Composing and Decomposing Data Types

A Closed Type Families Implementation of Data Types à la Carte

Patrick Bahr

Department of Computer Science
University of Copenhagen
paba@di.ku.dk

Abstract

Wouter Swierstra's *data types à la carte* is a technique to modularise data type definitions in Haskell. We give an alternative implementation of data types à la carte that offers more flexibility in composing and decomposing data types. To achieve this, we refine the subtyping constraint, which is at the centre of data types à la carte. On the one hand this refinement is more general, allowing subtypings that intuitively should hold but were not derivable beforehand. This aspect of our implementation removes previous restrictions on how data types can be combined. On the other hand our refinement is more restrictive, disallowing subtypings that lead to more than one possible injection and should therefore be considered programming errors. Furthermore, from this refined subtyping constraint we derive a new constraint to express type isomorphism. We show how this isomorphism constraint allows us to decompose data types and to define extensible functions on data types in an ad hoc manner. The implementation makes essential use of closed type families in Haskell. The use of closed type families instead of type classes comes with a set of trade-offs, which we review in detail. Finally, we show that our technique can be used for other similar problem domains.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords expression problem; closed type families; two-level types; modularity

1. Introduction

Data types à la carte (Swierstra 2008) is a simple, yet powerful approach to defining data types and functions on them in a modular fashion. It provides a solution to the expression problem, which is "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" (Wadler 1998).

The elegance of Swierstra's data types à la carte lies in its simplicity. It can be implemented and explained in a few lines of Haskell code.

WGP '14, August 31, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3042-8/14/08…$15.00.
http://dx.doi.org/10.1145/2633628.2633635

Central to this technique is the idea to represent a recursive data type as a two-level type (Sheard and Pasalic 2004) $\mathit{Fix}\ f$ consisting of a *signature functor* f and a *knot-tying* fixpoint constructor Fix. As a consequence, modularity over the data type $\mathit{Fix}\ f$ can be expressed in terms of the signature functor f. The key components to achieve this are (1) the sum operator :+: that allows the programmer to combine two signatures f and g to form their sum f :+: g, and (2) the binary constraint f :≺: g to express that a signature f is *subsumed* by a signature g.

In this paper we present an alternative definition of the subsumption relation :≺:. In its original form it has been defined as a Haskell type class with two parameters. While this provides a clean and simple implementation it suffers from a severe restriction of Haskell's type class resolution: there is no backtracking.

A consequence of this restriction is that we may not derive, for example, the following subsumption, even though every summand on the left also occurs on the right:

$$f \mathrel{:+:} (g \mathrel{:+:} h) \mathrel{:\prec:} (f \mathrel{:+:} g) \mathrel{:+:} h$$

Even the simpler relation $g \mathrel{:\prec:} (f \mathrel{:+:} g) \mathrel{:+:} h$ is out of reach. For many small scale uses of data types à la carte, this restriction is not an issue or can be worked around. However, in practice this restriction creates a number of problems. The most severe of these problems occur in the form of *leaky abstractions*: when refactoring a signature functor f by splitting it into two components f_1, f_2 such that $f \simeq f_1 \mathrel{:+:} f_2$, previous subsumption relations may not hold anymore.

In order to overcome these restrictions and avoid the problems that stem from them, we implement the type constraint :≺: using the recently introduced *closed type families* (Eisenberg et al. 2014). As we shall show, the resulting subsumption constraint :≺: is much more flexible and powerful. It permits new use cases such as an isomorphism constraint :≃:, which allows the programmer to decompose and recombine signatures in an ad hoc manner.

In particular, the contributions of this paper are the following:

- We define a binary type constraint :≺: that accurately characterises the intuitive notion of signature subsumption, namely such that $f \mathrel{:\prec:} g$ iff each of the summands in f is unique and has a unique counterpart in g.
- We demonstrate that this refinement of Swierstra's original definition of :≺: permits new use cases for data types à la carte. In particular, it allows us to compose signature functors – and thus data types – more freely without giving up the utility provided by the subsumption relation.
- With the refined version of :≺:, we are able to conservatively characterise isomorphism of signature functors: we define the constraint $f \mathrel{:\simeq:} g$ as the conjunction of $f \mathrel{:\prec:} g$ and $g \mathrel{:\prec:} f$.

- The isomorphism signature constraint $:\simeq:$ allows the programmer to also decompose signatures more flexibly. The fact that a signature f can be decomposed into g and h can be expressed as $f :\simeq: g :+: h$. We demonstrate the utility of this constraint by a number of examples.
- We add restrictions to the subsumption constraint $:\prec:$ in order to detect and avoid ambiguities that arise in the injection functions that are derived from instances of $f :\prec: g$. Such ambiguities arise when a summand occurs multiple times in the right-hand side g, e.g. in the case of the instance $f :\prec: f :+: f$.
- We give an analysis of the costs and benefits of replacing Swierstra's original implementation with our implementation.
- Our technique is applicable to other similar problem domains. We illustrate this observation on extensible product types.

The remainder of this paper is structured as follows: in section 2 we recap data types à la carte and demonstrate the problems that we address in this paper. Section 3 is a brief primer on closed type families and their idiosyncrasies. Our implementation of $:\prec:$ is given in three steps: in section 4 we give a simple backtracking variant of Swierstra's definition; section 5 generalises this implementation to allow arbitrary compound signatures on the left-hand side; and section 6 presents our final implementation, which provides better error messages and improves performance. In section 7 we review the limitations of our implementation, discuss related work, and illustrate other applications of our technique.

The subsumption constraint $:\prec:$ along with the surrounding infrastructure as presented in this paper has been implemented in the `compdata` Haskell library available on Hackage (Bahr and Hvitved 2014). As the implementation relies essentially on closed type families, it requires the Glasgow Haskell Compiler (GHC) version 7.8.

2. Data Types à la Carte

2.1 Defining Types and Functions

Data types à la carte (Swierstra 2008) is based on the idea of splitting a data type definition into a signature functor f and a knot-tying type constructor Fix such that $\mathit{Fix}\ f$ represents the original data type:

$$\mathbf{data}\ \mathit{Fix}\ f = \mathit{In}\ (f\ (\mathit{Fix}\ f))$$

The benefit of this representation is that it reduces the problem of extending recursive data types to the problem of extending functors. The latter is easily achieved by the sum construction:

$$\mathbf{data}\ (f :+: g)\ a = \mathit{Inl}\ (f\ a) \mid \mathit{Inr}\ (g\ a)$$

For example, instead of defining a data type of simple arithmetic expressions by a recursive data type

$$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

we define the functor

$$\mathbf{data}\ \mathit{Arith}\ a = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ a\ a$$

and build the desired data type by taking the fixpoint of Arith:

$$\mathbf{type}\ \mathit{Expr} = \mathit{Fix}\ \mathit{Arith}$$

Arith is the *signature* of the type Expr.

At a later point we can then extend Expr, e.g. with multiplication, using the sum operator:

$$\mathbf{data}\ \mathit{Mult}\ a = \mathit{Mult}\ a\ a$$
$$\mathbf{type}\ \mathit{Expr}' = \mathit{Fix}\ (\mathit{Arith} :+: \mathit{Mult})$$

Functions on data types à la carte follow the same two-level approach as the type definitions. Instead of defining functions by recursion, they are defined as a fold of an algebra. That is, to define a function of type $\mathit{Fix}\ f \to c$, we define a function of type $f\ c \to c$, called *algebra*, and lift it to the desired type by the following combinator:

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (f\ c \to c) \to \mathit{Fix}\ f \to c\\ &\mathit{fold}\ f = f\ (\mathit{fmap}\ (\mathit{fold}\ f)\ x)\end{aligned}$$

The definitions of algebras then follow the compositional structure of signatures. To this end, one defines a type class and instantiates it for each signature separately. For instance, assume that we want to define an evaluation function for Expr'. We first define a type class Eval, which contains an algebra of the appropriate type:

$$\begin{aligned}&\mathbf{class}\ \mathit{Eval}\ f\ \mathbf{where}\\ &\quad \mathit{evalAlg} :: f\ \mathit{Int} \to \mathit{Int}\end{aligned}$$

We then define the algebras for each of the *atomic signatures* by instantiating Eval:

$$\begin{aligned}&\mathbf{instance}\ \mathit{Eval}\ \mathit{Arith}\ \mathbf{where}\\ &\quad \mathit{evalAlg}\ (\mathit{Val}\ n) \quad = n\\ &\quad \mathit{evalAlg}\ (\mathit{Add}\ x\ y) = x + y\\ &\mathbf{instance}\ \mathit{Eval}\ \mathit{Mult}\ \mathbf{where}\\ &\quad \mathit{evalAlg}\ (\mathit{Mult}\ x\ y) = x * y\end{aligned}$$

We then lift the algebras to *compound signatures*:

$$\begin{aligned}&\mathbf{instance}\ (\mathit{Eval}\ f, \mathit{Eval}\ g) \Rightarrow \mathit{Eval}\ (f :+: g)\ \mathbf{where}\\ &\quad \mathit{evalAlg}\ (\mathit{Inl}\ x) = \mathit{evalAlg}\ x\\ &\quad \mathit{evalAlg}\ (\mathit{Inr}\ x) = \mathit{evalAlg}\ x\end{aligned}$$

Eventually, we obtain the following modular definition of an evaluation function:

$$\begin{aligned}&\mathit{eval} :: (\mathit{Eval}\ f, \mathit{Functor}\ f) \Rightarrow \mathit{Fix}\ f \to \mathit{Int}\\ &\mathit{eval} = \mathit{fold}\ \mathit{evalAlg}\end{aligned}$$

Due to its modular definition, eval can be instantiated to work on both Expr and Expr':

$$\begin{aligned}&\mathit{eval}_1 :: \mathit{Expr} \to \mathit{Int}\\ &\mathit{eval}_1 = \mathit{eval}\\ &\mathit{eval}_2 :: \mathit{Expr}' \to \mathit{Int}\\ &\mathit{eval}_2 = \mathit{eval}\end{aligned}$$

This ability to define functions on data types à la carte in a modular fashion is complemented with the ability to build and deconstruct values of such modular data types, which we shall describe in section 2.2 below. The contributions of this paper lie in this latter part of the infrastructure. However, as we shall see, the added expressiveness in constructing and deconstructing data types provides new ways of defining and combining functions on data types à la carte.

