Trees That Grow

Shayan Najd
(Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, U.K.
sh.najd@gmail.com)

Simon Peyton Jones
(Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com)

Abstract: We study the notion of extensibility in functional data types, as a new approach to the problem of decorating abstract syntax trees with additional information. We observed the need for such extensibility while redesigning the data types representing Haskell abstract syntax inside Glasgow Haskell Compiler (GHC).

Specifically, we describe a programming idiom that exploits type-level functions to allow a particular form of extensibility. The approach scales to support existentials and generalised algebraic data types, and we can use pattern synonyms to make it convenient in practice.

Key Words: functional programming, Haskell, algebraic data types, pattern matching, open data types, extensible data types, expression problem, tree decoration tree annotation

Category: D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.13 [Reusable Software]: Reusable libraries; D.3.2 [Language Classifications]: Extensible languages; D.3.3 [Programming Languages]: Language Constructs and Features-Data Types and Structures;

1 Introduction

Back in the late 1970's, David Turner's inspirational work on SK-combinators (Turner, 1979b,a), and his languages Sasl (Turner, 1976), KRC (Turner, 1982), and Miranda (Turner, 1985), were hugely influential in the early development of functional programming. They introduced a generation of young computer scientists to the joy and beauty of functional programming, in a very direct and concrete way: elegant ideas; simple, perspicuous writing; and compelling interactive implementations. David's work has had sustained impact; for example, Miranda had a major influence on the design of Haskell (Hudak *et al.*, 2007).

Algebraic Data Types (ADTs) and pattern matching are now firmly established as a core feature of any modern functional language. They first appeared as a usable feature in Hope (Burstall *et al.*, 1980), and were rapidly adopted in ML (Milner, 1984), and in David Turner's Miranda. ADTs make functional languages a fertile ground in which to define and process tree-like structures. However, *trees often cannot grow*; once a data type is defined and compiled, its definition cannot be extended by adding new data constructors, and/or by adding new fields to its existing data constructors.

This lack of extensibility can be very painful. For example, at the centre of all compilers stand tall trees representing the abstract syntax of terms. Compiler programs processing these trees often do so by decorating the trees with additional information. For instance, a name resolution phase adds information about names, and a type inference phase stores the inferred types in the relevant nodes. We refer to such extra information as *decorations*. The additional information may appear as new fields to the existing data constructors, and/or new data constructors in data types representing the trees.

The compiler writer is then faced with two unpalatable choices. She can define a new data type representing the output decorated tree, at the cost of much duplication. Or she can write a single data type with all the necessary fields and constructors, at the cost of having many unused fields and constructors at different stages of compilation.

This dilemma is very real. The Glasgow Haskell Compiler (GHC) has a single data type *HsSyn* that crosses several compiler phases; and a second entire data type *TH.Syntax* for Template Haskell. Moreover, some Haskell libraries, notably `haskell-src-exts` define yet another data type for Haskell source code. These data types are large (dozens of types, hundreds of constructors) and are very difficult to keep in sync.

In this paper we offer a systematic programming idiom that resolves the dilemma, by providing a way to extend data types within Haskell. We leverage type-level openness to allow extensibility of term-level data constructors.

Specifically, we make the following contributions

- We describe a simple but powerful programming idiom that allows a data type to be extended both with extra constructor-specific fields and with extra constructors (Section 3).
- We show that the idea can be extended to work for existentials and GADTs (Section 3.10).

We discuss related work in Section 5 and conclude in Section 6.

On a personal note, David's papers and language implementations played a major role in drawing one of us (Simon) into the world of functional programming. My very first paper, Yacc in Sasl (Peyton Jones, 1985), was a parser generator for Sasl, and David acted as a mentor for me, at a time when I had no idea what programming language research was, or how to do it. Thank you David: I will be forever grateful for your encouragement and guidance in the launch phase of my professional life.

2 The challenge

In this section, we demonstrate the problem of decorating trees, and sketch some conventional ways to address it.

2.1 Tree-Decoration Problem

A compiler might need several variants of data types representing terms. For example:

- We might want to label every node with its source location.
- After name resolution we might want to decorate names in the tree with additional information, such as their namespace.
- After type inference we might want to decorate some (but not all) constructors of the tree with inferred types.
- The type checker might record type abstractions and applications that are not present in the source code. For this it would need to add new data constructors to the type — and for these constructors a source location might not make sense.

One approach is to declare a completely new data type for each variant, but this is obviously unattractive because of the duplication it involves. In a realistic setting, the abstract syntax for a source language might have tens of data types (expressions, patterns, guards, comprehensions, declarations, sequences, bindings, matches, etc etc), and hundreds of data constructors altogether.

The Glasgow Haskell Compiler (GHC) makes an excellent (if incestuous) case study for the challenge of extensibility. In GHC, the syntax of Haskell, *HsSyn*, defines no fewer than 97 distinct data types with a total of 321 data constructors. It would be completely infeasible to define multiple variants of such a huge collection of types. Not only would it be terrible to duplicate the data structures, but we would also have to duplicate general functions like the pretty printer.

2.2 So what does GHC do?

Faced with this dilemma, what does GHC do in practice? It adopts a variety of strategies:

- *Straightforward parameterisation.* The entire syntax is parameterised over the type of variables, so that we have[1]

[1] These types are much simplified, but they convey the right idea for present purposes.

```
parse     :: String → HsExpr RdrName
rename    :: HsExpr RdrName → HsExpr Name
typecheck :: HsExpr Name → HsExpr Id
```

For example, the type checker replaces each *Name* with an *Id*; the latter is a *Name* decorated with a *Type*.

– *Extra data constructors.* The data types include parts like

```
data HsPat id = ...
   | ConPat    id [Located (HsPat id)]
   | ConPatOut id ... other fields ...
```

where the type checker is expected to replace all uses of *ConPat* with *ConPatOut*. This is clearly unsatisfactory because the passes before the type checker will never meet a *ConPatOut* but there is no static guarantee of that fact.

– *Alternating data types.* GHC needs to pin a source location on every source-syntax node (e.g., for reporting errors). It does so by alternating between two types. In the *ConPat* constructor above, the *Located* type is defined thus:

```
data Located x = L SrcLoc x
```

So there is a type-enforced alternation between *HsPat* and *Located* nodes. This idiom works quite well, but is often tiresome when traversing a tree because there are so many *L* nodes to skip over.

