```

Note that we need to explicitly instantiate the parametric type in x to *Name* in the last instance declaration, in order to trigger the instance for *Trm f Name* defined above.

Equality of terms, i.e. α-equivalence, has thus been reduced to providing instances of *EqD* for the difunctors comprising the signature of the term, which for *Lam* can be defined as follows:

```
instance EqD Lam where
  eqD (Lam f) (Lam g) = withName (λx → f x `peq` g x)
```

That is, f and g are considered equal if they are equal when applied to the same fresh name x.

8 Discussion and Related Work

Implementing languages with binders can be a difficult task. Using explicit variable names, we have to be careful in order to make sure that functions on ASTs are invariant under α-renaming. HOAS [15] is one way of tackling this problem, by reusing the binding mechanisms of the implementation language to define those of the object language. The challenge with HOAS, however, is that it is difficult to perform recursive computations over ASTs with binders [8, 13, 19, 25]. Besides what is documented in this paper, we have also lifted (generalised) parametric compositional data types to other (co)recursion schemes, such as anamorphisms and histomorphisms. Moreover, term homomorphisms can be straightforwardly extended with a state space: depending on how the state is propagated, this yields bottom-up resp. top-down tree transducers [7].

Our approach of using PHOAS [6] amounts to the same restriction on embedded functions as Fegeras and Sheard [8], and Washburn and Weirich [25]. However, unlike Washburn and Weirich's Haskell implementation, our approach does not rely on making the type of terms abstract. Not only is it interesting to see that we can do without type abstraction, in fact, we sometimes need to inspect terms in order to write functions that produce terms, such as our constant folding algorithm. With Washburn and Weirich's encoding this is not possible.

Ahn and Sheard [1] recently showed how to generalise the recursion schemes of Washburn and Weirich to Mendler-style recursion schemes, using the same representation for terms as Washburn and Weirich. Hence their approach also suffers from the inability to inspect terms. Although we could easily adopt Mendler-style recursion schemes in our setting, their generality does not make a difference in a

non-strict language such as Haskell. Additionally, Ahn and Sheard pose the open question whether there is a safe (i.e., terminating) way to apply histomorphisms to terms with negative recursive occurrences: although we have not investigated termination properties of our histomorphisms, we conjecture that the use of our parametric terms—which are purely inductive—may provide one solution.

The *finally tagless* approach of Carette et al. [5] has been proposed as an alternative solution to the expression problem [24]. While the approach is very simple and elegant, and also supports (typed) higher-order encodings, the approach falls short when we want to define recursive, modular computations that construct modular terms too. Atkey et al. [3], for instance, use the finally tagless approach to build a modular interpreter. However, the interpreter cannot be made modular in the return type, i.e. the language defining values. Hence, when Atkey et al. extend their expression language they need to also change the data type that represents values, which means that the approach is not fully modular. Although our interpreter in Section 4.1 also uses a fixed domain of values *Sem*, we can make the interpreter fully modular by also using a PCDT for the return type, and using a multi-parameter type class definition similar to the desugaring transformation in Section 3.2.3.

Nominal sets [16] is another approach for dealing with binders, in which variables are explicit, but recursively defined functions are guaranteed to be invariant with respect to α-equivalence of terms. Implementations of this approach, however, require extensions of the metalanguage [21], and the approach is therefore not immediately usable in Haskell.

Acknowledgement

The authors wish to thank Andrzej Filinski for his insightful comments on an earlier version of this paper.

References

[1] Ki Yung Ahn & Tim Sheard (2011): *A Hierarchy of Mendler style Recursion Combinators: Taming Inductive Datatypes with Negative Occurrences*. In: *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, ACM, New York, NY, USA, pp. 234–246, doi:10.1145/2034773.2034807.

[2] Robert Atkey (2009): *Syntax for Free: Representing Syntax with Binding Using Parametricity*. In Pierre-Louis Curien, editor: *Typed Lambda Calculi and Applications*, Springer Berlin / Heidelberg, pp. 35–49, doi:10.1007/978-3-642-02273-9_5.

[3] Robert Atkey, Sam Lindley & Jeremy Yallop (2009): *Unembedding Domain-Specific Languages*. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, ACM, New York, NY, USA, pp. 37–48, doi:10.1145/1596638.1596644.

[4] Patrick Bahr & Tom Hvitved (2011): *Compositional Data Types*. In: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/2036918.2036930.

[5] Jacques Carette, Oleg Kiselyov & Chung-Chieh Shan (2009): *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*. *Journal of Functional Programming* 19(05), pp. 509–543, doi:10.1017/S0956796809007205.