2.2 Signature Subsumption

In order to construct and deconstruct values, data types à la carte provides a binary type class $:\prec:$ on signatures that expresses that a signature is subsumed by another one, e.g.

$$\mathit{Arith} :\prec: (\mathit{Arith} :+: \mathit{Mult})$$

The type class $:\prec:$ is equipped with methods that can be used to define the following two functions that enable the programmer to construct and deconstruct values from a *compound* data type:

$$\begin{aligned}&\mathit{inject} \ \ :: (f :\prec: g) \Rightarrow f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g\\ &\mathit{project} :: (f :\prec: g) \Rightarrow \mathit{Fix}\ g \to \mathit{Maybe}\ (f\ (\mathit{Fix}\ g))\end{aligned}$$

For example, we can use inject, to lift the constructor Val to any type that at least contains Arith, which yields the following *smart constructor*:

$$val :: (Arith :\prec: f) \Rightarrow Int \rightarrow Fix\ f$$
$$val\ i = inject\ (Val\ i)$$

Similarly, we can use $project$ to pattern match any value of a type that contains $Arith$ against the Val constructor:

$$\begin{aligned} &getVal :: (Arith :\prec: f) \Rightarrow Fix\ f \rightarrow Maybe\ Int \\ &getVal\ i = \mathbf{case}\ project\ i\ \mathbf{of} \\ &\qquad Just\ (Val\ i) \rightarrow Just\ i \\ &\qquad \_ \qquad\qquad\quad \rightarrow Nothing \end{aligned}$$

To understand the behaviour of data types à la carte, we have to look at the definition of the type class $:\prec:$ and its instance declarations. The class declares two methods that form the underpinning of the implementation of $inject$ and $project$:

$$\begin{aligned} &\mathbf{class}\ f :\prec: g\ \mathbf{where} \\ &\quad inj :: f\ a \rightarrow \qquad\quad g\ a \\ &\quad prj :: g\ a \rightarrow Maybe\ (f\ a) \end{aligned}$$

The functions $inject$ and $project$ are defined in terms of these methods as follows:

$$\begin{aligned} &inject :: (f :\prec: g) \Rightarrow f\ (Fix\ g) \rightarrow Fix\ g \\ &inject = In \circ inj \\ &project :: (f :\prec: g) \Rightarrow Fix\ g \rightarrow Maybe\ (f\ (Fix\ g)) \\ &project\ (In\ g) = prj\ g \end{aligned}$$

What makes the type class $:\prec:$ work are the instance declarations:

$$\begin{aligned} &\mathbf{instance}\ f :\prec: f\ \mathbf{where} \\ &\quad inj = id \\ &\quad prj = Just \\ &\mathbf{instance}\ f :\prec: (f :+: g)\ \mathbf{where} \\ &\quad inj = Inl \\ &\quad prj\ (Inl\ f) = Just\ f \\ &\quad prj\ (Inr\ g) = Nothing \\ &\mathbf{instance}\ (f :\prec: g) \Rightarrow f :\prec: (h :+: g)\ \mathbf{where} \\ &\quad inj = Inr \circ inj \\ &\quad prj\ (Inr\ g) = prj\ g \\ &\quad prj\ (Inl\ h) = Nothing \end{aligned}$$

For the moment it is not important how inj and prj are implemented. More interesting is how we obtain instances of $f :\prec: g$. Of particular importance is the apparent asymmetry of the treatment of the $:+:$ operator: while $:\prec:$ is defined recursively for the right-hand side of $:+:$, we only have a non-recursive instance declaration for the left-hand side of $:+:$.

As a consequence, $:\prec:$ can be characterised syntactically as follows: we have an instance $f :\prec: g$ iff g is of the form

$$g_1 :+: (\ldots :+: (g_{n-1} :+: g_n) \ldots)$$

and $f = g_i$ for some $1 \leq i \leq n$. To match this behaviour of $:\prec:$, the operator $:+:$ is declared right-associative. Hence, we have that $f :\prec: g$ iff g is of the form $g_1 :+: \ldots :+: g_n$ and $f = g_i$ for some $1 \leq i \leq n$. However, we have to be careful as we do not have the following subsumption for example:

$$f_1 :+: f_2 :\prec: f_1 :+: f_2 :+: f_3$$

The problem is that the right-hand side is parenthesised as $f_1 :+: (f_2 :+: f_3)$. The common workaround for this problem is to split constraints of the form $f_1 :+: f_2 :\prec: g$ into two constraints: $f_1 :\prec: g$ and $f_2 :\prec: g$.

In summary: if we want to use $:\prec:$ in order to express signature subsumption, we have to make sure that the left-hand side is an atomic signature (i.e. not formed by $:+:$) and that the right-hand side is a right-associative sum.

In many use cases, these limitations of $:\prec:$ are unproblematic or can be worked around. However, as we shall demonstrate, these limitations do cause trouble for many other realistic use cases.

Let's start with the restriction that signatures on the right-hand side of $:\prec:$ must be right-associative sums. While, this seems innocuous at first, it clashes with abstraction and compositionality. For example, given two concrete signatures Foo and Bar, we may want to form a new signature $FooBar$ by summation as follows:

$$\mathbf{type}\ FooBar = Foo :+: Bar$$

However, if Foo was itself defined as the sum $A :+: B$, then we would not have that $A :\prec: FooBar$. Hence, none of the smart constructors of A can be used to construct a value of type $Fix\ FooBar$. In order to obtain this subsumption relation, we would have to define $FooBar$ in the following way, which breaks abstraction:

$$\mathbf{type}\ FooBar = A :+: B :+: Bar$$

Furthermore, this restriction to right-associative sums hinders refactoring. For instance, we might want to refactor the definition of the signature $Arith$ into two parts as follows:

$$\begin{aligned} &\mathbf{data}\ Val\ a\ = Val\ Int \\ &\mathbf{data}\ Add\ a = Add\ a\ a \\ &\mathbf{type}\ Arith\ = Val :+: Add \end{aligned}$$

In practice, such refactoring may become necessary in order to avoid duplication. For example, we could now define a type of values, e.g. to define an evaluation function:

$$\mathbf{type}\ Value = Fix\ Val$$

The types of the smart constructors for Val and Add are refactored accordingly, e.g.

$$val :: (Val :\prec: f) \Rightarrow Int \rightarrow Fix\ f$$

However, with this refactoring we do not have the anticipated subsumption relation $Val :\prec: Arith :+: Mult$, which renders the smart constructor val useless for constructing expressions of type $Expr'$. However, we do have the instance $Val :\prec: Mult :+: Arith$. This counterintuitive behaviour is caused by the asymmetry in the instance declarations for $:\prec:$.

Also the restriction to atomic signatures on the left-hand side of $:\prec:$ appears harmless at first. For example, smart constructors, such as val defined above, always follow this pattern. Similarly, the $project$ function is typically used for pattern matching, and thus atomic left-hand sides are sufficient.

However, there are use cases that do require compound signatures on the left hand side. To illustrate this, we give a recursive variant of $inject$, which can be considered an upcasting operation:

$$\begin{aligned} &deepInject :: (f :\prec: g, Functor\ f) \Rightarrow Fix\ f \rightarrow Fix\ g \\ &deepInject = fold\ inject \end{aligned}$$

The function $deepInject$ uses the injection derived from $f :\prec: g$ to upcast a complete value from signature f to signature g. For example we could imagine having an expression over integer literals and multiplication, i.e. of type $Fix\ (Val :+: Mult)$ and want to turn it into an expression of type $Expr'$. We could use $deepInject$ to do so, provided that $Val :+: Mult :\prec: Arith :+: Mult$. Alas, this is not the case, even though we have that $Val :\prec: Arith$.

Similarly we can define a function $deepProject$ that performs a downcasting operation (Bahr and Hvitved 2011). Its utility is unfortunately equally reduced due to the limitation of $:\prec:$.

