First-Class Data Types in Shallow Embedded Domain-Specific Languages using Metaprogramming

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

Pieter Koopman
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

Functional programming languages are excellent for hosting embedded domain specific languages (eDSLs) because of their rich type systems, minimal syntax, and referential transparency. However, data types defined in the host language are not automatically available in the embedded language. To do so, all the operations on the data type must be ported to the eDSL resulting in a lot of boilerplate.

This paper shows that by using metaprogramming, all first-order user-defined data types can be automatically made first class in shallow embedded DSLs. We show this by providing an implementation in Template Haskell for a typical DSL with two different semantics. Furthermore, we show that by utilising quasiquotation, there is hardly any burden on the syntax. Finally, the paper also serves as a gentle introduction to Template Haskell.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Source code generation**; **Functional languages**.

KEYWORDS

functional programming, domain-specific languages, metaprogramming, Haskell, Template Haskell

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2022. First-Class Data Types in Shallow Embedded Domain-Specific Languages using Metaprogramming. In *Symposium on Implementation and Application of Functional Languages (IFL 2022), August 31–September 02, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3587216.3587219

1 INTRODUCTION

Functional programming languages are excellent candidates for hosting embedded domain specific languages (eDSLs) because of their rich type systems, minimal syntax, and referential transparency. By expressing the language constructs in the host language, the parser, the type checker, and the run time can be inherited from the host language. Unfortunately, data types defined in the host language are not automatically available in the eDSL.

The two main strategies for embedding DSLs in a functional language are deep embedding (also called initial) and shallow embedding (also called final). Deep embedding represents the constructs in the language as data types and the semantics as functions over these data types. This makes extending the language with new semantics effortless: just add another function. In contrast, adding language constructs requires changing the data type and updating all existing semantics to support this new construct. Shallow embedding on the other hand models the language constructs as functions with the semantics embedded. Consequently, adding a construct is easy, i.e. it only entails adding another function. Contrarily, adding semantics requires adapting all language constructs. Lifting the functions to type classes, i.e. parametrising the constructs over the semantics, allows extension of the language both in constructs and in semantics orthogonally. This advanced style of embedding is called tagless-final or class-based shallow embedding [Kiselyov 2012].

While it is often possible to lift values of a user-defined data type to a value in the DSL, it is not possible to interact with it using DSL constructs, since they are not first-class citizens.

Concretely, it is not possible to (1) construct values from expressions using a constructor, (2) deconstruct values into expressions using a deconstructor or pattern matching, (3) test which constructor the value holds. The functions for this are simply not available automatically in the embedded language. For some semantics—such as an interpreter—it is possible to directly lift the functions from the host language to the DSL. In other cases—e.g. *compiling* DSLs such as a compiler or a printer—this is not possible [Elliott et al. 2003]. Thus, all of the operations on the data type have to be defined by hand requiring a lot of plumbing and resulting in a lot of boilerplate code.

To relieve the burden of adding all these functions, metaprogramming—and custom quasiquoters—can be used. Metaprogramming entails that some parts of the program are generated by a program itself, i.e. the program is data. Quasiquotation is a metaprogramming mechanism that allows entering verbatim code for which a—possibly user defined—translation is used to convert the verbatim code to host language AST nodes. Metaprogramming allows functions to be added to the program at compile time based on the structure of user-defined data types.

IFL 2022, August 31–September 02, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9831-2/22/08.
https://doi.org/10.1145/3587216.3587219

1.1 Contributions of the paper

This paper shows that with the use of metaprogramming, all first-order user-defined data types can automatically be made first class for shallow embedded DSLs. It does so by providing an implementation in Template Haskell for a typical DSL with two different semantics: an interpreter and a pretty printer. Furthermore, we show that by utilising quasiquotation, there is hardly any burden on the syntax. Finally, the paper also serves as a gentle introduction to Template Haskell and reflects on the process of using Template Haskell.

2 TAGLESS-FINAL EMBEDDING

Tagless-final embedding is an upgrade to standard shallow embedding achieved by lifting all language construct functions to type classes. As a result, views on the DSL are data types implementing these classes.

To illustrate the technique, a simple DSL, a language consisting of literals and addition, is outlined. This language, implemented according to the tagless-final style [Carette et al. 2009] in Haskell [Peyton Jones 2003] consists initially only of one type class containing two functions. The `lit` function lifts values from the host language to the DSL domain. The class constraint `Show` is enforced on the type variable `a` to make sure that the value can be printed. The infix function ⊕ represents the addition of two expressions in the DSL.

```
class Expr v where
    lit :: Show a ⇒ a → v a
    (⊕) :: Num a ⇒ v a → v a → v a
infixl 6 ⊕
```