[6] Adam Chlipala (2008): *Parametric Higher-Order Abstract Syntax for Mechanized Semantics*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ACM, New York, NY, USA, pp. 143–156, doi:10.1145/1411204.1411226.

[7] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison & Marc Tommasi (2007): *Tree Automata Techniques and Applications*. Draft.

[8] Leonidas Fegaras & Tim Sheard (1996): *Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space)*. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, New York, NY, USA, pp. 284–294, doi:10.1145/237721.237792.

[9] Maarten Fokkinga (1994): *Monadic Maps and Folds for Arbitrary Datatypes*. Technical Report, Memoranda Informatica, University of Twente.

[10] Patricia Johann & Neil Ghani (2008): *Foundations for Structured Programming with GADTs*. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, New York, NY, USA, pp. 297–308, doi:10.1145/1328438.1328475.

[11] Simon Marlow (2010): *Haskell 2010 Language Report*.

[12] Erik Meijer, Maarten Fokkinga & Ross Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In: *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 124–144, doi:10.1007/3540543961_7.

[13] Erik Meijer & Graham Hutton (1995): *Bananas in Space: Extending Fold and Unfold to Exponential Types*. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, ACM, New York, NY, USA, pp. 324–333, doi:10.1145/224164.224225.

[14] Simon Peyton Jones, Andrew Tolmach & Tony Hoare (2001): *Playing by the Rules: Rewriting as a practical optimisation technique in GHC*. In: *2001 ACM SIGPLAN 2001 Haskell Workshop*, pp. 203–233.

[15] Frank Pfenning & Conal Elliot (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, pp. 199–208, doi:10.1145/53990.54010.

[16] Andrew M. Pitts (2006): *Alpha-Structural Recursion and Induction*. *Journal of the ACM* 53, pp. 459–506, doi:10.1145/1147954.1147961.

[17] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty & Martin Sulzmann (2008): *Type Checking with Open Type Functions*. In: *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, ACM, New York, NY, USA, pp. 51–62, doi:10.1145/1411204.1411215.

[18] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann & Dimitrios Vytiniotis (2009): *Complete and Decidable Type Inference for GADTs*. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ACM, New York, NY, USA, pp. 341–352, doi:10.1145/1596550.1596599.

[19] Carsten Schürmann, Joëlle Despeyroux & Frank Pfenning (2001): *Primitive recursion for higher-order abstract syntax*. *Theoretical Computer Science* 266(1-2), pp. 1–57, doi:10.1016/S0304-3975(00)00418-7.

[20] Tim Sheard & Simon Peyton Jones (2002): *Template Meta-programming for Haskell*. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ACM, New York, NY, USA, pp. 1–16, doi:10.1145/581690.581691.

[21] Mark R. Shinwell, Andrew M. Pitts & Murdoch J. Gabbay (2003): *FreshML: Programming with Binders Made Simple*. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ACM, New York, NY, USA, pp. 263–274, doi:10.1145/944705.944729.

[22] Wouter Swierstra (2008): *Data types à la carte*. *Journal of Functional Programming* 18(04), pp. 423–436, doi:10.1017/S0956796808006758.

[23] Philip Wadler (1990): *Deforestation: Transforming programs to eliminate trees*. *Theoretical Computer Science* 73(2), pp. 231–248, doi:10.1016/0304-3975(90)90147-A.

[24] Philip Wadler (1998): *The Expression Problem*. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[25] Geoffrey Washburn & Stephanie Weirich (2008): *Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism*. *Journal of Functional Programming* 18(01), pp. 87–140, doi:10.1017/S0956796807006557.

```
import Data.Comp.Param
import Data.Comp.Param.Show ()
import Data.Comp.Param.Equality ()
import Data.Comp.Param.Ordering ()
import Data.Comp.Param.Derive
import Control.Monad.Error (MonadError, throwError)

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

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

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

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

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

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

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

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

-- * Constant folding
class Constf f g where
  constfAlg :: forall a. Alg f (Trm g a)

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

constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata constfAlg t)

instance (Difunctor f, f :<: g) ⇒ Constf f g where
  constfAlg = inject . dimap Var id -- default instance

instance (Plus :<: f, Lit :<: f) ⇒ Constf Plus f where
  constfAlg (Plus e1 e2) = case (project e1, project e2) of
                             (Just (Lit n),Just (Lit m)) → iLit (n + m)
                             _                           → e1 `iPlus` e2

-- * Call-by-value evaluation
data Sem m = Fun (Sem m → m (Sem m)) | Int Int

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

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

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

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

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

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

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

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

Figure 1: Complete example using the parametric compositional data types library.