Another shortcoming of the present implementation of signature subsumption can be seen in the type of the method prj:

$$prj :: g\ a \rightarrow Maybe\ (f\ a)$$

This method tries to cast a value of type $g\ a$ to the "smaller" type $f\ a$, returning $Nothing$ if it fails. However, returning $Nothing$ is unsatisfying in some settings. If a value of type $g\ a$ cannot be cast to the type $f\ a$, we would like a proof of that in the form of a value of type $h\ a$, where $f \simeq g :+: h$. Given the signature $Arith :+: Mult$, for example, we would like prj to have type

$$(Arith :+: Mult)\ a \to Either\ (Val\ a)\ ((Add :+: Mult)\ a)$$

instead of just

$$(Arith :+: Mult)\ a \to Maybe\ (Val\ a)$$

This refined projection method could, for example, be used to implement an evaluation function. We first split out the value part of the input signature, which is evaluated trivially, and then deal with the remainder of the signature – where actual evaluation is necessary – separately. In general, a more powerful projection function as outlined above could be used to define extensible functions in an ad hoc manner, without the need to use type classes. We shall see an example of such an ad hoc definition in the form of a desugaring function in section 5.2.1.

In the remainder of this paper we present an alternative implementation that resolves the issues we have described above. The implementation is presented in three steps from section 4 to section 6. Since this implementation uses a fairly recent extension to the Haskell language – closed type families – we give a brief introduction to this new feature in section 3. Readers familiar with closed type families in Haskell can safely skip that section.

3. Using Closed Type Families

Type families (Chakravarty et al. 2005) extend the type language of Haskell to allow the programmer to express limited forms of type-level computation:

$$\begin{array}{l}
\mathbf{type\ family}\ Element\ l \\
\mathbf{type\ instance}\ Element\ [a] = a \\
\mathbf{type\ instance}\ Element\ Text = Char
\end{array}$$

In the code above we first declare the type family $Element$ that takes a single type l as an argument and returns a type. Then we give two instances of this type family by giving appropriate mappings. Any list type $[a]$ is mapped to the type a, and the type $Text$ is mapped to the type $Char$.

Type families are by nature *partial*, they do not necessarily provide a mapping for each type. For example the type family $Element$ does not provide a mapping for types of the form $Array\ a$. But type families are also *open*. That is, we can extend the definition – without recompiling the original code – by another mapping, e.g.

$$\mathbf{type\ instance}\ Element\ (Array\ a) = a$$

This openness of type families makes them quite different from Haskell functions on the value level.

Recently, Eisenberg et al. (2014) introduced *closed* type families and implemented them in the Glasgow Haskell Compiler (GHC). In contrast to their open counterparts, closed type families are defined with a fixed *sequence* of equations that cannot be extended. Moreover, the order of the equations is relevant – similarly to function definitions in Haskell. For example, the following code defines a type family $Curry$ that curries a function type of the appropriate form and otherwise does nothing:

$$\begin{array}{l}
\mathbf{type\ family}\ Curry\ t\ \mathbf{where} \\
\quad Curry\ ((a, b) \to c) = a \to b \to c \\
\quad Curry\ a \qquad\qquad\quad = a
\end{array}$$

Note that the two equations are overlapping, e.g. they both apply to the type $(Int, String) \to Char$. But the equations are tried in order, and the first applicable equation is chosen. Hence, $Curry\ ((Int, String) \to Char)$ is simplified to $(Int, String) \to Char$. However, the semantics of closed type families is subtle. For example, given a type variable t, the type $Curry\ t$ does *not* simplify to t as one might at first expect. The equations in a type family are tried from top to bottom. But it is not sufficient that the left-hand side matches in order to make the equation applicable. In addition, it is required that none of the equations that appear before it can match – for any instantiation of type variables.[1] The example type t can be instantiated such that it matches the first equation, namely by instantiating t to $(a, b) \to c$. Therefore, $Curry\ t$ does not simplify to t. By contrast, $Curry\ (s, t)$ does indeed simplify to (s, t).

But closed type families go even beyond simple pattern matching by also allowing non-linear patterns, i.e. type variables may occur more than once on the left-hand side of equations. For example we may define the following type family that turns any product type of the form (a, a) into a function type $Bool \to a$:

$$\begin{array}{l}
\mathbf{type\ family}\ Prod\ t\ \mathbf{where} \\
\quad Prod\ (a, a) = Bool \to a \\
\quad Prod\ a \qquad = a
\end{array}$$

Closed type families are particularly handy for dealing with types produced by data type promotion (Yorgey et al. 2012), which lifts (a limited class of) data types to the kind level. For example the data type definition for $Bool$

$$\mathbf{data}\ Bool = True \mid False$$

yields also two types $True$ and $False$, each of kind $Bool$. We can then define type families on types of kind $Bool$ as we would define functions on type $Bool$. For example, we can define disjunction:

$$\begin{array}{l}
\mathbf{type\ family}\ Or\ a\ b\ \mathbf{where} \\
\quad Or\ False\ False = False \\
\quad Or\ a \qquad b \qquad = True
\end{array}$$

As for open type families, we can provide explicit kind annotations to closed type family definitions:

$$\begin{array}{l}
\mathbf{type\ family}\ Or\ (a :: Bool)\ (b :: Bool) :: Bool\ \mathbf{where} \\
\quad Or\ False\ False = False \\
\quad Or\ a \qquad b \qquad = True
\end{array}$$

An important fact to keep in mind is that the computations performed via type families happen all during compile time. Moreover, the results of these computations are not available during runtime. This complicates writing functions and terms that inhabit the types computed by type families. For instance, reconsider the type family $Prod$ that we defined above. It maps any type of the form (a, a) to the type $Bool \to a$ and any other type to itself. Thus, we have that any type t is isomorphic to $Prod\ t$. In particular, we should be able to write a function of type $t \to Prod\ t$ that implements one direction of this isomorphism. However, the straightforward attempt to implement this function fails:

$$\begin{array}{l}
prod :: t \to Prod\ t \\
prod\ (x, y) = \lambda b \to \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y \\
prod\ x \qquad = x
\end{array}$$

GHC will complain that it

```
couldn't match expected type 't' with actual
  type '(t0, t0)'
```

What we really want is to pattern match on the type t to check whether it is of the form (a, a) and then return an according

[1] In practice, GHC only approximates this idea conservatively using the notion of *apartness* (cf. Eisenberg et al. (2014)).

mapping from t to $Prod\ t$. There are two methods to achieve this in Haskell: use a GADT that reflects the type-level evidence to the term level, or use a type class to dispatch on the result of the type level computation.

For the first approach we introduce a GADT that reflects the pattern matching we would like to perform on the input type t:

```
data Ty t t' where
  IsProd  :: Ty (a, a) (Bool → a)
  NotProd :: Ty t       t
```

The first argument to Ty is the type we want to pattern match on and the second argument is the result of applying $Prod$ to that type. In other words, the inhabitants of type $Ty\ t\ t'$ are evidence that $Prod\ t = t'$. We can then write the desired function by pattern matching on this evidence:

```
prod' :: Ty t (Prod t) → t → Prod t
prod' IsProd  (x, y) = λb → if b then x else y
prod' NotProd x      = x
```

We can then use a type class to infer the evidence automatically:

```
class GetTy t t' where
  getTy :: Ty t t'
instance GetTy (a, a) (Bool → a) where
  getTy = IsProd
instance GetTy a a where
  getTy = NotProd
```

Finally, we obtain the definition of the function $prod$ by applying $prod'$ to the evidence provided by the function $getTy$:

```
prod :: GetTy t (Prod t) ⇒ t → Prod t
prod = prod' getTy
```

This approach is described by Eisenberg et al. (2014) in their implementation of a generic $zipWith$ function. However, the construction of explicit term-level evidence is unnecessary as it is immediately consumed by $prod'$. Instead, we can use the type class $GetTy$ to directly construct the function $prod'\ getTy$:

```
class GetTy s t where
  prod' :: s → t
instance GetTy (a, a) (Bool → a) where
  prod' (x, y) = λb → if b then x else y
instance GetTy a a where
  prod' x = x
prod :: GetTy t (Prod t) ⇒ t → Prod t
prod = prod'
```

Apart from being clearer, this approach also avoids the additional pattern matching on the type Ty. The overhead due to this pattern matching is negligible in this toy example. But as term-level evidence becomes more complex, the overhead from pattern matching may become significant. Therefore, we shall use direct approach for the rest of this paper.

4. Implementing Backtracking Subsumption

The fundamental problem that we need to solve to improve the definition of :≺: is to make it closed under summation from the left and right. If we implement :≺: as a type class, we have to choose one over the other, since there is no mechanism to backtrack. That is, when checking whether f :≺: g_1 :+: g_2, we have to commit to either checking f :≺: g_1 or f :≺: g_2. Haskell's type class system does not allow us to try one and upon failure try the other.

To implement a backtracking variant of :≺: using closed type families, we implement a type family that takes two signature functors and checks whether the first is a summand of the second one. With the help of the type family Or defined in section 3, we can implement such a type family quite easily:

```
type family Elem (e :: * → *) (f :: * → *) :: Bool where
  Elem e e         = True
  Elem e (l :+: r) = Or (Elem e l) (Elem e r)
  Elem e f         = False
```

The constraint :≺: can then be implemented by defining the following synonym:

```
type f :≺: g = Elem f g ~ True
```

That is, f is subsumed by g iff $Elem\ f\ g$ is equal to $True$. The above definition makes use of the `ConstraintKinds` extension of Haskell to define f :≺: g as a synonym for $Elem\ f\ g \sim True$.