The implementation of a view on the DSL is achieved by implementing the type classes with the data type representing the view. In the case of our example DSL, an interpreter accounting for failure may be implemented as an instance for the `Maybe` type. The standard infix functor application and infix sequential application are used so that potential failure is abstracted away from.[1]

```
instance Expr Maybe where
    lit a = Just a
    (⊕) l r = (+) <$> l <*> r
```

2.1 Adding language constructs

To add an extra language construct we define a new class housing it. For example, to add division we define a new class as follows:

```
class Div v where
    (⊘) :: Integral a ⇒ v a → v a → v a
infixl 7 ⊘
```

Division is an operation that is undefined if the right operand is equal to zero. To capture this behaviour, the `Nothing` constructor from `Maybe` is used to represent errors. Both sides of the division operator are evaluated. If the right-hand side is zero, the division is not performed and an error is returned instead:

```
instance Div Maybe where
    (⊘) l r = l >>= λx→r >>= λy→
        if y == 0 then Nothing else Just (x `div` y)
```

2.2 Adding semantics

To add semantics to the DSL, the existing classes are implemented with a novel data type representing the view on the DSL. First a data type representing the semantics is defined. In this case, the printer is kept very simple for brevity and just defined as a **newtype** of a string to store the printed representation.[2] Since the language is typed, the printer data type has to have a type variable but it is only used during typing—i.e. a phantom type [Leijen and Meijer 2000]:

```
newtype Printer a = P { runPrinter :: String }
```

The class instances for `Expr` and `Div` for the pretty printer are straightforward and as follows:

```
instance Expr Printer where
    lit a = P (show a)
    (⊕) l r = P ("(" ++ runPrinter l
             ++ "+" ++ runPrinter r ++ ")")

instance Div Printer where
    (⊘) l r = P ("(" ++ runPrinter l
             ++ "/" ++ runPrinter r ++ ")")
```

2.3 Functions

Adding functions to the language is achieved by adding a multi-parameter class to the DSL. The type of the class function allows for the implementation to only allow first-order functions by supplying the arguments in a tuple. Furthermore, with the `:-` operator the syntax becomes useable. Finally, by defining the functions as a higher order abstract syntax (HOAS) type safety is achieved [Chlipala 2008; Pfenning and Elliott 1988]. The complete definition looks as follows:

```
class Function a v where
    fun :: ((a → v s) → In (a → v s) (v u)) → v u
data In a b = a :- b
infix 1 :-
```

The `Function` type class is now used to define functions with little syntactic overhead[3]. The following listing shows an expression in the DSL utilising two user-defined functions:

```
  fun λincrement→ (λx     →x ⊕ lit 1)
:- fun λdivide→   (λ(x, y)→x ⊘ y    )
:- increment (divide (lit 38, lit 5))
```


[2] In this case a **newtype** is used instead of regular **data** declarations. **newtype**s are special data types only consisting a single constructor with one field to which the type is isomorphic. During compilation the constructor is completely removed resulting in no overhead [Peyton Jones 2003, §4.2.3].

[3] The BlockArguments extension of GHC is used to reduce the number of brackets that allows lambda's to be an argument to a function without brackets or explicit function application using $

The interpreter only requires one instance of the `Function` class that works for any argument type. In the implementation, the resulting function `g` is simultaneously provided to the definition `def`. Because the laziness of Haskell's lazy let bindings, this results in a fixed point calculation:

```
instance Function a Maybe where
  fun def = let g :- m = def g in m
```

The given `Printer` type is not sufficient to implement the instances for the `Function` class, it must be possible to generate fresh function names. After extending the `Printer` type to contain some sort of state to generate fresh function names and a `MonadWriter [String]`[4] to streamline the output, we define an instance for every arity.

To illustrate this, the instance for unary functions is shown, all other arities are implemented in similar fashion.

```
instance Function () Printer where ...
instance Function (Printer a) Printer where ...
  fun def = freshLabel >>= λf→
    let g :- m = def $ λa0→const ⊥
          <$> (tell ["f", show f, " ("]
                >> a0 >> tell [")"])
    in  tell ["let f", f, " a0 = "]
          >> g (const ⊥ <$> tell ["a0"])
    >>  tell [" in "] >> m
instance Function (Printer a, Printer b) Printer where ...
```

Running the given printer on the example code shown before produces roughly the following output, running the interpreter on this code results in `Just 8`.

2.4 Data types

Lifting values from the host language to the DSL is possible using the `lit` function as long as the type of the value has instances for all the class constraints. Unfortunately, once lifted, it is not possible to do anything with values of the user-defined data type other than passing them around. It is not possible to construct new values from expressions in the DSL, to deconstruct a value into the fields, nor to test of which constructor the value is. Furthermore, while in our language the only constraint is the automatically derivable `Show`, in real-world languages the class constraints may be very difficult to satisfy for complex types, for example serialisation to a single stack cell in the case of a compiler.