– *Phase-indexed fields.* GHC uses the power of type families (Chakravarty *et al.* (2005)) to describe fields that are present only before or after a specific phase. For example, we see

```
data HsExpr id = ...
   | ExplicitPArr (PostTc id Type) [LHsExpr id]
```

where the *PostTc* type family is defined thus:

```
type family   PostTc id      a
type instance PostTc RdrName a = ()
type instance PostTc Name    a = ()
type instance PostTc Id      a = a
```

This idiom makes use of the fact that *HsSyn* is parameterised on the type of identifiers, and that type makes a good proxy for the compiler phase. So the first field of an *ExplicitPArr* is () after parsing and after renaming, but is *Type* after type checking.

All this works well enough for GHC, but it is very GHC-specific. Other tools want to parse and analyse Haskell source code define their own data types; the widely-used library `haskell-src-exts` is a good example. Even GHC defines a completely separate data type for Template Haskell, in *Language.Haskell.TH.Syntax*.

These data types rapidly get out of sync, and involve a great deal of duplicated effort.

3 An idiom that supports data type extension

We now introduce a programming idiom that allows data types to be extended in more systematic way than the ad-hoc tricks described above.

We explain our solution with a running example. This paper is typeset from a literate Haskell source file using lhs2TeX (Hinze and Löh, 2015), and the code runs on GHC 8.0 using a set of well-established language extensions.

```
{-# LANGUAGE TypeFamilies, GADTs, DataKinds, ConstraintKinds #-}
{-# LANGUAGE EmptyCase, StandaloneDeriving #-}
{-# LANGUAGE TypeOperators, PatternSynonyms #-}
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
import GHC.Types (Constraint)
```

3.1 Extensible ADT Declarations

As a running example, consider the following language of simply-typed lambda terms with integer literals, and explicit type annotations:

$$
\begin{array}{lll}
i & \in & \text{integers} \\
x, y & \in & \text{variables} \\
A, B, C & \in & \text{TYP} ::= \textbf{Int} \mid A \to B \\
L, M, N & \in & \text{EXP} ::= i \mid x \mid M :: A \mid \boldsymbol{\lambda} x.\, N \mid L\ M
\end{array}
$$

In Haskell, the language above can be declared as the following data types:

```
data Exp = Lit  Integer          type Var = String
         | Var  Var            : data Typ = Int
         | Ann  Exp Typ        :          | Fun Typ Typ
         | Abs  Var Exp        :
         | App  Exp Exp
```

The data type *Exp* is not extensible. Our idea is to make it extensible like this:

```
data ExpX ξ = LitX  (XLit ξ) Integer
            | VarX  (XVar ξ) Var
            | AnnX  (XAnn ξ) (ExpX ξ) Typ
            | AbsX  (XAbs ξ) Var      (ExpX ξ)
            | AppX  (XApp ξ) (ExpX ξ) (ExpX ξ)
            | ExpX  (XExp ξ)
```

```
type family XLit ξ
type family XVar ξ
type family XAnn ξ
type family XAbs ξ
type family XApp ξ
type family XExp ξ
```

In this new data type declaration:

– ξ is a type index to Exp_X. We call ξ the *extension descriptor*, because it describes which extension is in use. For example Exp_X *TC* might be a variant of Exp_X for the type checker for a language; we will see many examples shortly.

– Each data constructor C has an extra field of type $X_C\ \xi$, where X_C is a type family, or type-level function (Chakravarty *et al.*, 2005). We can use this field to extend a data constructor with extra fields (Section 3.3). For example, if we define X_{App} *TC* to be *Typ*, the *App* constructor of a tree of type Exp_X *TC* will have a *Typ* field.

– The data type has one extra data constructor Exp_X, which has one field of type $X_{Exp}\ \xi$. We can use this field to extend the data type with new constructors (Section 3.4).

Now, we can use the above extensible data type to define a completely undecorated (UD) variant of Exp_X as follows.

```
type ExpUD = ExpX UD
data UD
type instance XLit UD = Void
type instance XVar UD = Void
```

```
type instance XAnn UD = Void
type instance XAbs UD = Void
type instance XApp UD = Void
type instance XExp UD = Void
```

Since the non-decorated variant does not introduce any forms of extensions, all mappings are set to *Void* which is declared (in *Data.Void*) like this:

```
data Void
void :: Void
void = error "Attempt to evaluate void"
```

```
absurd :: Void → a
absurd m = case m of { }
```

That is, *Void* is a data type with no constructors, so it is inhabited only by bottom.

With this instantiation, $\mathit{Exp}_X\ \mathit{UD}$ is almost exactly[2] isomorphic to the original data type Exp; that is, there is a 1-1 correspondence between values of $\mathit{Exp}_X\ \mathit{UD}$ and values of Exp.

The alert reader may realise that the **type instance** declarations can all be omitted because, in the absence of such instances, $X_{\mathit{Ann}}\ \mathit{UD}$ is irreducible and hence is an empty type just like Void. But then there is no way to prevent clients of $\mathit{Exp}_X\ \mathit{UD}$ from accidentally *adding* an instance for $X_{\mathit{Ann}}\ \mathit{UD}$, so we generally prefer to prevent that by giving an explicit instance.

3.2 Pattern Synonyms for Convenience

One can program directly over the new Exp_X type, but it is a bit less convenient than it was with the original Exp data type:

- When pattern matching, we must ignore the extra field in each constructor.
- When constructing, we must supply *void* in the extra field.