However, the above definition only covers one aspect of the original definition of :≺:. The original type class :≺: also provided two functions inj and $proj$. With the above setup alone we do not have any concrete type-level evidence to implement these two functions. Instead of only producing a Boolean as a result, the type family $Elem$ must also provide evidence of the fact that the first argument is contained in the second argument. We will represent such evidence by the following kind Pos, which intuitively denotes the position of an occurrence found by $Elem$:

```
data Pos = Here | Left Pos | Right Pos
```

Note that we make use of Haskell's data type promotion facility (Yorgey et al. 2012) to use Pos as a kind. For example, $Left$ is used as a type constructor of kind $Pos \to Pos$.

Instead of using the kind $Bool$, we then use the following kind Res, which provides the position of the occurrence found in the second argument of $Elem$:

```
data Res = Found Pos | NotFound
```

The definition of $Elem$ is easily refactored to produce a type-level evidence of kind Res:

```
type family Elem (e :: * → *) (p :: * → *) :: Res where
  Elem e e         = Found Here
  Elem e (l :+: r) = Choose (Elem e l) (Elem e r)
  Elem e p         = NotFound
type family Choose (l :: Res) (r :: Res) :: Res where
  Choose (Found x) y         = Found (Left x)
  Choose x         (Found y) = Found (Right y)
  Choose x         y         = NotFound
```

We replace the type family Or by the type family $Choose$, which produces an appropriate type-level evidence.

Using the result produced by $Elem$, we can derive the inj and prj function. Following the approach outlined in section 3 we define the following type class:

```
class Subsume (res :: Res) f g where
  inj' :: f a → g a
  prj' :: g a → Maybe (f a)
```

$Subsume$ is the same as :≺: from section 2, except that it has an additional type parameter of kind Res. With this setup we can define the instance declarations that we want, namely by recursion in the left- and the right hand-side of :+:. The additional argument of kind Res acts as an oracle that tells Haskell's type instance resolution which instance declaration to take.

Unfortunately, we cannot use the type class $Subsume$ as it is defined above since the type res does not occur in the type of either class methods inj' and prj'. The solution is simple, though: we add a dummy argument that mentions the type:

$$\textbf{data}\ \mathit{Proxy}\ a = P$$

$$\begin{array}{l}\textbf{class}\ \mathit{Subsume}\ (\mathit{res} :: \mathit{Res})\ f\ g\ \textbf{where}\\ \quad \mathit{inj}' :: \mathit{Proxy}\ \mathit{res} \to f\ a \to g\ a\\ \quad \mathit{prj}' :: \mathit{Proxy}\ \mathit{res} \to g\ a \to \mathit{Maybe}\ (f\ a)\end{array}$$

Providing instance declarations is easy now. The declarations follow the same idea as the original definition of $:\prec:$ from section 2. The only exception is that the case for the left summand is now analogous to the case for the right summand:

$$\begin{array}{l}\textbf{instance}\ \mathit{Subsume}\ (\mathit{Found}\ \mathit{Here})\ f\ f\ \textbf{where}\\ \quad \mathit{inj}'\ \_ = \mathit{id}\\ \quad \mathit{prj}'\ \_ = \mathit{Just}\\ \textbf{instance}\ \mathit{Subsume}\ (\mathit{Found}\ p)\ f\ l\\ \quad \Rightarrow \mathit{Subsume}\ (\mathit{Found}\ (\mathit{Left}\ p))\ f\ (l :+: r)\ \textbf{where}\\ \quad \mathit{inj}'\ \_ = \mathit{Inl} \circ \mathit{inj}'\ (P :: \mathit{Proxy}\ (\mathit{Found}\ p))\\ \quad \mathit{prj}'\ \_\ (\mathit{Inl}\ x) = \mathit{prj}'\ (P :: \mathit{Proxy}\ (\mathit{Found}\ p))\ x\\ \quad \mathit{prj}'\ \_\ (\mathit{Inr}\ \_) = \mathit{Nothing}\\ \textbf{instance}\ \mathit{Subsume}\ (\mathit{Found}\ p)\ f\ r\\ \quad \Rightarrow \mathit{Subsume}\ (\mathit{Found}\ (\mathit{Right}\ p))\ f\ (l :+: r)\ \textbf{where}\\ \quad \mathit{inj}'\ \_ = \mathit{Inr} \circ \mathit{inj}'\ (P :: \mathit{Proxy}\ (\mathit{Found}\ p))\\ \quad \mathit{prj}'\ \_\ (\mathit{Inr}\ x) = \mathit{prj}'\ (P :: \mathit{Proxy}\ (\mathit{Found}\ p))\ x\\ \quad \mathit{prj}'\ \_\ (\mathit{Inl}\ \_) = \mathit{Nothing}\end{array}$$

The subsumption constraint $:\prec:$ is then defined as follows:

$$\textbf{type}\ f :\prec: g = \mathit{Subsume}\ (\mathit{Elem}\ f\ g)\ f\ g$$

This allows us to define the final injection and projection functions as follows:

$$\begin{array}{l}\mathit{inj} :: \forall f\ g\ a\ .\ (f :\prec: g) \Rightarrow f\ a \to g\ a\\ \mathit{inj} = \mathit{inj}'\ (P :: \mathit{Proxy}\ (\mathit{Elem}\ f\ g))\\ \mathit{prj} :: \forall f\ g\ a\ .\ (f :\prec: g) \Rightarrow g\ a \to \mathit{Maybe}\ (f\ a)\\ \mathit{prj} = \mathit{prj}'\ (P :: \mathit{Proxy}\ (\mathit{Elem}\ f\ g))\end{array}$$

With this implementation we indeed obtain subsumption relations of the form[2]

$$g :\prec: (f :+: g) :+: h$$

For instance, in the example from section 2, we have the anticipated subsumption $\mathit{Val} :\prec: \mathit{Arith} :+: \mathit{Mult}$. Recall that in the type class-based implementation, we did not have this subsumption, but we did have the subsumption $\mathit{Val} :\prec: \mathit{Mult} :+: \mathit{Arith}$. With the above closed type families-based implementation we get both.

However, this new implementation still suffers from the same problem of ambiguity as the original type class-based one: we can still derive subsumptions that permit more than one injection function, e.g. $f :\prec: f :+: f$. Such subsumption relations are typically unintended and we should try to avoid them and instead provide an error message to the programmer to inform her about the ambiguity. For instance, we may forget that the Arith signature already contains the Val signature and try to derive the subsumption $\mathit{Val} :\prec: \mathit{Arith} :+: \mathit{Val}$.

The implementation we have given in this section can be easily extended to check for ambiguity. Firstly, we have to extend the kind Res by another type to indicate ambiguity:

$$\textbf{data}\ \mathit{Res} = \mathit{Found}\ \mathit{Pos} \mid \mathit{NotFound} \mid \mathit{Ambiguous}$$

Secondly, we extend the definition of the type family Choose by three additional equations:

$$\begin{array}{llll}\multicolumn{4}{l}{\textbf{type family}\ \mathit{Choose}\ (l :: \mathit{Res})\ (r :: \mathit{Res}) :: \mathit{Res}\ \textbf{where}}\\ \quad \mathit{Choose} & (\mathit{Found}\ x) & (\mathit{Found}\ y) & = \mathit{Ambiguous}\\ \quad \mathit{Choose} & \mathit{Ambiguous} & y & = \mathit{Ambiguous}\\ \quad \mathit{Choose} & x & \mathit{Ambiguous} & = \mathit{Ambiguous}\\ \quad \mathit{Choose} & (\mathit{Found}\ x) & y & = \mathit{Found}\ (\mathit{Left}\ x)\\ \quad \mathit{Choose} & x & (\mathit{Found}\ y) & = \mathit{Found}\ (\mathit{Right}\ y)\\ \quad \mathit{Choose} & x & y & = \mathit{NotFound}\end{array}$$

[2] The signatures on either sides have to be ground, though. This issue is discussed in detail in section 7.1.

The first equation detects ambiguities, while the second and third equation propagate any ambiguity that we have found. The remaining equations are the same ones we had before. Also the other definitions stay the same.

With the thus amended definition, we indeed avoid ambiguous embeddings from multiple occurrences of the same summand. For instance, the constraint $\mathit{Val} :\prec: \mathit{Arith} :+: \mathit{Val}$ is no longer satisfied and is rejected with the error message

```
No instance for
  (Subsume Ambiguous Val (Arith :+: Val))
```

Rejecting ambiguous subsumptions is not necessary. The law that we would expect the derived functions inj and prj to satisfy (Delaware et al. 2013) can be formulated as follows:

$$\mathit{prj}\ x = \mathit{Just}\ y \quad \text{iff} \quad \mathit{inj}\ y = x \qquad (\textsc{Inverse})$$

Swierstra's original implementation as well as the implementation given here (be it with checking for ambiguity or not) satisfy this law. Nonetheless we argue that ambiguity is typically undesired and should be considered an error.

The implementation that we presented in this section resolves some of the issues that we have identified in section 2. In particular, the implementation treats $:+:$ symmetrically, and it avoids ambiguous injections. However, it still does not allow arbitrary sums on the left hand side. For example, we cannot derive the following subsumption:

$$\mathit{Add} :+: \mathit{Mult} :\prec: \mathit{Arith} :+: \mathit{Mult}$$

We should be able to derive the above subsumption since Arith subsumes Add. However, our implementation as well as the original implementation of Swierstra can only derive a subsumption if the left-hand signature appears as a summand in the right-hand side signature. In the next section we further refine our implementation to deal with this case.