As a consequence, for user-defined data types—such as a programmer-defined list type[5]—to become first-class citizens in the DSL, language constructs for constructors, deconstructors and constructor predicates must be defined. Field selectors are also useful functions for working with user-defined data types, they are not considered for the sake of brevity but can be implemented using the deconstructor functions. The constructs for the list type would result in the following class definition:

```
class ListDSL v where
  -- constructors
  nil    :: v (List a)
  cons   :: v a → v (List a) → v (List a)
  -- deconstructors
  unNil  :: v (List a) → v b → v b
  unCons :: v (List a)
         → (v a → v (List a) → v b) → v b
  -- constructor predicates
  isNil  :: v (List a) → v Bool
  isCons :: v (List a) → v Bool
```

Furthermore, instances for the DSL's views need to be created. For example, to use the interpreter, the following instance must be available. Note that at first glance, it would feel natural to have `isNil` and `isCons` return `Nothing` since we are in the `Maybe` monad. However, this would fail the entire expression and the idea is that the constructor test can be done from within the DSL.

```
instance ListDSL Maybe where
  nil        = Just Nil
  cons hd tl = Cons <$> hd <*> tl
  unNil  d f = d >>= λNil→f
  unCons d f = d
    >>= λ(Cons hd tl)→f (Just hd) (Just tl)
  isNil  d   = d >>= λcase[6]
    Nil → Just True
    _   → Just False
  isCons d   = d >>= λcase
    Cons _ _ → Just True
    Nil      → Just False
```

Adding these classes and their corresponding instances is tedious and results in boilerplate code. We therefore resort to metaprogramming, and in particular Template Haskell [Sheard and Jones 2002] to alleviate this burden.

3 TEMPLATE METAPROGRAMMING

Metaprogramming is a special flavour of programming where programs have the ability to treat and manipulate programs or program fragments as data. There are several techniques to facilitate metaprogramming, moreover it has been around for many years now [Lilis and Savidis 2019]. Even though it has been around for many years, it is considered complex [Sheard 2001].

Template Haskell is GHC's de facto metaprogramming system, implemented as a compiler extension together with a library [Sheard and Jones 2002][Team 2021, §6.13.1]. Readers already familiar with Template Haskell can safely skip this section.

Template Haskell adds four main concepts to the language, namely AST data types, splicing, quasiquotation and reification. With this machinery, regular Haskell functions can be defined that are called at compile time, inserting generated code into the AST. These functions are monadic functions operating in the **Q** monad. The **Q** monad facilitates failure, reification and fresh identifier generation for hygienic macros [Kohlbecker et al. 1986]. Within the **Q** monad,

[4] `freshLabel :: Printer String`
`tell :: MonadWriter w m ⇒ w → m ()`

[5] For example: `data List a = Nil | Cons {hd :: a, tl :: List a}`

[6] `λcase` is an abbreviation for `λx→case x of ...` when using GHC's LambdaCase extension.

capturable and non-capturable identifiers can be generated using the `mkName` and `newName` functions respectively. The *Peter Parker principle*[7] holds for the **Q** monad as well because it executes at compile time and is very powerful. For example, it can subvert module boundaries, thus accessing constructors that were hidden; access the structure of abstract types; and it may cause side effects during compilation because it is possible to call **IO** operations [Terei et al. 2012]. To achieve the goal of embedding data types in a DSL we refrain from using these *unsafe* features.

3.0.1 Data types. Firstly, for all of Haskell's AST elements, data types are provided that are mostly isomorphic to the actual data types used in the compiler. With these data types, the entire syntax of a Haskell program can be specified. Often, a data type is suffixed with the context, e.g. there is a `VarE` and a `VarP` for a variable in an expression or in a pattern respectively. To give an impression of these data types, a selection of data types available in Template Haskell is given below:

```
data Dec = FunD Name [Clause] | DataD Cxt Name ...
    | SigD Name Type | ClassD Cxt Name | ...
data Clause = Clause [Pat] Body [Dec]
data Pat = LitP Lit | VarP Name | TupP [Pat]
    | WildP | ListP [Pat] | ...
data Body = GuardedB [(Guard, Exp)] | NormalB Exp
data Guard = NormalG Exp | PatG [Stmt]
data Exp = VarE Name | LitE Lit | AppE Exp Exp
    | TupE [Maybe Exp] | LamE [Pat] Exp | ...
data Lit = CharL Char | StringL String
    | IntegerL Integer | ...
```

To ease creating AST data types in the **Q** monad, lowercase variants of the constructors are available that lift the constructor to the **Q** monad. For example, for the `LamE` constructor, the following `lamE` function is available.

```
lamE :: [Q Pat] → Q Exp → Q Exp
lamE ps es = LamE <$> sequence ps <*> es
```