For example:

$$\begin{array}{lll}
\mathit{incLit} & :: \mathit{Exp} \to \mathit{Exp} & \\
\mathit{incLit} & (\mathit{Lit}\ i) & = \mathit{Lit}\ (i+1) \\
\mathit{incLit} & e & = e \\
\mathit{incLit}_X & :: \mathit{Exp}^{\mathit{UD}} \to \mathit{Exp}^{\mathit{UD}} & \\
\mathit{incLit}_X & (\mathit{Lit}_X\ \_\ i) & = \mathit{Lit}_X\ \mathit{void}\ (i+1) \quad \text{-- Tiresome clutter} \\
\mathit{incLit}_X & e & = e
\end{array}$$

Solving this kind of inconvenience is exactly what pattern synonyms were invented for (Pickering *et al.*, 2016). We may define a pattern synonym thus

$$\begin{array}{l}
\textbf{pattern}\ \mathit{Lit}^{\mathit{UD}} :: \mathit{Integer} \to \mathit{Exp}^{\mathit{UD}} \\
\textbf{pattern}\ \mathit{Lit}^{\mathit{UD}}\ i \leftarrow \mathit{Lit}_X\ \_\ \ \ i \\
\quad \textbf{where}\ \mathit{Lit}^{\mathit{UD}}\ i = \mathit{Lit}_X\ \mathit{void}\ i
\end{array}$$

and similarly for all the other data constructors. This is a so-called *bidirectional* pattern synonym. In a pattern $\mathit{Lit}^{\mathit{UD}}\ i$ expands to $\mathit{Lit}_X\ \_\ i$, while in a term $\mathit{Lit}^{\mathit{UD}}\ i$ expands to $\mathit{Lit}_X\ \mathit{void}\ i$. So now we can write

$$\begin{array}{lll}
\mathit{incLit}_X & :: \mathit{Exp}^{\mathit{UD}} \to \mathit{Exp}^{\mathit{UD}} & \\
\mathit{incLit}_X & (\mathit{Lit}^{\mathit{UD}}\ i) & = \mathit{Lit}^{\mathit{UD}}\ (i+1) \quad \text{-- No tiresome clutter} \\
\mathit{incLit}_X & e & = e
\end{array}$$

[2] We say "almost exactly" because the term value $\mathit{Exp}_X\ \mathit{void}$ has no counterpart in Exp; alas Haskell lacks an entirely uninhabited type. We can simply hide the constructor Exp_X from the client users to ameliorate this problem.

3.3 New Field Extensions

Now, consider the following simple type system for our example language.

$$\boxed{\Gamma \vdash M : A} \quad \frac{}{\Gamma \vdash i : \mathbf{Int}} \quad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma \vdash M : A}{\Gamma \vdash M \mathbin{\boldsymbol{::}} A : A}$$

$$\frac{x : A, \Gamma \vdash N : B}{\Gamma \vdash \boldsymbol{\lambda} x\boldsymbol{.}\, N : A \to B} \quad \frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash L\; M : B}$$

Before type checking, often abstract syntax trees (ASTs) are processed by a type inference engine. The output of the type inference engine is the same input tree decorated with additional type information. Type inference helps users to leave certain bits of their programs without explicit type annotations. Type inference also helps in simplifying the type checker: after type inference, and decorating the trees with the additional type information, type checking becomes a straightforward syntax-directed recursive definition. To accommodate for the additional information in the output, we need larger trees, and hence we need to *extend* the original declarations. For instance, the following highlights the required changes to the non-extensible *Exp* data type:

```
data Exp = ... | App Typ Exp Exp
```

The definition is just like that of *Exp*, save for extending constructor App_X with a new field of type *Typ*, as highlighted above. The duplication is unpleasant (particularly when the data type is much larger).

In the extensible setting both non-decorated and decorated variants of *Exp* can be defined as extensions to the same base extensible data type Exp_X. Following the same approach as before, we can also define a decorated variant of Exp_X suitable for type checking (TC) based on Exp_X as follows.

```
type Exp^TC = Exp_X TC                 type instance X_Ann TC = Void
data TC                                type instance X_Abs TC = Void
type instance X_Lit TC = Void          type instance X_App TC = Typ
type instance X_Var TC = Void          type instance X_Exp TC = Void
```

The difference (highlighted) is just that the *App* constructor gets an extra field of type *Typ*, just as required.

The pattern synonyms for Exp^{TC} can be defined as before, save for constructor App_X that takes an extra argument for the new field introduced by the extension, as highlighted below:

```
pattern App^TC :: Typ → Exp^TC → Exp^TC → Exp^TC
pattern App^TC a l m = App_X a l m
```

3.4 New Constructor Extensions

We could as well consider scenarios where extended data types introduce new constructors. For instance, consider a simple partial evaluation (PE) pass over the trees, where β-redices are normalised away in an input tree. After reducing a redex to a value, the partial evaluator stores the value as a relevant node inside the tree. To be able to decorate the tree with this new information (i.e., values), often new constructors should be introduced to the declarations. For instance, the following highlights the required changes (i.e., the new constructor *Val*) to the non-extensible *Exp* data type:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Val}\ = \ldots \\
\mathbf{data}\ \mathit{Exp} = \ldots \mid \mathit{Val}\ \mathit{Val}
\end{array}
$$

We can still reuse our extensible data type Exp_X to define a variant suitable for such partial evaluation (PE) by extending it with a new constructor $\mathit{Val}^{\mathit{PE}}$ as

$$
\begin{array}{ll}
\mathbf{type}\ \mathit{Exp}^{\mathit{PE}} = \mathit{Exp}_X\ \mathit{PE} & \mathbf{type\ instance}\ X_{\mathit{Ann}}\ \mathit{PE} = \mathit{Void} \\
\mathbf{data}\ \mathit{PE} & \mathbf{type\ instance}\ X_{\mathit{Abs}}\ \ \mathit{PE} = \mathit{Void} \\
\mathbf{type\ instance}\ X_{\mathit{Lit}}\ \ \mathit{PE} = \mathit{Void} & \mathbf{type\ instance}\ X_{\mathit{App}}\ \mathit{PE} = \mathit{Void} \\
\mathbf{type\ instance}\ X_{\mathit{Var}}\ \mathit{PE} = \mathit{Void} & \mathbf{type\ instance}\ X_{\mathit{Exp}}\ \ \mathit{PE} = \mathit{Val}
\end{array}
$$

The pattern synonyms for $\mathit{Exp}_X\ \mathit{PE}$ can be defined as before, except that we introduce a new pattern synonym $\mathit{Val}^{\mathit{PE}}$ that represents the new constructor introduced by the extension, as highlighted below:

$$
\begin{array}{l}
\mathbf{pattern}\ \mathit{Val}^{\mathit{PE}} :: \mathit{Val} \rightarrow \mathit{Exp}^{\mathit{PE}} \\
\mathbf{pattern}\ \mathit{Val}^{\mathit{PE}}\ v\ = \mathit{Exp}_X\ v
\end{array}
$$

