




# Deep Embedding with Class

Mart Lubbers<sup>(✉)</sup> 

Institute for Computing and Information Sciences, Radboud University Nijmegen,  
Nijmegen, The Netherlands  
`mart@cs.ru.nl`

**Abstract.** The two flavours of DSL embedding are shallow and deep embedding. In functional languages, shallow embedding models the language constructs as functions in which the semantics are embedded. Adding semantics is therefore cumbersome while adding constructs is a breeze. Upgrading the functions to type classes lifts this limitation to a certain extent.

Deeply embedded languages represent their language constructs as data and the semantics are functions on it. As a result, the language constructs are embedded in the semantics, hence adding new language constructs is laborious where adding semantics is trouble free.

This paper shows that by abstracting the semantics functions in deep embedding to type classes, it is possible to easily add language constructs as well. So-called classy deep embedding results in DSLs that are extensible both in language constructs and in semantics while maintaining a concrete abstract syntax tree. Additionally, little type-level trickery or complicated boilerplate code is required to achieve this.

**Keywords:** Functional programming · Haskell · Embedded domain-specific languages

## 1 Introduction

The two flavours of DSL embedding are deep and shallow embedding [4]. In functional programming languages, shallow embedding models language constructs as functions in the host language. As a result, adding new language constructs—extra functions—is easy. However, the semantics of the language is embedded in these functions, making it troublesome to add semantics since it requires updating all existing language constructs.

On the other hand, deep embedding models language constructs as data in the host language. The semantics of the language are represented by functions over the data. Consequently, adding new semantics, i.e. novel functions, is straightforward. It can be stated that the language constructs are embedded in the functions that form a semantics. If one wants to add a language construct, all semantics functions must be revisited and revised to avoid ending up with partial functions.

This juxtaposition has been known for many years [19] and discussed by many others [11] but most famously dubbed the *expression problem* by Wadler [24]:

The *expression problem* is a new name for an old problem. The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety (e.g., no casts).

In shallow embedding, abstracting the functions to type classes disentangles the language constructs from the semantics, allowing extension both ways. This technique is dubbed tagless-final embedding [5], nonetheless it is no silver bullet. Some semantics that require an intensional analysis of the syntax tree, such as transformation and optimisations, are difficult to implement in shallow embedding due to the lack of an explicit data structure representing the abstract syntax tree. The semantics of the DSL have to be combined and must hold some kind of state or context, so that structural information is not lost [10].

## 1.1 Research Contribution

This paper shows how to apply the technique observed in tagless-final embedding to deep embedding. The presented basic technique, christened *classy deep embedding*, does not require advanced type system extensions to be used. However, it is suitable for type system extensions such as generalised algebraic data types. While this paper is written as a literate Haskell [18] program using some minor extensions provided by GHC [23], the idea is applicable to other languages as well<sup>1</sup>.

## 2 Deep Embedding

Consider the simple language of integer literals and addition. In deep embedding, terms in the language are represented by data in the host language. Hence, defining the constructs is as simple as creating the following algebraic data type<sup>2</sup>.

$$\begin{aligned} \mathbf{data} \text{ Expr}_0 &= \text{Lit}_0 \text{ Int} \\ &| \text{Add}_0 \text{ Expr}_0 \text{ Expr}_0 \end{aligned}$$

Semantics are defined as functions on the  $\text{Expr}_0$  data type. For example, a function transforming the term to an integer—an evaluator—is implemented as follows.

$$\begin{aligned} \text{eval}_0 &:: \text{Expr}_0 \rightarrow \text{Int} \\ \text{eval}_0 (\text{Lit}_0 e) &= e \\ \text{eval}_0 (\text{Add}_0 e_1 e_2) &= \text{eval}_0 e_1 + \text{eval}_0 e_2 \end{aligned}$$

Adding semantics—e.g. a printer—just means adding another function while the existing functions remain untouched. I.e. the key property of deep embedding.

<sup>1</sup> Lubbers, M. (2022): Literate Haskell/lhs2TeX source code of the paper “Deep Embedding with Class”: TFP 2022. Zenodo. <https://doi.org/10.5281/zenodo.6650880>.

<sup>2</sup> All data types and functions are subscripted to indicate the evolution.

The following function, transforming the  $Expr_0$  data type to a string, defines a simple printer for our language.

```

print0 :: Expr0 → String
print0 (Lit0 v)    = show v
print0 (Add0 e1 e2) = "(" ++ print0 e1 ++ "-" ++ print0 e2 ++ ")"

```

While the language is concise and elegant, it is not very expressive. Traditionally, extending the language is achieved by adding a case to the  $Expr_0$  data type. So, adding subtraction to the language results in the following revised data type.

```

data Expr0 = Lit0 Int
           | Add0 Expr0 Expr0
           | Sub0 Expr0 Expr0

```

Extending the DSL with language constructs exposes the Achilles' heel of deep embedding. Adding a case to the data type means that all semantics functions have become partial and need to be updated to be able to handle this new case. This does not seem like an insurmountable problem, but it does pose a problem if either the functions or the data type itself are written by others or are contained in a closed library.

### 3 Shallow Embedding

Conversely, let us see how this would be done in shallow embedding. First, the data type is represented by functions in the host language with embedded semantics. Therefore, the evaluators for literals and addition both become a function in the host language as follows.

```

type Sems = Int
lits :: Int → Sems
lits i = i
adds :: Sems → Sems → Sems
adds e1 e2 = e1 + e2

```

Adding constructions to the language is done by adding functions. Hence, the following function adds subtraction to our language.

```

subs :: Sems → Sems → Sems
subs e1 e2 = e1 - e2

```

Adding semantics on the other hand—e.g. a printer—is not that simple because the semantics are part of the functions representing the language constructs. One way to add semantics is to change all functions to execute both semantics at the same time. In our case this means changing the type of  $Sem_s$  to be  $(Int, String)$  so that all functions operate on a tuple containing the result of the evaluator and the printed representation at the same time. Alternatively, a single semantics can be defined that represents a fold over the language constructs [8], delaying the selection of semantics to the moment the fold is applied.