3.0.2 Splicing. Special splicing syntax (`λ$(...)`) marks functions for compile-time execution. Other than that they always produce a value of an AST data type, they are regular functions. Depending on the context and location of the splice, the result type is either a list of declarations, a type, an expression or a pattern. The result of this function, when successful, is then spliced into the code and treated as regular code by the compiler. Consequently, the code that is generated may not be type safe, in which case the compiler provides a type error on the generated code. The following listing shows an example of a Template Haskell function generating on-the-fly functions for arbitrary field selection in a tuple. When called as `λ$(tsel 2 4)` it expands at compile time to `\\(_, _, f, _)→ f`:

```
tsel :: Int → Int → Q Exp
tsel field total = do
    f ← newName "f"
    lamE [ tupP [if i == field then varP f else wildP
         | i←[0..total-1]]] (varE f)
```

3.0.3 Quasiquotation. Another key concept of Template Haskell is Quasiquotation, the dual of splicing [Bawden 1999]. While it is possible to construct entire programs using the provided data types, it is a little cumbersome. Using *Oxford brackets* or single or double apostrophes, verbatim Haskell code can be entered that is converted automatically to the corresponding AST nodes easing the creation of language constructs. Depending on the context, different quasiquotes are used: •⟦...⟧ or ⟦$_e$ ...⟧ for expressions •⟦$_d$ ...⟧ for declarations •⟦$_p$ ...⟧ for patterns •⟦$_t$ ...⟧ for types •' ... for function names •" ... for type names

It is possible to escape the quasiquotes again by splicing. Variables defined within quasiquotes are always fresh—as if defined with `newName`—but it is possible to capture identifiers using `mkName`. For example, ⟦`\\x→x`⟧ translates to `newName "x" >>= \\x→lamE [varP x] (varE x)` and does not interfere with other `x`s already defined.

3.0.4 Reification. Reification is the act of querying the compiler for information about a certain name. For example, reifying a type name results in information about the type and the corresponding AST nodes of the type's definition. This information can then be used to generate code according to the structure of data types. Reification is done using the `reify :: Name → Q Info` function. The `Info` type is an ADT containing all the—known to the compiler—information about the matching type: constructors, instances, etc.

4 METAPROGRAMMING FOR GENERATING DSL FUNCTIONS

With the power of metaprogramming, we can generate the boilerplate code for our user-defined data types automatically at compile time. To generate the code required for the DSL, we define the `genDSL` function. The type belonging to the name passed as an argument to this function is made available for the DSL by generating the `typeDSL` class and view instances. For the `List` type it is called as: `λ$(genDSL "List)`.[8]

The `genDSL` function is a regular function—though Template Haskell requires that it is defined in a separate module—that has type: `Name → Q [Dec]`, i.e. given a name, it produces a list of declarations in the **Q** monad. The `genDSL` function first reifies the name to retrieve the structural information. If the name matches a type constructor containing a data type declaration, the structure of the type—the type variables, the type name and information about the constructors[9]—are passed to the `genDSL'` function. The `getConsName` function filters out unsupported data types such as GADTs and makes sure that every field has a name. For regular ADTs, the `adtFieldName` function is used to generate a name for the constructor based on the indices of the fields.[10] From this structure of the type, `genDSL'` generates a list of declarations containing a class definition (Section 4.1), instances for the interpreter (Section 4.2), and instances of the printer (Section 4.3) respectively.

```
genDSL :: Name → Q [Dec]
```

[7] With great power comes great responsibility.

[8] `"` is used instead of `'` to instruct the compiler to look up the information for `List` as a type and not as a constructor.

[9] Defined as **type** `VarBangType = (Name, Bang, Type)` by Template Haskell.

[10] `adtFieldName :: Name → Integer → Name`

```
genDSL name = reify name ≫= λcase
    TyConI (DataD cxt typeName tvs mkind
                  constructors derives)
        → mapM getConsName constructors
            ≫= λd→genDSL' tvs typeName d
    t → fail ("genDSL does not support: " ++ show t)

getConsName :: Con → Q (Name, [VarBangType])
getConsName (NormalC consName fs) = pure (consName,
    [(adtFieldName consName i, b, t)
    | (i, (b, t))←[0..] `zip` fs])
getConsName (RecC consName fs) = pure (consName, fs)
getConsName c
    = fail ("genDSL does not support: " ++ show c)

genDSL' :: [TyVarBndr] → Name → [(Name, [VarBangType])]
    → Q [Dec]
genDSL' typeVars typeName constructors = sequence
    [ mkClass, mkInterpreter, mkPrinter, ... ]
  where
    (consNames, fields) = unzip constructors
    ...
```

4.1 Class generation

The function for generating the class definition is defined in the **where** clause of the `genDSL'` function. Using the `classD` constructor, a single type class is created with a single type variable `v`. The `classD` function takes five arguments: (1) a context, i.e. the class constraints, which is empty in this case (2) a name, generated from the type name using the `className` function that simply appends the text `DSL` (3) a list of type variables, in this case the only type variable is the view on the DSL, i.e. `v` (4) functional dependencies, empty in our case (5) a list of function declarations, i.e. the class members, in this case it is a concatenation of the constructors, deconstructors, and constructor predicates Depending on the required information, either `zipWith` or `map` is used to apply the generation function to all constructors.

```
mkClass :: Q Dec
mkClass = classD (cxt []) (className typeName) [PlainTV (
     mkName "v")] []
    (  zipWith mkConstructor   consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ map     mkPredicate     consNames
    )
```