3.5 Normal Functions on Extended Data Types

Aided by the pattern synonyms, programming over the extended data type feels very much like programming over an ordinary non-extensible data type. For example, here is a type checker following the typing rules in Section 3.3:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{check} :: \mathit{Exp}^{\mathit{TC}} \rightarrow [(\mathit{Var}, \mathit{Typ})] \rightarrow \mathit{Typ} \rightarrow \mathit{Bool}} \\
\mathit{check}\ (\mathit{Lit}^{\mathit{TC}} & \_) & \_\ \mathit{Int} & = \mathit{True} \\
\mathit{check}\ (\mathit{Var}^{\mathit{TC}} & x) & \Gamma\ c & = \mathit{maybe}\ \mathit{False}\ (\equiv c)\ (\mathit{lookup}\ x\ \Gamma) \\
\mathit{check}\ (\mathit{Ann}^{\mathit{TC}} & m\ a) & \Gamma\ c & = a \equiv c \wedge \mathit{check}\ m\ \Gamma\ c \\
\mathit{check}\ (\mathit{Abs}^{\mathit{TC}} & x\ \ n) & \Gamma\ (\mathit{Fun}\ a\ b) & = \mathit{check}\ n\ ((x, a) : \Gamma)\ b \\
\mathit{check}\ (\mathit{App}^{\mathit{TC}}\ a\ l & m) & \Gamma\ c & = \mathit{check}\ l\ \Gamma\ (\mathit{Fun}\ a\ c) \wedge \mathit{check}\ m\ \Gamma\ a \\
\mathit{check}\ \_ & & \_\ \_ & = \mathit{False}
\end{array}
$$

One significant annoyance is that GHC is not yet clever enough to know when pattern synonyms are exhaustive, so the pattern-match exhaustiveness checker is effectively powerless.

3.6 Generic Functions on Extensible Data Types

We can sometimes exploit the common structure inherited from an extensible data type, to define generic functions acting uniformly over the extending data types. For instance, we can define a generic printer function once and reuse it for the extended data types. Let us begin with a simple printer that ignores the decorations introduced as new fields in the data type. For instance, such a printer works the same for both the undecorated data type Exp^{UD} and the decorated data type Exp^{TC}. Compilers often use such printers across multiple phases to print terms while reporting error messages.

For the new constructor extensions, we can either ignore them like we do for the new fields, or use function parameters to handle printing of these new constructors. We choose to do the latter in the following example.

```
printT :: Typ → String
printT Int              = "Int"
printT (Fun a b)        = "(" ++ printT a ++ ") → " ++ printT b

printE :: (XExp ξ → String) → ExpX ξ → String
printE _ (LitX  _ i)    = show i
printE _ (VarX  _ x)    = x
printE p (AnnX  _ m a)  = "(" ++ printE p m ++ ")::(" ++ printT a ++ ")"
printE p (AbsX  _ x n)  = "λ" ++ x ++ "." ++ printE p n
printE p (AppX  _ l m)  = "(" ++ printE p l ++ ") (" ++ printE p m ++ ")"
printE p (ExpX ξ)       = p ξ
```

Above, we chose to pass explicitly the function parameters used for printing the possible new constructors. We could as well use type classes.

Having defined the above generic printer, we can reuse it to define printers for extending data types Exp^{UD}, Exp^{TC}, and Exp^{PE} as follows.

```
printEUD :: ExpUD → String
printEUD = printE absurd
printETC :: ExpTC → String
printETC = printE absurd

printEPE :: ExpPE → String
printEPE = printE p
  where p v = "{{" ++ show v ++ "}}"
deriving instance Show Val
```

Since both Exp^{UD} and Exp^{TC} introduce no new constructors, the parameters passed to the generic function does plain matching on empty types. For Exp^{PE}, however, we pass a printer function handling values of the new constructor *Val*.

3.7 Type Classes for Extensible Data Types

For the generic printer *printE*, we chose to ignore the new field extensions. We could as well make a variant that also prints the new field extensions. Such a printer is useful for debugging purposes. To implement such a printer for Exp_X, as before, we need to provide five more function parameters to handle new field extensions in each of the five constructors. The type of generic function then becomes

$$\begin{aligned}
printE :: {} & (X_{Lit}\ \xi \to String) \to (X_{Var}\ \xi \to String) \to (X_{Ann}\ \xi \to String) \to \\
& (X_{Abs}\ \xi \to String) \to (X_{App}\ \xi \to String) \to (X_{Exp}\ \xi \to String) \to \\
& Exp_X\ \xi \to String
\end{aligned}$$

Here, with this approach, genericity comes at the price of a long list of parameters that need to be passed around. But this is exactly what Haskell's type classes were designed to solve! We can instead write[3]

$$\begin{aligned}
&\textbf{instance}\ (Show\ (X_{Lit}\ \xi), Show\ (X_{Var}\ \xi), Show\ (X_{Ann}\ \xi), \\
&\qquad\qquad Show\ (X_{Abs}\ \xi), Show\ (X_{App}\ \xi), Show\ (X_{Exp}\ \xi)) \Rightarrow \\
&\qquad\qquad Show\ (Exp_X\ \xi)\ \textbf{where} \\
&\quad show = ...
\end{aligned}$$

and all the extra parameter passing becomes invisible.

Using the constraint kinds extension in GHC, we can make the process of declaring such generic instances easier by abstracting over the constraint as

$$\begin{aligned}
&\textbf{type}\ Forall_X\ (\phi :: * \to Constraint)\ \xi \\
&\quad = (\phi\ (X_{Lit}\ \xi), \phi\ (X_{Var}\ \xi), \phi\ (X_{Ann}\ \xi) \\
&\qquad , \phi\ (X_{Abs}\ \xi), \phi\ (X_{App}\ \xi), \phi\ (X_{Exp}\ \xi))
\end{aligned}$$

Hence by using above, the header of the instance declaration for Exp_X becomes as simple as

$$\begin{aligned}
&\textbf{instance}\ Forall_X\ Show\ \xi \Rightarrow Show\ (Exp_X\ \xi)\ \textbf{where} \\
&\quad show = ...
\end{aligned}$$

In this case, and many others, we can even use Haskell's automatic (standalone) instance deriving to implement the *show* method for us:

$$\textbf{deriving instance}\ Forall_X\ Show\ \xi \Rightarrow Show\ (Exp_X\ \xi)$$

[3] The type checker complains about the decidablity of type class resolution, because the constraints in the instance context are no smaller than those in the head. Therefore, we need to supply the compiler with a $-XUndecidableInstances$ flag, because we, as the programmers, know that the process is terminating for our use cases.