### 3.1 Tagless-Final Embedding

Tagless-final embedding overcomes the limitations of standard shallow embedding. To upgrade to this embedding technique, the language constructs are changed from functions to type classes. For our language this results in the following type class definition.

```
class  $Expr_t$   $s$  where
   $lit_t :: Int \rightarrow s$ 
   $add_t :: s \rightarrow s \rightarrow s$ 
```

Semantics become data types<sup>3</sup> implementing these type classes, resulting in the following instance for the evaluator.

```
newtype  $Eval_t = E_t Int$ 
instance  $Expr_t Eval_t$  where
   $lit_t v = E_t v$ 
   $add_t (E_t e_1) (E_t e_2) = E_t (e_1 + e_2)$ 
```

Adding constructs—e.g. subtraction—just results in an extra type class and corresponding instances.

```
class  $Sub_t s$  where
   $sub_t :: s \rightarrow s \rightarrow s$ 
instance  $Sub_t Eval_t$  where
   $sub_t (E_t e_1) (E_t e_2) = E_t (e_1 - e_2)$ 
```

Finally, adding semantics such as a printer over the language is achieved by providing a data type representing the semantics accompanied by instances for the language constructs.

```
newtype  $Printer_t = P_t String$ 
instance  $Expr_t Printer_t$  where
   $lit_t i = P_t (show i)$ 
   $add_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "+" ++ e_2 ++ ")")$ 
instance  $Sub_t Printer_t$  where
   $sub_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "-" ++ e_2 ++ ")")$ 
```

---

<sup>3</sup> In this case **newtypes** are used instead of regular **data** declarations. A **newtype** is a special data type with a single constructor containing a single value only to which it is isomorphic. It allows the programmer to define separate class instances that the instances of the isomorphic type without any overhead. During compilation the constructor is completely removed [18, Sect. 4.2.3].

## 4 Lifting the Backends

Let us rethink the deeply embedded DSL design. Remember that in shallow embedding, the semantics are embedded in the language construct functions. Obtaining extensibility both in constructs and semantics was accomplished by abstracting the semantics functions to type classes, making the constructs overloaded in the semantics. In deep embedding, the constructs are embedded in the semantics functions instead. So, let us apply the same technique, i.e. make the semantics overloaded in the language constructs by abstracting the semantics functions to type classes. The same effect may be achieved when using similar techniques such as explicit dictionary passing or ML style modules. In our language this results in the following type class.

```
class Eval1 v where
  eval1 :: v → Int
data Expr1 = Lit1 Int
           | Add1 Expr1 Expr1
```

Implementing the semantics type class instances for the *Expr*<sub>1</sub> data type is an elementary exercise. By a copy-paste and some modifications, we come to the following implementation.

```
instance Eval1 Expr1 where
  eval1 (Lit1 v)      = v
  eval1 (Add1 e1 e2) = eval1 e1 + eval1 e2
```

Subtraction can now be defined in a separate data type, leaving the original data type intact. Instances for the additional semantics can now be implemented separately as instances of the type classes.

```
data Sub1 = Sub1 Expr1 Expr1
instance Eval1 Sub1 where
  eval1 (Sub1 e1 e2) = eval1 e1 - eval1 e2
```

## 5 Existential Data Types

The astute reader might have noticed that we have dissociated ourselves from the original data type. It is only possible to create an expression with a subtraction on the top level. The recursive knot is left untied and as a result, *Sub*<sub>1</sub> can never be reached from an *Expr*<sub>1</sub>.

Luckily, we can reconnect them by adding a special constructor to the *Expr*<sub>1</sub> data type for housing extensions. It contains an existentially quantified [15] type with type class constraints [13, 14] for all semantics type classes [23, Sect. 6.4.6] to allow it to house not just subtraction but any future extension.

```

data Expr2 =
  | Lit2 Int
  | Add2 Expr2 Expr2
  | forall x.Eval2 x ⇒ Ext2 x

```

The implementation of the extension case in the semantics type classes is in most cases just a matter of calling the function for the argument as can be seen in the semantics instances shown below.

```

instance Eval2 Expr2 where
  eval2 (Lit2 v) = v
  eval2 (Add2 e1 e2) = eval2 e1 + eval2 e2
  eval2 (Ext2 x) = eval2 x

```

Adding language construct extensions in different data types does mean that an extra *Ext<sub>2</sub>* tag is introduced when using the extension. This burden can be relieved by creating a smart constructor for it that automatically wraps the extension with the *Ext<sub>2</sub>* constructor so that it is of the type of the main data type.

```

sub2 :: Expr2 → Expr2 → Expr2
sub2 e1 e2 = Ext2 (Sub2 e1 e2)

```

In our example this means that the programmer can write<sup>4</sup>:

```

e2 :: Expr2
e2 = Lit2 42 'sub2' Lit2 1

```

instead of having to write

```

e'2 :: Expr2
e'2 = Ext2 (Lit2 42 'Sub2' Lit2 1)

```

## 5.1 Unbraiding the Semantics from the Data

This approach does reveal a minor problem. Namely, that all semantics type classes are braided into our datatypes via the *Ext<sub>2</sub>* constructor. Say if we add the printer again, the *Ext<sub>2</sub>* constructor has to be modified to contain the printer type class constraint as well<sup>5</sup>. Thus, if we add semantics, the main data type's type class constraints in the *Ext<sub>2</sub>* constructor need to be updated. To avoid this, the type classes can be bundled in a type class alias or type class collection as follows.

```

class (Eval2 x, Print2 x) ⇒ Semantics2 x
data Expr2 = Lit2 Int

```

<sup>4</sup> Backticks are used to use functions or constructors in an infix fashion [18, Sect. 4.3.3].