In all class members, the view `v` plays a crucial role. Therefore, a definition for `v` is accessible for all generation functions. Furthermore, the `res` type represents the *result* type, it is defined as the type including all type variables. This result type is derived from the type name and the list of type variables. In case of the `List` type, `res` is defined as `v (List a)` and is available for as well:

```
v = varT (mkName "v")
res = v `appT` foldl appT (conT typeName)
    (map getName typeVars)
  where getName (PlainTV name)    = varT name
        getName (KindedTV name _) = varT name
```

4.1.1 Constructors. The constructor definitions are generated from just the constructor names and the field information. All class members are defined using the `sigD` constructor that represents a function signature. The first argument is the name of the constructor function, a lowercase variant of the actual constructor name generated using the `constructorName` function. The second argument is the type of the function. A constructor C_k of type T where $T\ tv_0\ \ldots\ tv_n = \ldots \mid C_k\ a_0\ \ldots\ a_m \mid \ldots$ is defined as a DSL function $c_k :: v\ a_0 \to \ldots \to v\ a_m \to v\ (T\ v_0\ \ldots\ v_n)$. In the implementation, first the view `v` is applied to all the field types. Then, the constructor type is constructed by folding over the lifted field types with the result type as the initial value using `mkCFun`.

```
mkConstructor :: Name → [VarBangType] → Q Dec
mkConstructor n fs
    = sigD (constructorName n) (mkCFun fs res)

mkCFun :: [VarBangType] → Q Type → Q Type
mkCFun fs res = foldr (λx y→[|t$x → $y|])
    (map (λ(_, _, t)→v `appT` pure t) fs)
```

4.1.2 Deconstructors. The deconstructor is generated similarly to the constructor as the function for generating the constructor is the second argument modulo change in the result type. A deconstructor C_k of type T is defined as a DSL function $unC_k :: v\ (T\ v_0 \ldots v_n) \to (v\ a_0 \to \ldots \to v\ a_m \to v\ b) \to v\ b$. In the implementation, `mkCFun` is reused to construct the type of the deconstructor as follows:

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor n fs = sigD (deconstructorName n)
    [|t$res → $(mkCFun fs [|t$v $b|]) → $v $b|]
  where b = varT (mkName "b")
```

4.1.3 Constructor predicates. The last part of the class definition are the constructor predicates, a function that checks whether the provided value of type T contains a value with constructor C_k. A constructor predicate for constructor C_k of type T is defined as a DSL function $isC_k :: v\ (T\ v_0\ \ldots\ v_n) \to v\ Bool$. A constructor predicate—name prefixed by `is`—is generated for all constructors. They all have the same type:

```
mkPredicate :: Name → Q Dec
mkPredicate n = sigD (predicateName n)
    [|t$res → $v Bool|]
```

4.2 Interpreter instance generation

Generating the interpreter for the DSL means generating the class instance for the `Interpreter` data type using the `instanceD` function. The first argument of the instance is the context, this is left empty. The second argument of the instance is the type, the `Interpreter` data type applied to the class name. Finally, the class function instances are generated using the information derived from the structure of the type. The structure for generating the

function instances is very similar to the class definition, only for the function instances of the constructor predicates, the field information is required as well as the constructor names.

```
mkInterpreter :: Q Dec
mkInterpreter = instanceD (cxt [])
        [[t $(conT (className typeName)) Interpreter]]
    (  zipWith mkConstructor consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ zipWith mkPredicate consNames fields)
  where ...
```

4.2.1 Constructors. The interpreter is a view on the DSL that immediately executes all operations in the `Maybe` monad. Therefore, the constructor function can be implemented by lifting the actual constructor to the `Maybe` type using sequential application. I.e. for a constructor C_k this results in the following constructor: `ck a0 ... am = pure Ck <*> a0 <*> ... <*> am`. To avoid accidental shadowing, fresh names for all the arguments are generated. The `ifx` function is used as a shorthand for defining infix expressions[11]