5. Subsumption for Compound Signatures

In this section we generalise the implementation of the subsumption constraint $:\prec:$ to allow compound signatures on the left-hand side. This generalisation proves useful for a number of use cases. In particular, it will allow us to define an isomorphism constraint $:\simeq:$ on signatures. In this section we will give a straightforward implementation of $:\prec:$ that has these properties. In section 6, we shall give a revised implementation that provides better error messages and has better performance properties.

5.1 Decomposing Compound Signatures

Our first approach to generalise the subsumption constraint implemented in section 4 to compound signatures on the left-hand side follows a simple recipe: (1) decompose the left-hand side signature into its atomic summands, and (2) use the subsumption constraint from section 4 on these atomic summands.

The idea is to decompose the left-hand side signature f in a constraint $f :\prec: g$ and then try to obtain an embedding using $\mathit{Elem}\ f'\ g$ for each component f' of f. To this end we introduce the following kind Struc, which describes the structure of a (potentially) compound signature and provides types of kind Res for each atomic component in that structure:

$$\textbf{data}\ \mathit{Struc} = \mathit{Sum}\ \mathit{Struc}\ \mathit{Struc} \mid \mathit{Atom}\ \mathit{Res}$$

The following type family $GetStruc$ performs the decomposition on its first argument and refers to $Elem$ once it has found an atomic signature:

```
type family GetStruc f g :: Struc where
  GetStruc (f₁ :+: f₂) g = Sum (GetStruc f₁ g)
                               (GetStruc f₂ g)
  GetStruc f           g = Atom (Elem f g)
```

As before, we use a type class that traverses the evidence produced by $GetStruc$ in order to define the desired injection and projection functions:

```
class Subsume′ (s :: Struc) f g where
  inj′′ :: Proxy s → f a → g a
  prj′′ :: Proxy s → g a → Maybe (f a)
instance Subsume res f g
    ⇒ Subsume′ (Atom res) f g where
  inj′′ _ x = inj′ (P :: Proxy res) x

  prj′′ _ x = prj′ (P :: Proxy res) x
instance (Subsume′ s₁ f₁ g, Subsume′ s₂ f₂ g)
    ⇒ Subsume′ (Sum s₁ s₂) (f₁ :+: f₂) g where
  inj′′ _ (Inl x) = inj′′ (P :: Proxy s₁) x
  inj′′ _ (Inr x) = inj′′ (P :: Proxy s₂) x
  prj′′ _ x = case prj′′ (P :: Proxy s₁) x of
    Just y → Just (Inl y)
    _      → case prj′′ (P :: Proxy s₂) x of
               Just y   → Just (Inr y)
               Nothing  → Nothing
```

For the case of an atomic signature we use the injection and projection from the corresponding instance of $Subsume$. Whereas in the case of a sum we recurse.

We can then redefine the subsumption constraint $:\prec:$ as follows:

```
type f :≺: g = Subsume′ (GetStruc f g) f g
```

The injection and projection functions are redefined accordingly

```
inj :: ∀ f g a . (f :≺: g) ⇒ f a → g a
inj = inj′′ (P :: Proxy (GetStruc f g))
prj :: ∀ f g a . (f :≺: g) ⇒ g a → Maybe (f a)
prj = prj′′ (P :: Proxy (GetStruc f g))
```

Now we are finally able to derive non-trivial subsumptions with a compound left-hand side, e.g.

$$Val :+: Mult :\prec: Arith :+: Mult$$

For example, we can use the $deepInject$ function from section 2.2 to upcast any expression over $Val :+: Mult$ into an expression over $Arith :+: Mult$:

```
upcast :: Fix (Val :+: Mult) → Fix (Arith :+: Mult)
upcast = deepInject
```

However, this implementation of $:\prec:$ is still not fully satisfactory. Our implementation avoids ambiguity caused by subsumptions with multiple occurrences of the same signature on the right-hand side, e.g. $Val :\prec: Val :+: Val$. Since we now allow compound signatures on the left-hand side, the converse may happen: our implementation happily derives that $Val :+: Val :\prec: Val$.

This phenomenon is qualitatively worse than ambiguity, since it means that the derived functions inj and prj do not satisfy the INVERSE law. In particular, inj is not injective. The solution to this problem is simple: we add another constraint to the definition of $:\prec:$ that checks whether the left-hand side contains duplicates. Figure 1 contains the implementation of the type family $Dupl$, which checks for duplicate occurrences of the same atomic signature in a given signature. To this end, $Dupl$ takes an additional worklist parameter of kind $[* \to *]$, i.e. a list of signatures. $Dupl$ proceeds by decomposing the argument, recursing on the left summand and adding the right summand to the worklist. Once it reaches an atomic signature, it checks whether this atomic signature occurs in one of the signatures in the work list. Moreover, it repeats the check for every signature in the worklist. Note that $':$ and $'[]$ denote the constructors for type-level lists.

```
type family Dupl (f :: ∗ → ∗) (l :: [∗ → ∗]) :: Bool where
  Dupl (f :+: g) l = Dupl f (g ′: l)
  Dupl f         l = Or (Find f l) (Dupl′ l)
type family Dupl′ (l :: [∗ → ∗]) :: Bool where
  Dupl′ (f ′: l)  = Or (Dupl f l) (Dupl′ l)
  Dupl′ ′[]       = False
type family Find (f :: ∗ → ∗) (l :: [∗ → ∗]) :: Bool where
  Find f (g ′: l) = Or (Find′ f g) (Find f l)
  Find f ′[]      = False
type family Find′ (f :: ∗ → ∗) (g :: ∗ → ∗) :: Bool where
  Find′ f (g₁ :+: g₂) = Or (Find′ f g₁) (Find′ f g₂)
  Find′ f f           = True
  Find′ f g           = False
```

Figure 1. Checking for duplicate occurrences of signatures.

We can thus refine the implementation of $:\prec:$ as follows:

```
type f :≺: g = (Subsume′ (GetStruc f g) f g,
                Dupl f ′[] ∼ False)
```

With this new definition, subsumptions such as $Val :+: Val :\prec: Val$ are not derivable anymore.

5.2 Signature Isomorphisms

The added generality of $:\prec:$ brings a new set of use cases for data types à la carte. As we illustrated in section 2.2, the type of the projection function prj is somewhat unsatisfying: given $f :\prec: g$, the projection prj either returns a value over signature f or it returns $Nothing$. However, if projection into f fails, we have learned that the input must be coercible into a signature h with $g \simeq f :+: h$.

The new implementation of $:\prec:$ allows us to do just that by giving us a means to express $g \simeq f :+: h$ constructively. In particular, we can define the binary constraint $:\simeq:$ on signatures:

```
type f :≃: g = (f :≺: g, g :≺: f)
```

That is, we define signature isomorphism as subsumption in both directions.

We can now express that a signature f can be split into two disjoint sub-signatures f_1 and f_2 as the constraint $f :\simeq: f_1 :+: f_2$. The following function $split$ will allow us to do pattern matching according to such a decomposition into two disjoint sub-signatures:

```
split :: (f :≃: f₁ :+: f₂) ⇒
         (f₁ a → b) → (f₂ a → b) → f a → b
split f₁ f₂ x = case inj x of
                  Inl y → f₁ y
                  Inr y → f₂ y
```

Note that we, in fact, only need one of the two subsumptions that make up the isomorphism constraint in order to define $split$, namely $f :\prec: f_1 :+: f_2$. The inj function for this subsumption allows us to map $f\ a$ into $(f_1 :+: f_2)\ a$. The converse subsumption is only needed to make sure that f_1 and f_2 do not contain any "junk", i.e. signatures that are not already present in f.

$$
\begin{array}{l}
\textbf{class } \mathit{Desug}\ f\ g\ \textbf{where} \\
\quad \mathit{desugAlg} :: f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g \\
\textbf{instance } (\mathit{Add} :\prec: g) \Rightarrow \mathit{Desug}\ \mathit{Dbl}\ g\ \textbf{where} \\
\quad \mathit{desugAlg}\ (\mathit{Double}\ x) = \mathit{inject}\ (\mathit{Add}\ x\ x) \\
\textbf{instance } (\mathit{Desug}\ f_1\ g, \mathit{Desug}\ f_2\ g) \\
\qquad\qquad\qquad \Rightarrow \mathit{Desug}\ (f_1 :+: f_2)\ g\ \textbf{where} \\
\quad \mathit{desugAlg}\ (\mathit{Inl}\ x) = \mathit{desugAlg}\ x \\
\quad \mathit{desugAlg}\ (\mathit{Inr}\ x) = \mathit{desugAlg}\ x \\
\textbf{instance } (f :\prec: g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
\quad \mathit{desugAlg} = \mathit{inject} \\
\mathit{desugar} :: (\mathit{Desug}\ f\ g, \mathit{Functor}\ f) \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ g \\
\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}
\end{array}
$$

Figure 2. Desugaring using type classes.