3.8 Replacing Constructors

In some compiler passes, changes to trees can be beyond mere extensions. For instance, a pass may require the type of a field in a constructor to change. Consider the common pass in compilers where a chain of term applications are grouped together to form a saturated application where the term at the head of the chain is directly applied to a list of terms (often with η-expansions so that the size of the arguments list matches the arity of the head term). In our running example, to store the result of the saturation pass in a variant of *Exp*, we *change* the type of the arguments in constructor *App* to a list of terms:

$$\textbf{data}\ \mathit{Exp} = \ldots \mid \mathit{App}\ \mathit{Exp}\ [\mathit{Exp}]$$

Such a change to the type of a field in a constructor, and in general changes to a constructor beyond what can be achieved by adding new fields (and smart use of pattern synonyms) can still be achieved following our idiom by replacing the constructor with a new one.

The act of replacing a constructor can be seen as two distinct extensions: (a) adding a new constructor, (b) removing the old constructor. Removing a constructor is achieved by extending it with a new field of an empty type. As mentioned earlier, Haskell does not have such an empty type, as all types are inhabited by bottom, but we can achieve a similar result by not exposing the removed constructor to the client user (as a part of the interface of the extended data type).

Assuming App_X is not exposed to the client users, the following defines a variant of Exp_X with fully saturated applications (SA), where the type of arguments in application terms is *changed* to a list of terms .

$$\begin{array}{l}
\textbf{type}\ \mathit{Exp}^{\mathit{SA}} = \mathit{Exp}_X\ \mathit{SA}\\
\textbf{data}\ \mathit{SA}\\
\textbf{type instance}\ X_{\mathit{Lit}}\ \ \mathit{SA} = \mathit{Void}\\
\textbf{type instance}\ X_{\mathit{Var}}\ \mathit{SA} = \mathit{Void}
\end{array}$$

$$\begin{array}{l}
\textbf{type instance}\ X_{\mathit{Ann}}\ \mathit{SA} = \mathit{Void}\\
\textbf{type instance}\ X_{\mathit{Abs}}\ \ \mathit{SA} = \mathit{Void}\\
\textbf{type instance}\ X_{\mathit{App}}\ \ \mathit{SA} = \mathit{Void}\\
\textbf{type instance}\ X_{\mathit{Exp}}\ \ \mathit{SA}\\
\quad = (\mathit{Exp}^{\mathit{SA}}, [\mathit{Exp}^{\mathit{SA}}])
\end{array}$$

Now the new exposed application constructor can be defined by the following pattern synonym:

$$\begin{array}{l}
\textbf{pattern}\ \mathit{App}^{\mathit{SA}} :: \mathit{Exp}^{\mathit{SA}} \to [\mathit{Exp}^{\mathit{SA}}] \to \mathit{Exp}^{\mathit{SA}}\\
\textbf{pattern}\ \mathit{App}^{\mathit{SA}}\ l\ \mathit{ms} = \mathit{Exp}_X\ (l, \mathit{ms})
\end{array}$$

3.9 Extensions Using Type Parameters

The running example we considered so far has had no type parameters, besides the extension descriptor parameter that we have introduced. Many of the data types that we want to extend do have type parameters. For instance, consider a variant of *Exp* that is parametric at the type of variables:

$$\textbf{data}\ \mathit{Exp}\ \alpha = \ldots \mid \mathit{Var}\ \alpha \mid \mathit{Abs}\ \alpha\ (\mathit{Exp}\ \alpha)$$

Also consider a variant of the above with additional **let** expressions, as often introduced by passes such as *let-insertion*:

$$\textbf{data}\ \mathit{Exp}\ \alpha = \ldots \mid \mathit{Let}\ \alpha\ (\mathit{Exp}\ \alpha)\ (\mathit{Exp}\ \alpha)$$

Our idiom can also describe such an extension even though the extension (i.e., type variables in **let** bindings) is referring to the type parameter α. The general idea is to directly pass the type parameters in a constructor to the corresponding extension type functions:

$$\begin{aligned}
&\textbf{data}\ \mathit{Exp}_X\ \xi\ \alpha\\
&\quad = \mathit{Lit}_X\ \ (X_{\mathit{Lit}}\ \xi\ \alpha)\ \mathit{Integer}\\
&\quad \mid \mathit{Var}_X\ \ (X_{\mathit{Var}}\ \xi\ \alpha)\ \mathit{Var}\\
&\quad \mid \mathit{Ann}_X\ (X_{\mathit{Ann}}\ \xi\ \alpha)\ (\mathit{Exp}_X\ \xi\ \alpha)\ \mathit{Typ}\\
&\quad \mid \mathit{Abs}_X\ \ (X_{\mathit{Abs}}\ \xi\ \alpha)\ \mathit{Var}\ (\mathit{Exp}_X\ \xi\ \alpha)\\
&\quad \mid \mathit{App}_X\ (X_{\mathit{App}}\ \xi\ \alpha)\ (\mathit{Exp}_X\ \xi\ \alpha)\ (\mathit{Exp}_X\ \xi\ \alpha)\\
&\quad \mid \mathit{Exp}_X\ (X_{\mathit{Exp}}\ \xi\ \alpha)
\end{aligned}
\qquad
\begin{aligned}
&\textbf{type family}\ X_{\mathit{Lit}}\ \xi\ \alpha\\
&\textbf{type family}\ X_{\mathit{Var}}\ \xi\ \alpha\\
&\textbf{type family}\ X_{\mathit{Ann}}\ \xi\ \alpha\\
&\textbf{type family}\ X_{\mathit{Abs}}\ \xi\ \alpha\\
&\textbf{type family}\ X_{\mathit{App}}\ \xi\ \alpha\\
&\textbf{type family}\ X_{\mathit{Exp}}\ \xi\ \alpha
\end{aligned}$$