<sup>5</sup> Resulting in the following constructor: **forall** x.(Eval<sub>2</sub> x, Print<sub>2</sub> x) ⇒ Ext<sub>2</sub> x.

```

|                               Add2 Expr2 Expr2
| forall x.Semantics2 x ⇒ Ext2 x

```

The class alias removes the need for the programmer to visit the main data type when adding additional semantics. Unfortunately, the compiler does need to visit the main data type again. Some may argue that adding semantics happens less frequently than adding language constructs but in reality it means that we have to concede that the language is not as easily extensible in semantics as in language constructs. More exotic type system extensions such as constraint kinds [3, 25] can untangle the semantics from the data types by making the data types parametrised by the particular semantics. However, by adding some boilerplate, even without this extension, the language constructs can be parametrised by the semantics by putting the semantics functions in a data type. First the data types for the language constructs are parametrised by the type variable  $d$  as follows.

```

data Expr3 d =           Lit3 Int
|                       Add3 (Expr3 d) (Expr3 d)
| forall x.Ext3 (d x) x

```

```

data Sub3 d = Sub3 (Expr3 d) (Expr3 d)

```

The  $d$  type variable is inhabited by an explicit dictionary for the semantics, i.e. a witness to the class instance. Therefore, for all semantics type classes, a data type is made that contains the semantics function for the given semantics. This means that for  $Eval_3$ , a dictionary with the function  $EvalDict_3$  is defined, a type class  $HasEval_3$  for retrieving the function from the dictionary and an instance for  $HasEval_3$  for  $EvalDict_3$ .

```

newtype EvalDict3 v = EvalDict3 (v → Int)
class HasEval3 d where
  getEval3 :: d v → v → Int
instance HasEval3 EvalDict3 where
  getEval3 (EvalDict3 e) = e

```

The instances for the type classes change as well according to the change in the datatype. Given that there is a  $HasEval_3$  instance for the witness type  $d$ , we can provide an implementation of  $Eval_3$  for  $Expr_3 d$ .

```

instance HasEval3 d ⇒ Eval3 (Expr3 d) where
  eval3 (Lit3 v) = v
  eval3 (Add3 e1 e2) = eval3 e1 + eval3 e2
  eval3 (Ext3 d x) = getEval3 d x

```

```

instance HasEval3 d ⇒ Eval3 (Sub3 d) where
  eval3 (Sub3 e1 e2) = eval3 e1 - eval3 e2

```

Because the  $Ext_3$  constructor from  $Expr_3$  now contains a value of type  $d$ , the smart constructor for  $Sub_3$  must somehow come up with this value. To achieve this, a type class is introduced that allows the generation of such a dictionary.

```
class  $GDict$   $a$  where
   $gdict :: a$ 
```

This type class has individual instances for all semantics dictionaries, linking the class instance to the witness value. I.e. if there is a type class instance known, a witness value can be conjured using the  $gdict$  function.

```
instance  $Eval_3$   $v \Rightarrow GDict$  ( $EvalDict_3$   $v$ ) where
   $gdict = EvalDict_3$   $eval_3$ 
```

With these instances, the semantics function can be retrieved from the  $Ext_3$  constructor and in the smart constructors they can be generated as follows:

```
 $sub_3 :: GDict$  ( $d$  ( $Sub_3$   $d$ ))  $\Rightarrow Expr_3$   $d \rightarrow Expr_3$   $d \rightarrow Expr_3$   $d$ 
 $sub_3$   $e_1$   $e_2 = Ext_3$   $gdict$  ( $Sub_3$   $e_1$   $e_2$ )
```

Finally, we reached the end goal, orthogonal extension of both language constructs as shown by adding subtraction to the language and in language semantics. Adding the printer can now be done without touching the original code as follows. First the printer type class, dictionaries and instances for  $GDict$  are defined.

```
class  $Print_3$   $v$  where
   $print_3 :: v \rightarrow String$ 
newtype  $PrintDict_3$   $v = PrintDict_3$  ( $v \rightarrow String$ )
class  $HasPrint_3$   $d$  where
   $getPrint_3 :: d \rightarrow v \rightarrow String$ 
instance  $HasPrint_3$   $PrintDict_3$  where
   $getPrint_3$  ( $PrintDict_3$   $e$ ) =  $e$ 
instance  $Print_3$   $v \Rightarrow GDict$  ( $PrintDict_3$   $v$ ) where
   $gdict = PrintDict_3$   $print_3$ 
```

Then the instances for  $Print_3$  of all the language constructs can be defined.

```
instance  $HasPrint_3$   $d \Rightarrow Print_3$  ( $Expr_3$   $d$ ) where
   $print_3$  ( $Lit_3$   $v$ ) =  $show$   $v$ 
   $print_3$  ( $Add_3$   $e_1$   $e_2$ ) = "(" ++  $print_3$   $e_1$  ++ "+" ++  $print_3$   $e_2$  ++ ")"
   $print_3$  ( $Ext_3$   $d$   $x$ ) =  $getPrint_3$   $d$   $x$ 
```

```
instance  $HasPrint_3$   $d \Rightarrow Print_3$  ( $Sub_3$   $d$ ) where
   $print_3$  ( $Sub_3$   $e_1$   $e_2$ ) = "(" ++  $print_3$   $e_1$  ++ "-" ++  $print_3$   $e_2$  ++ ")"
```



## 6 Transformation Semantics

Most semantics convert a term to some final representation and can be expressed just by functions on the cases. However, the implementation of semantics such as transformation or optimisation may benefit from a so-called intentional analysis of the abstract syntax tree. In shallow embedding, the implementation for these types of semantics is difficult because there is no tangible abstract syntax tree. In off-the-shelf deep embedding this is effortless since the function can pattern match on the constructor or structures of constructors.