5.2.1 Example: Desugaring

To illustrate the utility of the isomorphism constraint and in particular the *split* combinator, consider the following signature functor

$$\textbf{data } \mathit{Dbl}\ a = \mathit{Double}\ a$$

with the intended semantics that Double doubles its argument. This Double operator can be considered syntactic sugar for the arithmetic expression language $\mathit{Fix}\ \mathit{Arith}$, since we can translate $\mathit{Double}\ x$ into $\mathit{Add}\ x\ x$. So we should be able to implement a desugaring function of type $\mathit{Fix}\ f \to \mathit{Fix}\ g$ such that g is "f without Dbl" and g contains at least Add. Using the power of data types à la carte we can implement such a desugaring function.

To do so, however, we have to follow the pattern described in section 2.1, i.e. define a suitable type class and provide the necessary instance declarations. Figure 2 gives the detailed implementation. Moreover, the resulting type of the desugaring function will not immediately describe the relationship between the two signatures f and g. With the new isomorphism constraint $:\simeq:$ we can do better and give a function with the following type, without any additional type class infrastructure:

$$
\begin{array}{l}
\mathit{desugar} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Add} :\prec: g, \mathit{Functor}\ f) \\
\qquad\qquad \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ g
\end{array}
$$

The type signature explains the relationship between f and g in a direct and succinct way. The implementation itself is straightforward. However, we have to give type annotations in order to make explicit how f should we decomposed:

$$
\begin{array}{l}
\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg} \\
\mathit{desugAlg} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Add} :\prec: g) \\
\qquad\qquad \Rightarrow f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g \\
\mathit{desugAlg} = \mathit{split}\ (\lambda x \qquad\qquad \to \mathit{In}\ x) \\
\qquad\qquad\qquad (\lambda(\mathit{Double}\ x) \to \mathit{inject}\ (\mathit{Add}\ x\ x))
\end{array}
$$

The algebra that is used to implement the desugaring uses *split* to pattern match according to the isomorphism $f :\simeq: g :+: \mathit{Dbl}$. The first case of this pattern matching performs the trivial transformation via In whereas the second case performs the desired desugaring of Double.

5.2.2 Example: Overriding Default Implementation

Consider the implementation of a modular evaluation function eval shown in Figure 3. The implementation follows the typical pattern for defining a function on data types à la carte: a type class that provides the underlying algebra is declared, instances are declared for the sum construction and each atomic signature, and finally the function is defined as a fold over the thus defined modular algebra.

$$
\begin{array}{l}
\textbf{class } \mathit{Eval}\ f\ \textbf{where} \\
\quad \mathit{evalAlg} :: f\ \mathit{Int} \to \mathit{Int} \\
\textbf{instance } (\mathit{Eval}\ f, \mathit{Eval}\ g) \Rightarrow \mathit{Eval}\ (f :+: g)\ \textbf{where} \\
\quad \mathit{evalAlg}\ (\mathit{Inl}\ x) = \mathit{evalAlg}\ x \\
\quad \mathit{evalAlg}\ (\mathit{Inr}\ x) = \mathit{evalAlg}\ x \\
\textbf{instance } \mathit{Eval}\ \mathit{Add}\ \textbf{where} \\
\quad \mathit{evalAlg}\ (\mathit{Add}\ x\ y) = x + y \\
\textbf{instance } \mathit{Eval}\ \mathit{Dbl}\ \textbf{where} \\
\quad \mathit{evalAlg}\ (\mathit{Double}\ x) = x + x \\
\textbf{instance } \mathit{Eval}\ \mathit{Val}\ \textbf{where} \\
\quad \mathit{evalAlg}\ (\mathit{Val}\ n) = n \\
\mathit{eval} :: (\mathit{Eval}\ f, \mathit{Functor}\ f) \Rightarrow \mathit{Fix}\ f \to \mathit{Int} \\
\mathit{eval} = \mathit{fold}\ \mathit{evalAlg}
\end{array}
$$

Figure 3. Modular evaluation function.

This approach yields a modular and extensible function definition. However, the modularity is restricted as this setup does not allow us to replace one of the instance declarations. For example, if we wish to have an alternative evaluation function that evaluates $\mathit{Double}\ x$ to $2 * x$ instead of $x + x$, we have to define a separate type class, duplicate all instance declarations (except the one for Dbl) and provide a new instance declaration for Dbl that implements the alternative evaluation.

Using the *split* combinator, we can override the evaluation implementation for Dbl without writing a new evaluation function from scratch. To achieve this, we split the signature f into the form $g :+: \mathit{Dbl}$, use the default implementation for g, and provide a new implementation for Dbl:

$$
\begin{array}{l}
\mathit{eval}' :: \forall\ f\ g\ .\ (f :\simeq: g :+: \mathit{Dbl}, \mathit{Eval}\ g, \mathit{Functor}\ f) \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{Proxy}\ g \to \mathit{Fix}\ f \to \mathit{Int} \\
\mathit{eval}'\ \_ = \mathit{fold}\ \mathit{evalAlg}' \\
\quad \textbf{where } \mathit{evalAlg}' = \mathit{split}\ (\lambda(x :: g\ \mathit{Int}) \to \mathit{evalAlg}\ x) \\
\qquad\qquad\qquad\qquad\qquad (\lambda(\mathit{Double}\ x) \to 2 * x)
\end{array}
$$

Note that we have to provide the type g that is used in the split as an explicit type argument via a proxy. For example we can instantiate the above evaluation function to a concrete signature as follows:

$$
\begin{array}{l}
\mathit{evaluate} :: \mathit{Fix}\ (\mathit{Arith} :+: \mathit{Dbl}) \to \mathit{Int} \\
\mathit{evaluate} = \mathit{eval}'\ (P :: \mathit{Proxy}\ \mathit{Arith})
\end{array}
$$

While use of *split* in the above two examples produces more succinct code and avoids code duplication, one might expect that it incurs a runtime performance penalty since the pattern matching according to the isomorphism $f :\simeq: g :+: \mathit{Dbl}$ means that values over f have to be first decomposed and then composed again to obtain values over $g :+: \mathit{Dbl}$. To test this hypothesis, we have performed a number of benchmarks using the *Criterion* Haskell library. We tested extended implementations of the desugaring as well the evaluation example presented above. We were not able to see any difference in the runtime between the implementations using *split* and the implementations using type classes. Surprisingly, this still holds as we increase the number of summands in the signatures. We measured the runtime for examples using signatures with up to 25 summands and did not see any difference in runtime performance.

6. Improving Performance and Error Messages

In this section we shall refine the implementation we presented in section 5 in order to produce more efficient injection and projection functions as well as more helpful error messages.

```
data Pos = Here | Left Pos | Right Pos | Sum Pos Pos
data Res = Found Pos | NotFound | Ambiguous

type family Elem (f :: ∗ → ∗) (g :: ∗ → ∗) :: Res where
  Elem f f            = Found Here
  Elem f (g1 :+: g2)  = Choose f (g1 :+: g2)
                                 (Elem f g1) (Elem f g2)
  Elem f g            = NotFound

type family Choose f g (l :: Res) (r :: Res) :: Res where
  Choose f g (Found x)  (Found y)  = Ambiguous
  Choose f g Ambiguous  y          = Ambiguous
  Choose f g x          Ambiguous  = Ambiguous
  Choose f g (Found x)  y          = Found (Left x)
  Choose f g x          (Found y)  = Found (Right y)
  Choose (f1 :+: f2) g x y = Sum′ (Elem f1 g) (Elem f2 g)
  Choose f           g x y = NotFound

type family Sum′ (l :: Res) (r :: Res) :: Res where
  Sum′ (Found x)  (Found y)  = Found (Sum x y)
  Sum′ Ambiguous  y          = Ambiguous
  Sum′ x          Ambiguous  = Ambiguous
  Sum′ x          y          = NotFound
```

Figure 4. Implementation of *Elem*.

6.1 A More Efficient Implementation

The implementation of $:\prec:$ from section 5 is a straightforward extension of the simple implementation given in section 4: it decomposes the left-hand side of a subsumption constraint into its atomic components and then uses the simple implementation on each of these atomic components. In some circumstances this approach causes the derived implementations of *inj* and *prj* to perform unnecessary decomposition and recomposition of its arguments.

For example, consider the seemingly innocuous subsumption *Arith* $:\prec:$ *Arith* $:+:$ *Mult*. Since *Arith* is defined as the sum *Val*$:+:$*Add*, the function *inj* is effectively implemented as follows:

```
inj :: Arith a → (Arith :+: Mult) a
inj (Inl x) = Inl (Inl x)
inj (Inr x) = Inl (Inr x)
```

It pattern matches on its argument only to reconstruct the original argument again. Instead, *inj* could be implemented simply as *Inl*.

In order to achieve this behaviour, we shall refine the implementation of the type family *Subsume* such that it interleaves the deconstruction of the left-hand side signature with the search for an embedding into the right-hand side. The resulting implementation of the *Elem* type family is shown in Figure 4.

The kind *Res* is defined as previously, but we have changed the kind *Pos* to include a type constructor *Sum*. This additional type constructor corresponds to the type constructor of the same name for the kind *Struc* (cf. section 5.1). It indicates that the left-hand side signature is a sum, and that we need to decompose it into its two summands in order to find the desired embedding.