The extension introducing **let** expressions (LE) is defined same as before, this time with access to the type parameter:

$$\begin{aligned}
&\textbf{type}\ \mathit{Exp}^{\mathit{LE}}\ \alpha = \mathit{Exp}_X\ \mathit{LE}\ \alpha\\
&\textbf{data}\ \mathit{LE}\\
&\textbf{type instance}\ X_{\mathit{Lit}}\ \mathit{LE}\ \alpha = \mathit{Void}\\
&\textbf{type instance}\ X_{\mathit{Var}}\ \mathit{LE}\ \alpha = \mathit{Void}
\end{aligned}
\qquad
\begin{aligned}
&\textbf{type instance}\ X_{\mathit{Ann}}\ \mathit{LE}\ \alpha = \mathit{Void}\\
&\textbf{type instance}\ X_{\mathit{Abs}}\ \mathit{LE}\ \alpha = \mathit{Void}\\
&\textbf{type instance}\ X_{\mathit{App}}\ \mathit{LE}\ \alpha = \mathit{Void}\\
&\textbf{type instance}\ X_{\mathit{Exp}}\ \mathit{LE}\ \alpha\\
&\quad = (\alpha, \mathit{Exp}^{\mathit{LE}}\ \alpha, \mathit{Exp}^{\mathit{LE}}\ \alpha)
\end{aligned}$$

Now, we can define a pattern synonym for the new constructor as before:

$$\begin{aligned}
&\textbf{pattern}\ \mathit{Let}^{\mathit{LE}} :: \alpha \to \mathit{Exp}^{\mathit{LE}}\ \alpha \to \mathit{Exp}^{\mathit{LE}}\ \alpha \to \mathit{Exp}^{\mathit{LE}}\ \alpha\\
&\textbf{pattern}\ \mathit{Let}^{\mathit{LE}}\ x\ m\ n = \mathit{Exp}_X\ (x, m, n)
\end{aligned}$$

Similarly, we can support extensible data types with more than one type variable by passing them all to the extension type functions.

3.10 Existentials and GADTs

So far, we have considered extensibility in normal algebraic data type declarations. However, in GHC, data may be defined by generalised algebraic data type (GADT) declarations. For instance, consider the following GADT declaration of a simple embedded domain-specific language of constants, application, addition, and Boolean conjunction, with one existential variable a in *App*:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ a\ \textbf{where} \\
\quad \mathit{Con} :: c \to \mathit{Exp}\ c \\
\quad \mathit{App} :: \mathit{Exp}\ (a \to b) \to \mathit{Exp}\ a \to \mathit{Exp}\ b \\
\quad \mathit{Add} :: \mathit{Exp}\ (\mathit{Int} \quad \to \mathit{Int} \to \quad \mathit{Int}) \\
\quad \mathit{And} :: \mathit{Exp}\ (\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool})
\end{array}
$$

We cannot print terms of this type: due to the polymorphic type of the field in *Con*, we need a printer for values of type α when printing $\mathit{Exp}\ \alpha$, and since a is locally quantified in *App* and unavailable outside, we cannot supply such a printer from outside. We need to extend constructor *App* to store a printer for a right inside the constructor:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ a\ \textbf{where} \\
\quad \mathit{App} :: (a \to \mathit{String}) \to \mathit{Exp}\ (a \to b) \to \mathit{Exp}\ a \to \mathit{Exp}\ b \\
\quad \ldots
\end{array}
$$

Our idiom scales to generalised algebraic data types and supports extensibility as above. To do so, we need to be able to access existential variables when defining extensions. As in the previous section, we can do so by simply passing the existential types to the extension type functions as well. For the above example, we have the following extensible declaration:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}_X\ \xi\ a\ \textbf{where} \\
\quad \mathit{Con}_X :: X_{\mathit{Con}}\ \xi\ c \quad\ \to c \to \mathit{Exp}_X\ \xi\ c \\
\quad \mathit{App}_X :: X_{\mathit{App}}\ \xi\ a\ b \to \mathit{Exp}_X\ \xi\ (a \to b) \to \mathit{Exp}_X\ \xi\ a \to \mathit{Exp}_X\ \xi\ b \\
\quad \mathit{Add}_X :: X_{\mathit{Add}}\ \xi \qquad \to \mathit{Exp}_X\ \xi\ (\mathit{Int} \quad \to \mathit{Int} \to \quad \mathit{Int}) \\
\quad \mathit{And}_X :: X_{\mathit{And}}\ \xi \qquad \to \mathit{Exp}_X\ \xi\ (\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}) \\
\quad \mathit{Exp}_X :: X_{\mathit{Exp}}\ \xi\ a \quad\ \to \mathit{Exp}_X\ \xi\ a
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{type family}\ X_{\mathit{Con}}\ \xi\ c & & \textbf{type family}\ X_{\mathit{And}}\ \xi \\
\textbf{type family}\ X_{\mathit{App}}\ \xi\ a\ b & \vdots & \textbf{type family}\ X_{\mathit{Exp}}\ \xi\ a \\
\textbf{type family}\ X_{\mathit{Add}}\ \xi & \vdots &
\end{array}
$$

We can now define a variant of Exp_X, where App_X is extended with a new field to store a printer (Pr) for the existential type a:

```
type Exp^Pr a = Exp_X Pr a
data Pr
type instance X_Con Pr c   = Void
type instance X_App Pr a b = (a → String)
  ⋮
type instance X_Add Pr   = Void
type instance X_And Pr   = Void
type instance X_Exp Pr a = Void
```

As before, we can define pattern synonyms such as the following:

```
pattern App^Pr :: (a → String) → Exp^Pr (a → b) → Exp^Pr a → Exp^Pr b
pattern App^Pr p l m = App_X p l m
```

One other solution for writing such a printer is to constrain the indices of *Exp* with *Show* type class. This involves adding a local type constraint for the existential type *a* in *AppE*. Our idiom is also capable of expressing such extensions to the set of local type constraints. For this purpose, we need to introduce a *proof* data type *Proof* ϕ *a* that matching on its constructor convinces GHC that the constraint ϕ *a* is satisfied. So to define a variant of Exp_X where the existential type is constrained with *Show* type class (Sh), we have the following.

```
data Proof φ a where
  Proof :: φ a ⇒ Proof φ a
type Exp^Sh a = Exp_X Sh a
data Sh
type instance X_Con Sh c   = Void
type instance X_App Sh a b = Proof Show a
  ⋮
type instance X_Add Sh   = Void
type instance X_And Sh   = Void
type instance X_Exp Sh a = Void
```

Now, we can define pattern synonyms such as following:

```
pattern App^Sh :: () ⇒ Show a ⇒ Exp^Sh (a → b) → Exp^Sh a → Exp^Sh b
pattern App^Sh l m = App_X Proof l m
```