```
mkConstructor :: Name → [VarBangType] → Q Dec
mkConstructor consName fs = do
    fresh ← sequence [newName "a" | _←fs]
    fun (constructorName consName) (map varP fresh)
        (foldl (ifx "<*>") [[pure $(conE consName)]]
        (map varE fresh))
```

4.2.2 Deconstructors. In the case of a deconstructor a function with two arguments is created: the object itself (`f`) and the function doing something with the individual fields (`d`). To avoid accidental shadowing first fresh names for the arguments and fields are generated. Then, a function is created with the two arguments. First `d` is evaluated and bound to a host language function that deconstructs the constructor and passes the fields to `f`. I.e. a deconstructor function C_k is defined as: `unCk d f = d >>= \\(Ck a0 .. am)→ f (pure a0) ... (pure am))`.[12]

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor consName fs = do
    d ← newName "d"
    f ← newName "f"
    fresh ← mapM (newName . nameBase . fst3) fs
    fun (deconstructorName consName) [varP d, varP f]
        [[$(varE d) >>= λ($(match f))→$(fapp f fresh)]]
  where fapp f = foldl appE (varE f)
          . map (λf→[[pure $(varE f)]])
        match f = pure (ConP consName (map VarP f))
```

ifx :: String → Q Exp → Q Exp → Q Exp

[11] `ifx op a b = infixE (Just a) (varE (mkName op)) (Just b)`

[12] The `nameBase :: Name → String` function from the Template Haskell library is used to convert a name to a string.

4.2.3 Constructor predicates. Constructor predicates evaluate the argument and make a case distinction on the result to determine the constructor. To be able to generate a valid pattern in the case distinction, the total number of fields must be known. To avoid having to explicitly generate a fresh name for the first argument, a lambda function is used. In general, the constructor selector for C_k results in the following code `isCk f = f >>= \\case Ck _ ... _ → pure True; _ → pure False`. Generating this code is done with the following function:

```
mkPredicate :: Name → [(Var, Bang, Type)] → Q Dec
mkPredicate n fs = fun (predicateName n) []
    [[λx→x >>= λcase
        $(conP n [wildP | _←fs]) → pure True
        _                        → pure False]]
```

4.3 Pretty printer instance generation

Generating the printer happen analogously to the interpreter, a class instance for the `Printer` data type using the `instanceD` function.

```
mkPrinter :: Q Dec
mkPrinter = instanceD (cxt []) [[t $(conT (className
    typeName)) Printer]]
    (  zipWith mkConstructor consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ map mkPredicate consNames)
```

To be able to define a printer that is somewhat more powerful, we provide instances for `MonadWriter`; add a state for fresh variables and a context; and define some helper functions the `Printer` datatype. The `printLit` function is a variant of `MonadWriters tell` that prints a literal string, but it can be of any type (it is a phantom type anyway). `printCons` prints a constructor name followed by an expression, it inserts parenthesis only when required depending on the state. `paren` always prints parenthesis around the given printer. `>>>` is a variant of the sequence operator `>>` from the `Monad` class, it prints whitespace in between the arguments.

```
printLit  :: String → Printer a
printCons :: String → Printer a → Printer a
paren     :: Printer a → Printer a
(>>>)     :: Printer a1 → Printer a2 → Printer a3
pl        :: String → Q Exp
```

4.3.1 Constructors. For a constructor C_k the printer is defined as: `ck a0 ... am = printCons "Ck" (printLit "" >>> a0 >>> ... >>> am)`. To generate the second argument to the `printCons` function, a fold is used with `printLit ""` as the initial element to account for constructors without any fields as well, e.g. `Nil` is translated to `nil = printCons "Nil" (printLit "")`.

```
mkConstructor :: Name → [VarBangType] → Q Dec
mkConstructor consName fs = do
    fresh ← sequence [newName "f" | _← fs]
    fun (constructorName consName) (map varP fresh)
        (pcons `appE` pargs fresh)
  where pcons = [[printCons $(lift (nameBase consName))]]
        pargs fresh = foldl (ifx ">>> ") (pl "") fresh
```

```
        (map varE fresh)
```

4.3.2 Deconstructors. Printing the deconstructor for C_k is defined as:

```
unCk d f
  =  printLit "unCk d"
  >> paren (
       printLit "λ(Ck" >> printLit "a0 . am" >>
   printLit ")→"
     >> f (printLit "a0") ... (printLit "am")
  )
```

The implementation for this is a little elaborate and it heavily uses the `pl` function, a helper function that translates a string literal `s` to ⟦`printLit λ$(lift s)`⟧, i.e. it lifts the `printLit` function to the Template Haskell domain.

```
mkDeconstructor :: Name → [VarBangType] → Q Dec
mkDeconstructor consName fs = do
    d ← newName "d"
    f ← newName "f"
    fresh ← sequence [newName "a" | _←fs]
    fun (deconstructorName consName) (map varP [d, f])
        ⟦ $(pl (nameBase (deconstructorName consName)))
        >> $(pl (nameBase d))
        >> paren ($(pl ('\\':'(':nameBase consName))
                  >> $lam >> printLit ")→"
                  >>  $(hoas f))⟧
  where
    lam = pl $ unwords [nameBase f | (f, _, _)←fs]
    hoas f = foldl appE (varE f)
        [pl (nameBase f) | (f, _, _)←fs]
```