The definition of the type family *Elem* is similar to the original definition of *Elem* in section 4. The only difference is that it passes the two original signatures to the *Choose* type family. These two additional arguments to *Choose* are needed for the additional equation that was added compared to the original definition from section 4, namely the equation

```
Choose (f1 :+: f2) g x y = Sum′ (Elem f1 g) (Elem f2 g)
```

```
class Subsume (e :: Emb) (f :: ∗ → ∗) (g :: ∗ → ∗) where
  inj′ :: Proxy e → f a → g a
  prj′ :: Proxy e → g a → Maybe (f a)

instance Subsume (Found Here) f f where
  inj′ _ = id
  prj′ _ = Just

instance Subsume (Found p) f g
  ⇒ Subsume (Found (Left p)) f (g :+: g′) where
  inj′ _ = Inl ∘ inj′ (P :: Proxy (Found p))
  prj′ _  (Inl x) = prj′ (P :: Proxy (Found p)) x
  prj′ _  _       = Nothing

instance Subsume (Found p) f g
  ⇒ Subsume (Found (Right p)) f (g′ :+: g) where
  inj′ _ = Inr ∘ inj′ (P :: Proxy (Found p))
  prj′ _  (Inr x) = prj′ (P :: Proxy (Found p)) x
  prj′ _  _       = Nothing

instance (Subsume (Found p1) f1 g,
          Subsume (Found p2) f2 g)
  ⇒ Subsume (Found (Sum p1 p2)) (f1 :+: f2) g where
  inj′ _ (Inl x)  = inj′ (P :: Proxy (Found p1)) x
  inj′ _ (Inr x)  = inj′ (P :: Proxy (Found p2)) x
  prj′ _ x = case prj′ (P :: Proxy (Found p1)) x of
    Just y → Just (Inl y)
    _      → case prj′ (P :: Proxy (Found p2)) x of
               Just y → Just (Inr y)
               _      → Nothing
```

Figure 5. Implementation of *Subsume*.

Here we try to decompose the left-hand signature in case we were not able to find an embedding for the whole signature. *Elem* is used recursively on the two summands. If both yield a position, these positions are combined by *Sum*, otherwise *Ambiguous* and *NotFound* are propagated.

For instance we have the following type equalities

```
Elem Arith (Arith :+: Mult) ∼ Found (Left Here)
Elem (Val :+: Mult) (Arith :+: Mult)
      ∼ Found (Sum (Left (Left Here)) (Right Here))
```

Finally, we need to adjust the type class *Subsume* to this reimplementation of *Elem*. The implementation of *Subsume* is shown in Figure 5. The instance declarations follow the structure of *Pos*: *Here* produces a reflexive subsumption; *Left* and *Right* expect a sum on the right-hand side and recurse on the left resp. the right summand; and *Sum* expects a sum on the left-hand side of the subsumption and recurses on both summands.

The definition of the constraint $:\prec:$ itself remains the same. In particular, we can reuse the type family *Dupl* for checking for duplicates on the left-hand side:

```
type f :≺: g = (Subsume (Elem f g) f g,
                Dupl f ′[] ∼ False)
```

One can check that the derived implementations for *inj* and *prj* indeed satisfy the INVERSE law.

6.2 Error Messages

Our implementation of $:\prec:$ already produces quite helpful error messages. For instance, consider the following function definition:

$$injVal :: Val\ a \to (Arith :+: Val)\ a$$
$$injVal = inj$$

The use of inj requires the subsumption $Val :\prec: Arith :+: Val$, which should be rejected since Val occurs twice in the right-hand side. GHC produces the following error message, which informs the programmer that Val is not subsumed by $Arith :+: Val$ and that ambiguity is the culprit:

```
No instance for
   (Subsume Ambiguous Val (Arith :+: Val))
      arising from a use of 'inj'
```

In the following example we try to use an injection that requires $Dbl :\prec: Mult :+: Arith$:

$$injDbl :: Dbl\ a \to (Mult :+: Arith)\ a$$
$$injDbl = inj$$

As this is not the case, GHC produces the following error message, informing the programmer that Dbl cannot be found in $Mult :+: Arith$, and thus there is no such subsumption:

```
No instance for
   (Subsume 'NotFound Dbl (Mult :+: Arith))
      arising from a use of 'inj'
```

Compare this to Swierstra's original type class-based implementation, which would produce the following error message:

```
No instance for (Dbl :<: Add)
   arising from a use of 'inj'
```

This error message is not quite as helpful, since it does not indicate the original subsumption relation that should be satisfied, namely $Dbl :\prec: Mult :+: Arith$. Giving this information can be quite valuable. For example, maybe the error ways caused by accidently using $Mult$ instead of Dbl in the sum on the right-hand side.

While the $Subsume$ type class produces reasonably helpful error messages, the second part of the $:\prec:$ constraint, namely $Dupl\ f\ '[] \sim False$, does certainly not. If we try to derive a subsumption relation with duplicates on the left-hand side, e.g. $Val :+: Arith :\prec: Arith$, then GHC provides the error message:

```
Couldn't match type 'True' with 'False'
  In the expression: inj
```

To circumvent this problem, we replace the equality check by a type class that has only one instance, namely for $False$. In addition, we also give it the signature that is checked for duplicates as an argument, so it will show up in error messages:

$$\textbf{type}\ f :\prec: g = (Subsume\ (Elem\ f\ g)\ f\ g,$$
$$NoDupl\ f\ (Dupl\ f\ '[]))$$

$$\textbf{class}\ NoDupl\ f\ s$$

$$\textbf{instance}\ NoDupl\ f\ False$$

With this definition we get the following more helpful error message:

```
No instance for (NoDupl (Val :+: Arith) True)
  In the expression: inj
```

Finally, we should note that the refined subsumption constraint $:\prec:$ defined in this section is more liberal with ambiguous embeddings compared its previous version presented in section 5. We redefined $Elem$ such that it tries to find embeddings as early as possible in order to avoid unnecessary decomposition of signatures. As a consequence, we can derive the following subsumption:

$$Add :+: Val :\prec: (Add :+: Val) :+: Val$$

$Elem$ immediately returns $Found\ (Left\ Here)$ without further decomposing the left-hand side signature. However, there are obviously two ways of embedding Val from the left-hand side into the right-hand side signature.

This issue can be avoided by also requiring that right-hand sides do not contain duplicates. Thus we redefine $:\prec:$ one last time:

$$\textbf{type}\ f :\prec: g = (Subsume\ (Elem\ f\ g)\ f\ g,$$
$$NoDupl\ f\ (Dupl\ f\ '[]),$$
$$NoDupl\ g\ (Dupl\ g\ '[]))$$

This definition is more restrictive than before as it also disallows duplication on the right-hand side even though it is not in the image of the embedding. For instance, we can no longer derive

$$Val :\prec: Add :+: Add :+: Val$$

which was possible with the definition of subsumption from section 5. As duplication of signatures on either sides of the subsumption relation is almost certainly unintentional, this more restrictive behaviour is to be preferred.

7. Discussion

7.1 Limitations

The new implementation of the signature subsumption constraint $:\prec:$ improves the original implementation in many respects as we have shown throughout the paper. But, unfortunately, replacing type classes by type families has some drawbacks.

Ground Signatures The most important limitation is that $:\prec:$ only works for ground types, i.e. neither side may contain variables. This is to be excepted since we cannot rule out both ambiguity and duplication if the signatures on either side of $:\prec:$ are not fully instantiated. For example, we may not derive that $Val :\prec: f :+: Val$, since if f were instantiated by Val, then the subsumption would be ambiguous.

Concretely, this restriction manifests itself in the implicit requirement for *apartness* in the semantics of closed type families (Eisenberg et al. 2014). Specifically, an equation of a closed type family is applied only if it matches and is apart from any other equation occurring above it (unless it would yield the same result). Intuitively, the apartness requirement means that there is no possible instantiation of type variables that would make a previous equation applicable. (More correctly, it is a conservative approximation of this intuition.)

For example, if we were to write the function

$$valInj :: Val\ a \to (f :+: Val)\ a$$
$$valInj = inj$$

which requires the constraint $Val :\prec: f :+: Val$ to be derivable, the simplification of the type $Elem\ Val\ (f :+: Val)$ gets stuck at

$$Choose\ Val\ (f :+: Val)\ (Elem\ Val\ f)\ (Found\ Here)$$

The fifth equation for the type family $Choose$ (cf. Figure 4) matches. However, if f was instantiated to Val, then the first equation would match; and if f was instantiated to $Val :+: Val$ the second equation would match. Therefore, we cannot (and should not) apply the fifth equation.

This restriction to ground signatures becomes even more apparent for the $Dupl$ type family (cf. Figure 1). Intuitively, it is clear that we cannot rule out that a signature functor contains duplicates if it contains a variable summand, as the variable may be instantiated by $Val :+: Val$, say. Concretely, this can be seen in the definition of $Dupl$. The type $Dupl\ f\ l$ cannot be simplified if f is a variable: the first equation of $Dupl$ does no match, but it may match if f is instantiated to a sum.

Error Messages Due to the apartness restriction of closed type families, simplification of types may fail as we have described above. This may lead to overly verbose error messages. For example, if we ask GHC to type check the function definition for $valInj$ given above we receive the following error message:

```
No instance for
  (Subsume (Choose Val (f :+: Val)
                   (Elem Val f) (Found Here))
         Val (f :+: Val))
      arising from a use of 'inj'
```

Here the error message is polluted with the type that could not be simplified further due to lack of apartness as described above. Nonetheless, the error message still contains the relevant information: there is no instance for $Subsume\ (\dots)\ Val\ (f :+: Val)$, i.e. Val is not subsumed by $f :+: Val$.