Similarly, we can add new locally quantified variables using a data type like

```
data Exists f where
  Exists :: f a → Exists f
```

3.11 Variations on Theme

Our idiom is just that: a programming idiom. For field extensions (Section 3.3), nothing requires us to add an extra field to *every* constructor, or to use a different type function for every constructor. Similarly if we do not want to extend the data type with new constructors we do not need to provide the extra data constructor that supports such extension (Section 3.4). For example, here is a more specialised variant of our running example

$$\begin{array}{llllll}
\textbf{data}\ \mathit{Exp}_X\ \xi = & \mathit{Lit}_X & (X_{Leaf}\ \xi) & \mathit{Integer} & & \textbf{type family}\ X_{Leaf}\ \xi \\
\quad\mid & \mathit{Var}_X & (X_{Leaf}\ \xi) & \mathit{Var} & & \vdots\ \textbf{type family}\ X_{AA}\ \ \xi \\
\quad\mid & \mathit{Ann}_X & & (\mathit{Exp}_X\ \xi) & \mathit{Typ} & \vdots \\
\quad\mid & \mathit{Abs}_X & (X_{AA}\ \ \xi) & \mathit{Var} & (\mathit{Exp}_X\ \xi) & \vdots \\
\quad\mid & \mathit{App}_X & (X_{AA}\ \ \xi) & (\mathit{Exp}_X\ \xi) & (\mathit{Exp}_X\ \xi) &
\end{array}$$

Here constructors Lit_X and Var_X share a single extension-field type, $X_{Leaf}\ \xi$; and similarly Abs_X and App_X; the constructor Ann_X does not have an extension field; and we cannot add new data constructors.

3.12 Shortcomings and Scope

Our approach comes with a number of shortcomings.

- *Efficiency.* Every constructor carries an extra extension field, whether or not it is used.
- *Exhaustiveness checks.* Our use of pattern synonyms (which is optional, of course) defeats GHC's current pattern-match exhaustiveness checker. And even if we did not use a pattern synonym, the extra constructor (Exp_X in our running example) will be flagged as unmatched even when we are not using it. Both are problems of engineering, rather than fundamental. [4]
- *Boilerplate.* When adding a new phase descriptor, there is a slightly uneasy choice between (a) adding lots of tiresome declarations

  type instance $X_C\ \xi = \mathit{Void}$

  one for each constructor C whose extension field is not used, and (b) omitting the instance, and hoping that no one adds it later.

  Similarly, writing lots of pattern-synonym declarations can be painful.

  One alternative we have considered is to generate the boilerplate using Template Haskell, or even to define a new language extension. But it seems better first to gain more experience of using the idiom.

Our idiom can naturally scale to support mutually recursive declarations by passing the *same* extension descriptor to all of the declarations.
We have seen that our idiom is capable of expressing extensions to a generalised algebraic data type declaration such as adding new fields, adding new

[4] In fact, there are already partially implemented general features in GHC regarding both completeness of a set of pattern synonyms, and improving the totality checker to recognise absurdity.

constructors, adding new local constraints, and adding new existential variables. We have also seen that, we can replace constructors, and access global and local type variables in our extensions.

In addition to these changes, we can combine our idiom with pattern synonyms and module system features to express other changes like

- *change to the order of fields*, such as
  pattern $(\odot) :: \mathit{Exp} \to \mathit{Exp} \to \mathit{Exp}$
  pattern $m \odot l = \mathit{App}\ l\ m$
- *removing fields*, such as
  pattern $K :: \mathit{Exp} \to \mathit{Exp}$
  pattern $K\ n \leftarrow \mathit{Abs}\ \_\ \ n$
    where $K\ n = \mathit{Abs}$ "_" n
- *fixing values of fields*, such as
  pattern $\mathit{One} :: \mathit{Exp}$
  pattern $\mathit{One} = \mathit{Lit}\ 1$

Yet, there are other possible forms of changes to a data type declaration, like adding new type variables. In the next section, we take a few steps further.

4 Extension Descriptors

So far in our examples, the extension description parameters have been empty types used as indices to define extensions. However, extension descriptors are themselves ordinary algebraic data types, and in this section we study extensibility using more complex extension descriptors.

4.1 New Type Parameter Extensions

Suppose we wanted to add a new field of type α to some or all of the data constructors in the type. Then we would need to add α as a parameter of the data type itself. Can we do that?

In our example, suppose we wanted to add a source location to every node in the tree. Source location decorations associated with a node may appear as new fields, or as new constructors wrapping nodes with a source location. Strictly speaking, the latter approach is less precise compared to the former: such wrapper constructors can be applied to a node more than once, or not applied at all. With the former, the programmer is in control: using the optional type (e.g., *Maybe*) of source locations in decorations models the optional application of wrapper constructors, and using the list of source locations models the multiple applications of wrapper constructors to a node. Regardless of the decoration approach, the type of source locations (annotations in general) is often kept polymorphic, allowing programmers to define generic functions like *fmap*, *fold*, and *traverse*. A good example is the AST in `Haskell-Src-Exts`, where the polymorphic annotations are used for different purposes, including the source locations used in an exact printer. In our extensible setting, support for polymorphic source locations amounts to (1) extending the AST declarations with a new type parameter α (the type of source locations) and (2) extending all the constructors with a new field of type α. To do so, we need the ability to extend an ADT data type declaration with a set of type variables, and to access these variables to define extensions, such as new fields. Our encoding is capable of expressing such new type parameter extensions: the idea is to carry the extra type parameters in the extension descriptors. For instance, the following defines an extension to Exp_X with a new type variable α, and uses it to define polymorphic annotations An as new field extensions.

```
type Exp^An α = Exp_X (An α)
data An α
type instance X_Lit (An α) = α
type instance X_Var (An α) = α
type instance X_Ann (An α) = α
type instance X_Abs (An α) = α
type instance X_App (An α) = α
type instance X_Exp (An α) = Void
```

Notice that we made the definition of the extension descriptor parametric, and then we could access the parameter when defining extensions.