To demonstrate intensional analyses in classy deep embedding we write an optimizer that removes addition and subtraction by zero. In classy deep embedding, adding new semantics means first adding a new type class housing the function including the machinery for the extension constructor.

```

class Opt3 v where
  opt3 :: v → v
newtype OptDict3 v = OptDict3 (v → v)
class HasOpt3 d where
  getOpt3 :: d v → v → v
instance HasOpt3 OptDict3 where
  getOpt3 (OptDict3 e) = e
instance Opt3 v ⇒ GDict (OptDict3 v) where
  gdict = OptDict3 opt3

```

The implementation of the optimizer for the *Expr*<sub>3</sub> data type is no complicated task. The only interesting bit occurs in the *Add*<sub>3</sub> constructor, where we pattern match on the optimised children to determine whether an addition with zero is performed. If this is the case, the addition is removed.

```

instance HasOpt3 d ⇒ Opt3 (Expr3 d) where
  opt3 (Lit3 v)      = Lit3 v
  opt3 (Add3 e1 e2) = case (opt3 e1, opt3 e2) of
    (Lit3 0, e2      ) → e2
    (e1,      Lit3 0) → e1
    (e1,      e2      ) → Add3 e1 e2
  opt3 (Ext3 d x) = Ext3 d (getOpt3 d x)

```

Replicating this for the *Opt*<sub>3</sub> instance of *Sub*<sub>3</sub> seems a clear-cut task at first glance.

```

instance HasOpt3 d ⇒ Opt3 (Sub3 d) where
  opt3 (Sub3 e1 e2) = case (opt3 e1, opt3 e2) of
    (e1, Lit3 0) → e1
    (e1, e2      ) → Sub3 e1 e2

```

Unsurprisingly, this code is rejected by the compiler. When a literal zero is matched as the right-hand side of a subtraction, the left-hand side of type *Expr*<sub>3</sub>

is returned. However, the type signature of the function dictates that it should be of type  $Sub_3$ . To overcome this problem we add a convolution constructor.

## 6.1 Convolution

Adding a loopback case or convolution constructor to  $Sub_3$  allows the removal of the  $Sub_3$  constructor while remaining the  $Sub_3$  type. It should be noted that a loopback case is *only* required if the transformation actually removes tags. This changes the  $Sub_3$  data type as follows.

```
data  $Sub_4$   $d = Sub_4$       ( $Expr_4$   $d$ ) ( $Expr_4$   $d$ )
      |  $SubLoop_4$  ( $Expr_4$   $d$ )
instance  $HasEval_4$   $d \Rightarrow Eval_4$  ( $Sub_4$   $d$ ) where
   $eval_4$  ( $Sub_4$        $e_1$   $e_2$ ) =  $eval_4$   $e_1 - eval_4$   $e_2$ 
   $eval_4$  ( $SubLoop_4$   $e_1$ )    =  $eval_4$   $e_1$ 
```

With this loopback case in the toolbox, the following  $Sub$  instance optimises away subtraction with zero literals.

```
instance  $HasOpt_4$   $d \Rightarrow Opt_4$  ( $Sub_4$   $d$ ) where
   $opt_4$  ( $Sub_4$   $e_1$   $e_2$ ) = case ( $opt_4$   $e_1$ ,  $opt_4$   $e_2$ ) of
    ( $e'_1$ ,  $Lit_4$  0)  $\rightarrow SubLoop_4$   $e'_1$ 
    ( $e'_1$ ,  $e'_2$     )  $\rightarrow Sub_4$   $e'_1$   $e'_2$ 
   $opt_4$  ( $SubLoop_4$   $e$ ) =  $SubLoop_4$  ( $opt_4$   $e$ )
```

## 6.2 Pattern Matching

Pattern matching within datatypes and from an extension to the main data type works out of the box. Cross-extensional pattern matching on the other hand—matching on a particular extension—is something that requires a bit of extra care. Take for example negation propagation and double negation elimination. Pattern matching on values with an existential type is not possible without leveraging dynamic typing [1, 2]. To enable dynamic typing support, the *Typeable* type class as provided by *Data.Dynamic* [22] is added to the list of constraints in all places where we need to pattern match across extensions. As a result, the *Typeable* type class constraints are added to the quantified type variable  $x$  of the  $Ext_4$  constructor and to  $ds$  in the smart constructors.

```
data  $Expr_4$   $d =$ 
      |  $Lit_4$   $Int$ 
      |  $Add_4$  ( $Expr_4$   $d$ ) ( $Expr_4$   $d$ )
      | forall  $x$ .  $Typeable$   $x \Rightarrow Ext_4$  ( $d$   $x$ )  $x$ 
```

First let us add negation to the language by defining a datatype representing it. Negation elimination requires the removal of negation constructors, so a convolution constructor is defined as well.

```

data  $Neg_4$   $d = Neg_4$       ( $Expr_4$   $d$ )
      |  $NegLoop_4$  ( $Expr_4$   $d$ )
 $neg_4 :: (Typeable$   $d, GDict$  ( $d$  ( $Neg_4$   $d$ )))  $\Rightarrow Expr_4$   $d \rightarrow Expr_4$   $d$ 
 $neg_4$   $e = Ext_4$   $gdict$  ( $Neg_4$   $e$ )

```

The evaluation and printer instances for the  $Neg_4$  datatype are defined as follows.

```

instance  $HasEval_4$   $d \Rightarrow Eval_4$  ( $Neg_4$   $d$ ) where
   $eval_4$  ( $Neg_4$        $e$ ) =  $negate$  ( $eval_4$   $e$ )
   $eval_4$  ( $NegLoop_4$   $e$ ) =  $eval_4$   $e$ 
instance  $HasPrint_4$   $d \Rightarrow Print_4$  ( $Neg_4$   $d$ ) where
   $print_4$  ( $Neg_4$        $e$ ) = "(~" ++  $print_4$   $e$  ++ ")"
   $print_4$  ( $NegLoop_4$   $e$ ) =  $print_4$   $e$ 

```

The  $Opt_4$  instance contains the interesting bit. If the sub expression of a negation is an addition, negation is propagated downwards. If the sub expression is again a negation, something that can only be found out by a dynamic pattern match, it is replaced by a  $NegLoop_4$  constructor.

```

instance ( $Typeable$   $d, GDict$  ( $d$  ( $Neg_4$   $d$ )),  $HasOpt_4$   $d$ )  $\Rightarrow$ 
   $Opt_4$  ( $Neg_4$   $d$ ) where
   $opt_4$  ( $Neg_4$  ( $Add_4$   $e_1$   $e_2$ ))
    =  $NegLoop_4$  ( $Add_4$  ( $opt_4$  ( $neg_4$   $e_1$ )) ( $opt_4$  ( $neg_4$   $e_2$ )))
   $opt_4$  ( $Neg_4$  ( $Ext_4$   $d$   $x$ ))
    = case  $fromDynamic$  ( $toDyn$  ( $getOpt_4$   $d$   $x$ )) of
       $Just$  ( $Neg_4$   $e$ )  $\rightarrow NegLoop_4$   $e$ 
      _  $\rightarrow Neg_4$  ( $Ext_4$   $d$  ( $getOpt_4$   $d$   $x$ ))
   $opt_4$  ( $Neg_4$        $e$ ) =  $Neg_4$  ( $opt_4$   $e$ )
   $opt_4$  ( $NegLoop_4$   $e$ ) =  $NegLoop_4$  ( $opt_4$   $e$ )