Apart from the unnecessary verbosity, error messages like the one above also expose the user of the library to implementation details that are not part of the API. In particular, the above error mentions the type class $Subsume$ and the type families $Choose$ and $Elem$, with which a user of the library should not be concerned.

As a result, comprehending the error messages for our library requires some practice. Ideally, as library authors we would like to adjust the error messages that our library produces such that they adhere to the abstractions of the API and explain errors in terms of the domain of the library. Alas, GHC does not provide any interface that would allow such customisation of error messages.

Recently, Christiansen (2014) presented a simple, reflection-based mechanism to customise error messages in the dependently typed functional programming language *Idris* (Brady 2013). With an customisation interface for error messages similar to Christiansen's, we would be able to drastically simplify error messages, which would make our library much easier to use.

Compile Time Performance Using the implementation from section 5 we can easily deal with large signatures comprising 25 summands without a noticeable delay in type checking. Unfortunately, we did notice a significant impact on type checking performance with the implementation from section 6: for a larger program using signatures consisting of more than 10 summands, type checking becomes impractically slow (in the order of minutes!).

We found that this performance bottleneck was caused by the following equation for the $Choose$ type family (cf. Figure 4):

$$Choose\ (f_1 :+: f_2)\ g\ x\ y = Sum'\ (Elem\ f_1\ g)\ (Elem\ f_2\ g)$$

To avoid this problem, we remove this equation and instead add the following as the second equation for $Elem$:

$$Elem\ (f_1 :+: f_2)\ g = Sum'\ (Elem\ f_1\ g)\ (Elem\ f_2\ g)$$

This change also makes it possible to remove the first two arguments from $Choose$, since they become unnecessary.

The resulting implementation would produce the same (suboptimal) injection and projection functions as the implementation from section 5. We can, however, restore the semantics of the original implementation by post-processing the result of $Elem$ appropriately. This approach also allows us to remove the explicit check for duplicates of the right-hand side signatures of subsumption constraints. Moreover, checking for duplicates on the left-hand side can be done by inspecting the result obtained from $Elem$, which yields an additional speedup. As a result we get even better compile time performance than the implementation from section 5, allowing us to work with large signatures without problems.

7.2 Related Work

The limitation of the original implementation of data types à la carte is rooted in the fact that Haskell's search for suitable instances does not backtrack. Morris and Jones (2010) proposed an alternative to Haskell's overlapping type class instances, called *instance chains*, that does perform backtracking. As demonstrated by Morris and Jones (2010), instance chains can be used to give a backtracking implementation of $:\prec:$. In particular, they also give an implementation that avoids ambiguity, i.e. subsumptions with multiple possible injections. We expect that their backtracking implementation $:\prec:$ can be extended to also allow compound left-hand sides and to express the isomorphism constraint $:\simeq:$. Unfortunately, however, instance chains have not been implemented in Haskell.

The theorem proving assistants Isabelle (Nipkow et al. 2002) and Coq (Bertot and Castéran 2004) both implement a type class system similar to Haskell's. Both systems, however, resolve type class instances by backtracking (Nipkow and Snelting 1991; Sozeau and Oury 2008). Thus the natural type class-based definition of $:\prec:$ can be given directly in these systems.

7.3 Promoting Functions

Our implementation uses data type promotion (Yorgey et al. 2012), to promote data types such as Pos and Emb to the kind level such that we can define closed type families on the resulting kinds. Recently, Eisenberg and Stolarek (2014) introduced a library that promotes function definitions to closed type family definitions. This function promotion mechanism allows the programmer to use the familiar syntax of Haskell function definitions to define closed type families. In particular, the programmer may then use constructs like **case** and **let**, which are not supported in closed type family definitions.

For example, we may define the type family Sum' from Figure 4 in the following way:

$$
\begin{array}{llll}
\$\,(promote\ [d\ | & & & \\
\quad sum' :: Emb \to Emb \to Emb & & & \\
\quad sum'\ (Found\ x) & (Found\ y) & = Found\ (Sum\ x\ y) & \\
\quad sum'\ Ambiguous\ y & & = Ambiguous & \\
\quad sum'\ x & Ambiguous & = Ambiguous & \\
\quad sum'\ x & y & = NotFound & |])
\end{array}
$$

The above code defines a function sum' with the specified type. This definition is then passed to the $promote$ function, which generates a corresponding definition of a type family Sum'. The resulting definition of Sum' is equivalent to the one given in Figure 4.

Since Sum' is quite simple, we do not gain any advantage over the original definition. It would be more helpful if we were able to write $Elem$ in this style (cf. Figure 4). A more natural definition of $Elem$ would replace the use of the helper type family $Choose$ with a **case** expression. Alas, we cannot use function promotion to define $Elem$, since $Elem$ is defined on kinds containing the kind $*$, which has no counterpart at the type level. Similarly, also the type family $Dupl$ in Figure 1 works on kinds containing $*$ and is thus out of reach for a definition via promotion.

7.4 Other Applications

The implementation presented in this paper can be transferred easily to applications of similar structure. For instance, we can implement a variant of $:\prec:$ that works on types of kind $*$ instead of $* \to *$.

Note that while Haskell provides support for *kind polymorphism* (Yorgey et al. 2012), we do need to re-implement $:\prec:$ and the underlying machinery essentially for each kind we want to use it on. This lack of polymorphism is due to the type constructor $:+:$. According to the definition of $:+:$, the kind of signatures can be at most generalised to the polymorphic kind $k \to *$.

More interestingly, we can also transfer $:\prec:$ from binary sums to binary products, with the intended semantics that $e :\prec: p$ indicates that every component of e is also a component of p. For instance,

$(Int, Bool) \mathrel{:\prec:} (Bool, (Char, Int))$. Using the technique described in this paper, we can implement put and get functions as follows:

$$put :: (e \mathrel{:\prec:} p) \Rightarrow p \to e \to p$$
$$get :: (e \mathrel{:\prec:} p) \Rightarrow p \to e$$

These functions satisfy the expected equations:

$$put\ p\ (get\ p) = p$$
$$get\ (put\ p\ e) = e$$
$$put\ (put\ p\ e)\ e' = put\ p\ e'$$

This setup is especially useful for implementing automata in a modular fashion (Bahr 2012) as it allows us to easily combine state spaces of different automata using binary products.

More generally, binary products with automatically derived put and get functions as described above can be used as a lightweight alternative to the implementation of extensible records of Kiselyov et al. (2004). It is lightweight, as it does not require to give type-level identifiers to the components of the extensible record/product type. Instead, our implementation uses the type information in order to select the right component.

Implementing extensible product types by dispatching on the type information alone is typically not a good choice as it is error-prone. For example, consider the following selector function:

$$getInt :: (Age, Int) \to Int$$
$$getInt = get$$

It may seem obvious what the semantics of $getInt$ is. But what happens if Age happens to be defined by

$$\mathbf{type}\ Age = Int$$

There is no obvious choice whether $getInt$ should return the first or the second component. Luckily, with our implementation this situation cannot occur. The detection of ambiguities that we implemented for the subsumption constraint on signatures carries over to this implementation as well. In the above situation, the programmer would receive an error message. She would then have to resolve the problem by defining Age as a $\mathbf{newtype}$ instead.

Kiselyov et al. (2004) implement a similar idea in the form type-indexed products. They use type classes to implement a constraint that checks for duplication. However, their products are always list-like and have no additional structure. Our implementation retains the nested structure of the binary products. As mentioned above, we are able to derive the subtyping $(Int, Bool) \mathrel{:\prec:} (Bool, (Char, Int))$, which thus yields a get function of type $(Bool, (Char, Int)) \to (Int, Bool)$. Using the subtyping constraint we can also implement an isomorphism constraint $:\simeq:$ such that we have for example

$$(Int, (Char, Bool)) :\simeq: (Bool, (Char, Int))$$

together with automatically derived functions that witness the isomorphism.

We have used an implementation of extensible product types as described above in an embedding of attribute grammars in Haskell (Bahr and Axelsson 2014). The fact that components are selected according to the type information makes it easier to combine attribute grammar fragments in a modular fashion compared to an implementation that uses extensible records à la Kiselyov et al. (2004) such as the embedding by Viera et al. (2009).

Acknowledgements This work was funded by the Danish Council for Independent Research, Grant 12-132365.

References

P. Bahr. Modular tree automata. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, volume 7342 of *LNCS*, pages 263–299. Springer, 2012.

P. Bahr and E. Axelsson. Generalising tree traversals to DAGs: Exploiting sharing without the pain. Unpublished manuscript, available from the author's web site, 2014.

P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94. ACM, 2011.

P. Bahr and T. Hvitved. `compdata` Haskell library. Available on `http://hackage.haskell.org/package/compdata`, 2014.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253. ACM, 2005.

D. R. Christiansen. Reflect on your mistakes! Lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.

B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218. ACM, 2013.

R. A. Eisenberg and J. Stolarek. Promoting functions to type families in haskell. Unpublished manuscript, 2014.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 671–683. ACM, 2014.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004.

J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 375–386. ACM, 2010.

T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 1–14. Springer, 1991.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, 2004.

M. Sozeau and N. Oury. First-class type classes. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.

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

M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 245–246. ACM Press, 2009.

P. Wadler. The expression problem. Available on `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.