4.2 Hierarchy of Extension Descriptors

In practice, compilers may have multiple variants of an AST, many of which are closely related to each other. For instance in GHC, the AST in the front-end of the compiler, named *HsSyn*, has three major variations used in the parsing, renaming, and type-checking passes. GHC also has an entirely separate variant as a part of its metaprogramming mechanism Template Haskell. The first three are closely related, while the last is quite different. We can organise such variations by putting them in hierarchies of indices and use this hierarchy when defining extensions. For instance, for GHC, we may define the extension descriptor as

```
data GHC (c :: Component)
data Component = Compiler Pass | TemplateHaskell
data Pass      = Parser | Renamer | TypeChecker
```

Having the above as a hierarchy of extension descriptors, we get the four variations of *HsSyn* AST in the extensible setting. For instance, the type checker AST would be of the type *HsSyn* (*Compiler TypeChecker*).

It also allows us to define generic extension descriptors such as

```
type family PostTC p where
  PostTC TypeChecker = Typ
  PostTC _           = Void
type instance X_App (GHC TemplateHaskell) = Void
type instance X_App (GHC (Compiler p))    = PostTC p
```

5 Discussion and Related Works

The problem of extensibility in data types is a hot topic in computer science. There are many different approach to this problem. To name a few: structural and nominal subtyping, extensible records and variants, and numerous approaches to Wadler's expression problem. There are too many solutions to mention; the reader may consult the references in Torgersen (2004); Axelsson (2012); Swierstra (2008); Lindley and Cheney (2012); Bahr and Hvitved (2011); Löh and Hinze (2006), for some examples. However, surprisingly, our problem, and hence our solution, has unique distinguishing characteristics:

Need for both directions of data extensibility: We need, and provide, extensibility on two major directions of data extensibility: adding new fields to existing constructors, and adding new constructors. The so-called "expression problem" and its solutions are often only concerned with the latter form of extensibility.

Generic programming is a plus, not a must: Our primary goal is to re-use the data type declaration itself, rather than to re-use existing functions that operate over that type. For example, in GHC, the parser produces *HsSyn RdrName*, the renamer consumes that and produces *HsSyn Name*, which the type checker consumes to produce *HsSyn Id*. All three passes are monomorhic: they consume and produce a single fixed variant of the underlying data type.

In contrast, work addressing the expression problem is often concerned with re-usability and compositionality of functions defined per cases.

As we have seen with some examples (e.g., the generic printer), one can write and reuse functions that are polymorphic in the extension descriptor, but only by (a) simply discarding or preserving the decorations, or (b) using auxiliary higher order functions to process the decorations. If one wishes to take functions written only for a specific variant of a data type and reuse them, as an after-thought, for other variants, certain forms of static guarantees (possibly, beyond what types currently provide) are required for safety. One common practice here is to focus on certain subclass of data types.

Trees are declared: In our setting, trees are often declared, rather than them being anonymous. There are well-known trade-offs between declared and anonymous data structures. The former is simpler and less error-prone, and the latter enables more opportunities for generic programming. Row polymorphism, and the similar, often infer the structure of data from their uses, leading to large types, bad performance, and complicated error messages. Our approach is based on declaring both extensible and extended data types (by describing the exact extensions). It resembles the long lasting problem of supporting anonymous records, such as Trex (Gaster, 1998), in GHC, where solutions with declared flavour often dodge the problem by leaving programmers to do some of the work by providing more information.

Similar to our idiom in spirit is McBride's Ornaments (Dagand and McBride, 2014; Williams *et al.*, 2014). The key idea of ornaments is to declare transformations of data types that preserve the recursive structures of data types, with focus on reusing functions defined on the original for the transformed data types. While our idiom can benefit from works on ornaments for such reuse, there are decorations in practice that do not preserve the recursive structures. For instance, in GHC, for better or worse, the constructor representing if-expressions (like some others) is decorated with one additional expression to store user-defined macros rebinding if-syntax, hence not preserving the recursive structure.

Works with the current technology: Existing solutions often demand changes to the compiler. Some other, come at the price of losing certain desirable properties, such as decidablity of type inference, or predictability of the performance. In contrast, our solution works in GHC right now (v8.0).

6 Conclusion

In the 1980s we were mainly concerned with functional programming over *terms*, but this paper has mainly focused on functional programming over *types*, with the interesting new twist that type functions (unlike term functions) can be open. We have explored how to leverage that type-level openness to allow extensibility of term-level data constructors. David, we hope that you approve. Happy birthday!

7 Acknowledgement

The authors thank Axelsson, Broberg, Carette, Dagand, Diatchki, Eisenberg, Löh, McBride, Rémy, Weirich, and Zimmerman for the fruitful and encouraging discussions about this work. This work grew out of the Summer of Haskell 2016 project "Native metaprogramming in Haskell", done by Najd under the mentorship of Peyton Jones and Carette; we are grateful to the Haskell.Org community and the organisers. Najd was funded by a Google Fellowship in Programming Technology.

References

E. Axelsson. A generic abstract syntax model for embedded languages. In *International Conference on Functional Programming (ICFP)*, 2012.

P. Bahr and T. Hvitved. Compositional data types. In *Workshop on Generic Programming*, 2011.

R. Burstall, D. MacQueen, and D. Sannella. HOPE - an experimental applicative language. In *ACM Lisp Conference*, pages 1236–143, 1980.

M. M. T. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In *Principles of Programming Languages (POPL)*, 2005.

P.-É. Dagand and C. McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 005 2014.

B. R. Gaster. *Records, variants and qualified types.* PhD thesis, University of Nottingham, 1998.

R. Hinze and A. Löh. lhs2TeX, 2015.

P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages (HOPL-III)*, June 2007.

S. Lindley and J. Cheney. Row-based effect types for database integration. In *Types in Language Design and Implementation (TLDI)*, 2012.

A. Löh and R. Hinze. Open data types and open functions. In *Principles and Practice of Declarative Programming (PPDP)*, 2006.

R. Milner. A proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

S. L. Peyton Jones. Yacc in Sasl — an exercise in functional programming. *Software Practice and Experience*, 15(8):807–820, August 1985.

M. Pickering, G. Érdi, S. Peyton Jones, and R. A. Eisenberg. Pattern synonyms. In *Haskell Symposium*, 2016.

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

M. Torgersen. The expression problem revisited. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

D. A. Turner. The SASL language manual. Technical report, University of St Andrews, 1976.

D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, June 1979.

D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

D. A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional Programming and its Applications*. CUP, 1982.

D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 1–16. Springer, 1985.

T. Williams, P. Dagand, and D. Rémy. Ornaments in practice. In *Workshop on Generic Programming (WGP)*, 2014.