```

Loopback cases do make cross-extensional pattern matching less modular in general. For example,  $Ext_4$   $d$  ( $SubLoop_4$  ( $Lit_4$  0)) is equivalent to  $Lit_4$  0 in the optimisation semantics and would require an extra pattern match. Fortunately, this problem can be mitigated—if required—by just introducing an additional optimisation semantics that removes loopback cases. Luckily, one does not need to resort to these arguably blunt matters often. Dependent language functionality often does not need to span extensions, i.e. it is possible to group them in the same data type.

### 6.3 Chaining Semantics

Now that the data types are parametrised by the semantics a final problem needs to be overcome. The data type is parametrised by the semantics, thus, using multiple semantics, such as evaluation after optimising is not straightforwardly

possible. Luckily, a solution is readily at hand: introduce an ad-hoc combination semantics.

```

data OptPrintDict4 v = OPD4 (OptDict4 v) (PrintDict4 v)
instance HasOpt4 OptPrintDict4 where
  getOpt4 (OPD4 v _) = getOpt4 v
instance HasPrint4 OptPrintDict4 where
  getPrint4 (OPD4 _ v) = getPrint4 v
instance (Opt4 v, Print4 v) ⇒ GDict (OptPrintDict4 v) where
  gdict = OPD4 gdict gdict

```

And this allows us to write  $\text{print}_4 (\text{opt}_4 e_1)$  resulting in " $((\sim 42) + (\sim 38))$ " when  $e_1$  represents  $(\sim (42 + 38)) - 0$  and is thus defined as follows.

```

e1 :: Expr4 OptPrintDict4
e1 = neg4 (Lit4 42 ‘Add4‘ Lit4 38) ‘sub4‘ Lit4 0

```

When using classy deep embedding to the fullest, the ability of the compiler to infer very general types expires. As a consequence, defining reusable expressions that are overloaded in their semantics requires quite some type class constraints that cannot be inferred by the compiler (yet) if they use many extensions. Solving this remains future work. For example, the expression  $\sim (42 - 38) + 1$  has to be defined as:

```

e3 :: (Typeable d
      , GDict (d (Neg4 d))
      , GDict (d (Sub4 d))) ⇒ Expr4 d
e3 = neg4 (Lit4 42 ‘sub4‘ Lit4 38) ‘Add4‘ Lit4 1

```

## 7 Generalised Algebraic Data Types

Generalised algebraic data types (GADTs) are enriched data types that allow the type instantiation of the constructor to be explicitly defined [7,9]. Leveraging GADTs, deeply embedded DSLs can be made statically type safe even when different value types are supported. Even when GADTs are not supported natively in the language, they can be simulated using embedding-projection pairs or equivalence types [6, Sect. 2.2]. Where some solutions to the expression problem do not easily generalise to GADTs (see Sect. 9), classy deep embedding does. Generalising the data structure of our DSL is fairly straightforward and to spice things up a bit, we add an equality and boolean not language construct. To make the existing DSL constructs more general, we relax the types of those constructors. For example, operations on integers now work on all numerals instead. Moreover, the  $\text{Lit}_g$  constructor can be used to lift values of any type to the DSL domain as long as they have a  $\text{Show}$  instance, required for the printer. Since some optimisations on  $\text{Not}_g$  remove constructors and therefore use cross-extensional

pattern matches, *Typeable* constraints are added to  $a$ . Furthermore, because the optimisations for  $Add_g$  and  $Sub_g$  are now more general, they do not only work for *Ints* but for any type with a *Num* instance, the *Eq* constraint is added to these constructors as well. Finally, not to repeat ourselves too much, we only show the parts that substantially changed. The omitted definitions and implementation can be found in Appendix A.

```

data  $Expr_g$   $d$   $a$  where
   $Lit_g$       ::  $Show$   $a$             $\Rightarrow$   $a \rightarrow Expr_g$   $d$   $a$ 
   $Add_g$      :: ( $Eq$   $a$ ,  $Num$   $a$ )  $\Rightarrow$   $Expr_g$   $d$   $a \rightarrow Expr_g$   $d$   $a \rightarrow Expr_g$   $d$   $a$ 
   $Ext_g$      ::  $Typeable$   $x$         $\Rightarrow$   $d$   $x \rightarrow x$   $a \rightarrow Expr_g$   $d$   $a$ 
data  $Neg_g$   $d$   $a$  where
   $Neg_g$      :: ( $Typeable$   $a$ ,  $Num$   $a$ )  $\Rightarrow$   $Expr_g$   $d$   $a \rightarrow Neg_g$   $d$   $a$ 
   $NegLoop_g$  ::  $Expr_g$   $d$   $a \rightarrow Neg_g$   $d$   $a$ 
data  $Not_g$   $d$   $a$  where
   $Not_g$     ::  $Expr_g$   $d$   $Bool \rightarrow Not_g$   $d$   $Bool$ 
   $NotLoop_g$  ::  $Expr_g$   $d$   $a \rightarrow Not_g$   $d$   $a$ 

```

The smart constructors for the language extensions inherit the class constraints of their data types and include a *Typeable* constraint on the  $d$  type variable for it to be usable in the  $Ext_g$  constructor as can be seen in the smart constructor for  $Neg_g$ :

```

 $neg_g$  :: ( $Typeable$   $d$ ,  $GDict$  ( $d$  ( $Neg_g$   $d$ )),  $Typeable$   $a$ ,  $Num$   $a$ )  $\Rightarrow$ 
   $Expr_g$   $d$   $a \rightarrow Expr_g$   $d$   $a$ 
 $neg_g$   $e = Ext_g$   $gdict$  ( $Neg_g$   $e$ )
 $not_g$  :: ( $Typeable$   $d$ ,  $GDict$  ( $d$  ( $Not_g$   $d$ )))  $\Rightarrow$ 
   $Expr_g$   $d$   $Bool \rightarrow Expr_g$   $d$   $Bool$ 
 $not_g$   $e = Ext_g$   $gdict$  ( $Not_g$   $e$ )