4.3.3 Constructor predicates. For the printer, the constructor selector for C_k results in the following code `isCk f = printLit "isCk" >> f`.

```
mkPredicate :: Name → Q Dec
mkPredicate n = fun (predicateName n) []
    ⟦λx→ $(pl $ nameBase $ predicateName n) >> x⟧
```

5 PATTERN MATCHING

It is possible to construct and deconstruct values from other DSL expressions, and to perform tests on the constructor but with a clunky and unwieldy syntax. They have become first-class citizens in a grotesque way. For example, given that we have some language constructs to denote failure and conditionals,[13] writing a list summation function in our DSL would be done as follows. For the sake of the argument we take a little shortcut here and assume that the interpretation of the DSL supports lazy evaluation by using the host language as a metaprogramming language as well, allowing us to use functions in the host language to construct expressions in the DSL.

```
program :: (ListDSL v, Support v, ...) ⇒ v Int
program
    = fun λsum→(λl→if'(isNil l)
        (lit 0)
        (unCons l (λhd tl→hd ⊕ sum tl)))
    :- sum (cons (lit 38) (cons (lit 4) nil))
```

A similar Haskell implementation is much more elegant and less cluttered because of the support for pattern matching. Pattern matching offers a convenient syntax for doing deconstruction and constructor tests at the same time.

```
sum :: List Int → Int
sum Nil = 0
sum (List hd tl) = hd + sum tl

main = sum (Cons 38 (Cons 4 Nil))
```

5.1 Custom quasiquoters

The syntax burden of eDSLs can be reduced using quasiquotation. In Template Haskell, quasiquotation is a convenient way to create Haskell language constructs by entering them verbatim using Oxford brackets. However, it is also possible to create so-called custom quasiquoters [Mainland 2007]. If the programmer writes down a fragment of code between *tagged* Oxford brackets, the compiler executes the associated quasiquoter functions at compile time. A quasiquoter is a value of the following data type:

```
data QuasiQuoter = QuasiQuoter
    { quoteExp  :: String → Q Exp
    , quotePat  :: String → Q Pat
    , quoteType :: String → Q Type
    , quoteDec  :: String → Q Dec
    }
```

The code between *dsl* brackets (⟦$_{dsl}$ ...⟧) is preprocessed by the `dsl` quasiquoter. Because the functions are executed at compile time, errors—thrown using the `MonadFail` instance of the **Q** monad—in these functions result in compile time errors. The AST nodes produced by the quasiquoter are inserted into the location and checked as if they were written by the programmer.

To illustrate writing a custom quasiquoter, we show an implementation of a quasiquoter for binary literals. The `bin` quasiquoter is only defined for expressions and parses subsequent zeros and ones as a binary number and splices it back in the code as a regular integer. Thus, ⟦$_{bin}$ `101010`⟧ results in the literal integer expression `42`. If an invalid character is used, a compile-time error is shown. The quasiquoter is defined as follows:

```
bin :: QuasiQuoter
bin = QuasiQuoter { quoteExp = parseBin }
  where
    parseBin :: String → Q Exp
    parseBin s = LitE . IntegerL <$> foldM bindigit 0 s

    bindigit :: Integer → Char → Q Integer
    bindigit acc '0' = pure (2 * acc)
```

```
class Support v where
    if'    :: v Bool → v a → v a → v a
    bottom :: String → v a
```

[13]

```
bindigit acc '1' = pure (2 * acc + 1)
bindigit acc c = fail ("invalid char: " ++ show c)
```

5.2 Quasiquotation for pattern matching

Custom quasiquoters allow the DSL user to enter fragments verbatim, bypassing the syntax of the host language. Pattern matching in general is not suitable for a custom quasiquoter because it does not really fit in one of the four syntactic categories for which custom quasiquoter support is available. However, a concrete use of pattern matching, interesting enough to be beneficial, but simple enough for a demonstration is the *simple case expression*, a case expression that does not contain nested patterns and is always exhaustive. They correspond to multi-way conditional expressions and can thus be converted to DSL constructs straightforwardly [Peyton Jones 1987, §4.4].

In contrast to the binary literal quasiquoter example, we do not create the parser by hand. The parser combinator library *parsec* is used instead to ease the creation of the parser [Leijen and Meijer 2001]. First the location of the quasiquoted code is retrieved using the `location` function that operates in the **Q** monad. This location is inserted in the parsec parser so that errors are localised in the source code. Then, the `expr` parser is called that returns an `Exp` in the **Q** monad. The `expr` parser uses parsec's commodity expression parser primitive `buildExpressionParser`. The resulting parser translates the string directly into Template Haskell's AST data types in the **Q** monad. The most interesting parser is the parser for the case expression that is an alternative in the basic expression parser `basic`. A case expression is parsed when a keyword **case** is followed by an expression that is in turn followed by a non-empty list of matches. A match is parsed when a pattern (`pat`) is followed by an arrow and an expression. The results of this parser are fed into the `mkCase` function that transforms the case into an expression using DSL primitives such as conditionals, deconstructors and constructor predicates. The above translates to the following skeleton implementation:

```
expr :: Parser (Q Exp)
expr = buildExpressionParser [...] basic
  where
    basic :: Parser (Q Exp)
    basic =  ...
        <|> mkCase <$ reserved "case" <*> expr
                   <* reserved "of"   <*> many1 match
        <|> ...

    match :: Parser (Q Pat, Q Exp)
    match = (,) <$> pat <* reserved "→" <*> expr

    pat :: Parser (Q Pat)
    pat = conP <$> con <*> many var
```