```

Upgrading the semantics type classes to support GADTs is done by an easy textual search and replace. All occurrences of  $v$  are now parametrised by type variable  $a$ :

```

class  $Eval_g$   $v$  where
   $eval_g$  ::  $v$   $a \rightarrow a$ 
class  $Print_g$   $v$  where
   $print_g$  ::  $v$   $a \rightarrow String$ 
class  $Opt_g$   $v$  where
   $opt_g$    ::  $v$   $a \rightarrow v$   $a$ 

```

Now that the shape of the type classes has changed, the dictionary data types and the type classes need to be adapted as well. The introduced type variable  $a$  is not an argument to the type class, so it should not be an argument to the dictionary data type. To represent this type class function, a rank-2 polymorphic function is needed [23, Sect. 6.4.15] [17]. Concretely, for the evaluator this results in the following definitions:

```

newtype EvalDictg v = EvalDictg (forall a.v a → a)
class HasEvalg d where
  getEvalg :: d v → v a → a
instance HasEvalg EvalDictg where
  getEvalg (EvalDictg e) = e

```

The *GDict* type class is general enough, so the instances can remain the same. The *Eval<sub>g</sub>* instance of *GDict* looks as follows:

```

instance Evalg v ⇒ GDict (EvalDictg v) where
  gdict = EvalDictg evalg

```

Finally, the implementations for the instances can be ported without complication show using the optimisation instance of *Not<sub>g</sub>*:

```

instance (Typeable d, GDict (d (Notg d)), HasOptg d) ⇒
  Optg (Notg d) where
  optg (Notg (Extg d x))
    = case fromDynamic (toDyn (getOptg d x)) :: Maybe (Notg d Bool) of
      Just (Notg e) → NotLoopg e
      -             → Notg (Extg d (getOptg d x))
  optg (Notg e)    = Notg (optg e)
  optg (NotLoopg e) = NotLoopg (optg e)

```

## 8 Conclusion

Classy deep embedding is a novel organically grown embedding technique that alleviates deep embedding from the extensibility problem in most cases.

By abstracting the semantics functions to type classes they become overloaded in the language constructs. Thus, making it possible to add new language constructs in a separate type. These extensions are brought together in a special extension constructor residing in the main data type. This extension case is overloaded by the language construct using a data type containing the class dictionary. As a result, orthogonal extension is possible for language constructs and semantics using only little syntactic overhead or type annotations. The basic technique only requires—well established through history and relatively standard—existential data types. However, if needed, the technique generalises to GADTs as well, adding rank-2 types to the list of type system requirements as well. Finally, the abstract syntax tree remains observable which makes it suitable for intensional analyses, albeit using occasional dynamic typing for truly cross-extensional transformations.

Defining reusable expressions overloaded in semantics or using multiple semantics on a single expression requires some boilerplate still, getting around this remains future work.

## 9 Related Work

Embedded DSL techniques in functional languages have been a topic of research for many years, thus we do not claim a complete overview of related work.

Clearly, classy deep embedding bears most similarity to the *Datatypes à la Carte* [21]. In Swierstra’s approach, semantics are lifted to type classes similarly to classy deep embedding. Each language construct is their own datatype parametrised by a type parameter. This parameter contains some type level representation of language constructs that are in use. In classy deep embedding, extensions do not have to be enumerated at the type level but are captured in the extension case. Because all the constructs are expressed in the type system, nifty type system tricks need to be employed to convince the compiler that everything is type safe and the class constraints can be solved. Furthermore, it requires some boilerplate code such as functor instances for the data types. In return, pattern matching is easier and does not require dynamic typing. Classy deep embedding only strains the programmer with writing the extension case for the main data type and the occasional loopback constructor.

Löh and Hinze proposed a language extension that allows open data types and open functions, i.e. functions and data types that can be extended with more cases later on [12]. They hinted at the possibility of using type classes for open functions but had serious concerns that pattern matching would be crippled because constructors are becoming types, thus ultimately becoming impossible to type. In contrast, this paper shows that pattern matching is easily attainable—albeit using dynamic types—and that the terms can be typed without complicated type system extensions.

A technique similar to classy deep embedding was proposed by Najd and Peyton Jones to tackle a slightly different problem, namely that of reusing a data type for multiple purposes in a slightly different form [16]. For example to decorate the abstract syntax tree of a compiler differently for each phase of the compiler. They propose to add an extension descriptor as a type variable to a data type and a type family that can be used to decorate constructors with extra information and add additional constructors to the data type using an extension constructor. Classy deep embedding works similarly but uses existentially quantified type variables to describe possible extensions instead of type variables and type families. In classy deep embedding, the extensions do not need to be encoded in the type system and less boilerplate is required. Furthermore, pattern matching on extensions becomes a bit more complicated but in return it allows for multiple extensions to be added orthogonally and avoids the necessity for type system extensions.

Tagless-final embedding is the shallowly embedded counterpart of classy deep embedding and was invented for the same purpose; overcoming the issues with standard shallow embedding [5]. Classy deep embedding was organically grown from observing the evolution of tagless-final embedding. The main difference between tagless-final embedding and classy deep embedding—and in general between shallow and deep embedding—is that intensional analyses of

the abstract syntax tree is more difficult because there is no tangible abstract syntax tree data structure. In classy deep embedding, it is possible to define transformations even across extensions.

Hybrid approaches between deep and shallow embedding exist as well. For example, Svenningsson et al. show that by expressing the deeply embedded language in a shallowly embedded core language, extensions can be made orthogonally as well [20]. This paper differs from those approaches in the sense that it does not require a core language in which all extensions need to be expressible.

**Acknowledgements.** This research is partly funded by the Royal Netherlands Navy. Furthermore, I would like to thank Pieter and Rinus for the fruitful discussions, Ralf for inspiring me to write a functional pearl, and the anonymous reviewers for their valuable and honest comments.

## A Appendix

### A.1 Data Type Definitions

**data**  $Sub_g d a$  **where**

$$\begin{aligned} Sub_g &:: (Eq\ a, Num\ a) \Rightarrow Expr_g\ d\ a \rightarrow Expr_g\ d\ a \rightarrow Sub_g\ d\ a \\ SubLoop_g &:: Expr_g\ d\ a \rightarrow Sub_g\ d\ a \end{aligned}$$

**data**  $Eq_g d a$  **where**

$$\begin{aligned} Eq_g &:: (Typeable\ a, Eq\ a) \Rightarrow Expr_g\ d\ a \rightarrow Expr_g\ d\ a \rightarrow Eq_g\ d\ Bool \\ EqLoop_g &:: Expr_g\ d\ a \rightarrow Eq_g\ d\ a \end{aligned}$$

### A.2 Smart Constructors

$$\begin{aligned} sub_g &:: (Typeable\ d, GDict\ (d\ (Sub_g\ d)), Eq\ a, Num\ a) \Rightarrow \\ &Expr_g\ d\ a \rightarrow Expr_g\ d\ a \rightarrow Expr_g\ d\ a \\ sub_g\ e_1\ e_2 &= Ext_g\ gdict\ (Sub_g\ e_1\ e_2) \\ eq_g &:: (Typeable\ d, GDict\ (d\ (Eq_g\ d)), Eq\ a, Typeable\ a) \Rightarrow \\ &Expr_g\ d\ a \rightarrow Expr_g\ d\ a \rightarrow Expr_g\ d\ Bool \\ eq_g\ e_1\ e_2 &= Ext_g\ gdict\ (Eq_g\ e_1\ e_2) \end{aligned}$$

### A.3 Semantics Classes and Data Types

```
newtype  $PrintDict_g v = PrintDict_g$  (forall  $a.v\ a \rightarrow String$ )
class  $HasPrint_g d$  where
   $getPrint_g :: d\ v \rightarrow v\ a \rightarrow String$ 
instance  $HasPrint_g\ PrintDict_g$  where
   $getPrint_g\ (PrintDict_g\ e) = e$ 
```



```

newtype  $OptDict_g$   $v = OptDict_g$  (forall  $a.v$   $a \rightarrow v$   $a$ )
class  $HasOpt_g$   $d$  where
   $getOpt_g :: d$   $v \rightarrow v$   $a \rightarrow v$   $a$ 
instance  $HasOpt_g$   $OptDict_g$  where
   $getOpt_g (OptDict_g e) = e$ 

```

#### A.4 $GDict$ instances

```

instance  $Print_g$   $v \Rightarrow GDict (PrintDict_g v)$  where
   $gdic$  =  $PrintDict_g$   $print_g$ 
instance  $Opt_g$   $v \Rightarrow GDict (OptDict_g v)$  where
   $gdic$  =  $OptDict_g$   $opt_g$ 

```

#### A.5 Evaluator Instances

```

instance  $HasEval_g$   $d \Rightarrow Eval_g (Expr_g d)$  where
   $eval_g (Lit_g v) = v$ 
   $eval_g (Add_g e_1 e_2) = eval_g e_1 + eval_g e_2$ 
   $eval_g (Ext_g d x) = getEval_g d x$ 

instance  $HasEval_g$   $d \Rightarrow Eval_g (Sub_g d)$  where
   $eval_g (Sub_g e_1 e_2) = eval_g e_1 - eval_g e_2$ 
   $eval_g (SubLoop_g e) = eval_g e$ 

instance  $HasEval_g$   $d \Rightarrow Eval_g (Neg_g d)$  where
   $eval_g (Neg_g e) = negate (eval_g e)$ 
   $eval_g (NegLoop_g e) = eval_g e$ 

instance  $HasEval_g$   $d \Rightarrow Eval_g (Eq_g d)$  where
   $eval_g (Eq_g e_1 e_2) = eval_g e_1 \equiv eval_g e_2$ 
   $eval_g (EqLoop_g e) = eval_g e$ 

instance  $HasEval_g$   $d \Rightarrow Eval_g (Not_g d)$  where
   $eval_g (Not_g e) = not (eval_g e)$ 
   $eval_g (NotLoop_g e) = eval_g e$ 

```

#### A.6 Printer Instances

```

instance  $HasPrint_g$   $d \Rightarrow Print_g (Expr_g d)$  where
   $print_g (Lit_g v) = show v$ 
   $print_g (Add_g e_1 e_2) = "(" ++ print_g e_1 ++ "+" ++ print_g e_2 ++ ")"$ 
   $print_g (Ext_g d x) = getPrint_g d x$ 

```

**instance**  $HasPrint_g d \Rightarrow Print_g (Sub_g d)$  **where**  
 $print_g (Sub_g e_1 e_2) = "(" \ ++ print_g e_1 \ ++ "-" \ ++ print_g e_2 \ ++ ")"$   
 $print_g (SubLoop_g e) = print_g e$

**instance**  $HasPrint_g d \Rightarrow Print_g (Neg_g d)$  **where**  
 $print_g (Neg_g e) = "(negate " \ ++ print_g e \ ++ ")"$   
 $print_g (NegLoop_g e) = print_g e$

**instance**  $HasPrint_g d \Rightarrow Print_g (Eq_g d)$  **where**  
 $print_g (Eq_g e_1 e_2) = "(" \ ++ print_g e_1 \ ++ "==" \ ++ print_g e_2 \ ++ ")"$   
 $print_g (EqLoop_g e) = print_g e$

**instance**  $HasPrint_g d \Rightarrow Print_g (Not_g d)$  **where**  
 $print_g (Not_g e) = "(not " \ ++ print_g e \ ++ ")"$   
 $print_g (NotLoop_g e) = print_g e$

## A.7 Optimisation Instances

**instance**  $HasOpt_g d \Rightarrow Opt_g (Expr_g d)$  **where**  
 $opt_g (Lit_g v) = Lit_g v$   
 $opt_g (Add_g e_1 e_2) = \mathbf{case} (opt_g e_1, opt_g e_2)$  **of**  
 $(Lit_g 0, e'_2) \rightarrow e'_2$   
 $(e'_1, Lit_g 0) \rightarrow e'_1$   
 $(e'_1, e'_2) \rightarrow Add_g e'_1 e'_2$   
 $opt_g (Ext_g d x) = Ext_g d (getOpt_g d x)$

**instance**  $HasOpt_g d \Rightarrow Opt_g (Sub_g d)$  **where**  
 $opt_g (Sub_g e_1 e_2) = \mathbf{case} (opt_g e_1, opt_g e_2)$  **of**  
 $(e'_1, Lit_g 0) \rightarrow SubLoop_g e'_1$   
 $(e'_1, e'_2) \rightarrow Sub_g e'_1 e'_2$   
 $opt_g (SubLoop_g e) = SubLoop_g (opt_g e)$

**instance**  $(Typeable d, GDict (d (Neg_g d)), HasOpt_g d) \Rightarrow$   
 $Opt_g (Neg_g d)$  **where**  
 $opt_g (Neg_g (Add_g e_1 e_2))$   
 $= NegLoop_g (Add_g (opt_g (neg_g e_1)) (opt_g (neg_g e_2)))$   
 $opt_g (Neg_g (Ext_g d x))$   
 $= \mathbf{case} \mathit{fromDynamic} (\mathit{toDyn} (getOpt_g d x))$  **of**  
 $Just (Neg_g e) \rightarrow NegLoop_g e$   
 $- \rightarrow Neg_g (Ext_g d (getOpt_g d x))$   
 $opt_g (Neg_g e) = Neg_g (opt_g e)$   
 $opt_g (NegLoop_g e) = NegLoop_g (opt_g e)$

**instance**  $HasOpt_g d \Rightarrow Opt_g (Eq_g d)$  **where**  
 $opt_g (Eq_g e_1 e_2) = Eq_g (opt_g e_1) (opt_g e_2)$   
 $opt_g (EqLoop_g e) = EqLoop_g (opt_g e)$

## References

1. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.* **13**(2), 237–268 (1991). <https://doi.org/10.1145/103135.103138>
2. Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pp. 157–166. ICFP 2002, Association for Computing Machinery, New York (2002). <https://doi.org/10.1145/581478.581494>
3. Bolingbroke, M.: Constraint kinds for GHC (2011). <http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/>. Blog post. Accessed 09 Sep 2021
4. Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Stavridou, V., Melham, T.F., Boute, R.T. (eds.) *IFIP TC10/WG*, vol. 10, pp. 129–156. Elsevier, Amsterdam, NL (1992). Event-place: Nijmegen, NL
5. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5), 509–543 (2009). <https://doi.org/10.1017/S0956796809007205>
6. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pp. 90–104. Association for Computing Machinery, Pittsburgh, PA (2002). <https://doi.org/10.1145/581690.581698>
7. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. TR2003-1901, Cornell University (2003). <https://ecommons.cornell.edu/handle/1813/5614>
8. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pp. 339–347. ICFP 2014, Association for Computing Machinery, New York (2014). <https://doi.org/10.1145/2628136.2628138>
9. Hinze, R.: Fun with phantom types. In: Gibbons, J., de Moor, O. (eds.) *The Fun of Programming*, pp. 245–262. Bloomsbury Publishing, Palgrave, Cornerstones of Computing (2003)
10. Kiselyov, O.: Typed tagless final interpreters. In: Gibbons, J. (ed.) *Generic and Indexed Programming*. LNCS, vol. 7470, pp. 130–174. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32202-0\\_3](https://doi.org/10.1007/978-3-642-32202-0_3)
11. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. In: Jul, E. (ed.) *ECOOP 1998*. LNCS, vol. 1445, pp. 91–113. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054088>
12. Löh, A., Hinze, R.: Open data types and open functions. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 133–144. PPDP 2006, Association for Computing Machinery, New York (2006). <https://doi.org/10.1145/1140335.1140352>
13. Läufer, K.: Combining type classes and existential types. In: *Proceedings of the Latin American Informatic Conference (PANEL)*. ITESM-CEM, Monterrey, Mexico (1994)
14. Läufer, K.: Type classes with existential types. *J. Funct. Program.* **6**(3), 485–518 (1996). <https://doi.org/10.1017/S0956796800001817>
15. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* **10**(3), 470–502 (1988). <https://doi.org/10.1145/44501.45065>

16. Najd, S., Peyton Jones, S.: Trees that grow. *J. Univ. Comput. Sci.* **23**(1), 42–62 (2017)
17. Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 54–67. POPL 1996, Association for Computing Machinery, New York (1996). <https://doi.org/10.1145/237721.237729>
18. Peyton Jones, S. (ed.): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge (2003)
19. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to data abstraction. In: Gries, D. (ed.) *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, pp. 309–317. Springer, New York (1978). [https://doi.org/10.1007/978-1-4612-6315-9\\_22](https://doi.org/10.1007/978-1-4612-6315-9_22)
20. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Loidl, H.-W., Peña, R. (eds.) *TFP 2012. LNCS*, vol. 7829, pp. 21–36. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40447-4\\_2](https://doi.org/10.1007/978-3-642-40447-4_2)
21. Swierstra, W.: Data types à la carte. *J. Funct. Program.* **18**(4), 423–436 (2008). <https://doi.org/10.1017/S0956796808006758>
22. Team, G.: *Data. Dynamic* (2021). <https://hackage.haskell.org/package/base-4.14.1.0/docs/Data-Dynamic.html>. Accessed 24 Feb 2021
23. Team, G.: *GHC User’s Guide Documentation* (2021). <https://downloads.haskell.org/~ghc/latest/docs/users%20guide.pdf>. Accessed 24 Feb 2021
24. Wadler, P.: The expression problem (1998-11-12). <https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>. Accessed 24 Feb 2021
25. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, pp. 53–66. TLDI 2012, Association for Computing Machinery, New York (2012). <https://doi.org/10.1145/2103786.2103795>