Case expressions are transformed into constructors, deconstructors and constructor predicates, e.g. **case** `e1` **of** `Cons hd tl → e2; Nil → e3;` is converted to:

```
if' (isList e1)
    (unCons e1 (λhd tl→e2))
    (if' (isNil e1)
        (unNil e1 e3)
        (bottom "Exhausted case"))
```

The `mkCase` (line 1) function transforms a case expression into constructors, deconstructors and constructor predicates. Line 3 first evaluates the patterns. Then the patterns and their expressions are folded using the `mkCase`` function (line 5). While a case exhaustion error is used as the initial value, this is never called since all case expressions are exhaustive. For every case, code is generated that checks whether the constructor used in the pattern matches the constructor of the value using constructor predicates (line 11). If the constructor matches, the deconstructor (line 12) is used to bind all names to the correct identifiers and evaluate the expression. If the constructor does not match, the continuation (`λ$rest`) is used (line 9).

```
1  mkCase :: Q Exp → [(Q Pat, Q Exp)] → Q Exp
2  mkCase name cases = do
3      pats ← mapM fst cases
4      foldr (uncurry mkCase') ⟦bottom "Exhausted case"⟧
5          (zip pats (map snd cases))
6    where
7      mkCase' :: Pat → Q Exp → Q Exp → Q Exp
8      mkCase' (ConP cons fs) e rest
9          = ⟦if' $pred $then_ $rest⟧
10       where
11         pred  = varE (predicateName cons) `appE` name
12         then_ = ⟦$(varE (deconstructorName cons))
13                  $name $(lamE [pure f | f←fs] e)⟧
```

Finally, with this quasiquotation mechanism we can define our list summation using a case expression. As a byproduct, syntactic cruft such as the special symbols for the operators and calls to `lit` can be removed as well resulting in the following summation implementation:

```
program :: (ListDSL v, DSL v, ...) ⇒ v Int
program
    = fun λsum→(λl→ ⟦dsl case l of
        Cons hd tl → hd + sum tl
        Nil        → 0⟧)
    :- sum (cons (lit 38) (cons (lit 4) nil))
```

6 RELATED WORK

Generic or polytypic programming is a promising technique at first glance for automating the generation of function implementations [Lämmel and Jones 2003]. However, while it is possible to define a function that works on all first-order types, adding a new function with a new name to the language is not possible. This does not mean that generic programming is not useable for embedding pattern matches. In generic programming, types are represented as sums of products and using this representation it is possible to define pattern matching functions.

For example, Rhiger showed a method for expressing statically typed pattern matching using typed higher-order functions [Rhiger

2009]. If not the host language but the DSL contains higher order functions, the same technique could be applied to port pattern matching to DSLs though using an explicit sums of products representation. Atkey et al. describe embedding pattern matching in a DSL by giving patterns an explicit representation in the DSL by using pairs, sums and injections [Atkey et al. 2009, §3.3].

McDonell et al. extends on this idea, resulting in a very similar but different solution to ours [McDonell et al. 2021]. They used the technique that Atkey et al. showed and applied it to deep embedding using the concrete syntax of the host language. The scaffolding—e.g. generating the pairs, sums and injections—for embedding is automated using generics but the required pattern synonyms are generated using Template Haskell. The key difference to our approach is that we specialise the implementation for each of the backends instead of providing a general implementation of data type handling operations. Furthermore, our implementation does not require a generic function to trace all constructors, resulting in problems with (mutual) recursion.

Young et al. added pattern matching to a deeply embedded DSL using a compiler plugin [Young et al. 2021]. This plugin implements an `externalise :: a → E a` function that allows lifting all machinery required for pattern matching automatically from the host language to the DSL. Under the hood, this function translates the pattern match to constructors, deconstructors, and constructor predicates. The main difference with this work is that it requires a compiler plugin while our metaprogramming approach works on any compiler supporting a metaprogramming system similar to Template Haskell.

6.1 Related work on Template Haskell

Metaprogramming in general is a very broad research topic and has been around for years already. We therefore do not claim an exhaustive overview of related work on all aspects of metaprogramming. However, we have tried to present most research on metaprogramming in Template Haskell. Czarnecki et al. provide a more detailed comparison of different metaprogramming techniques. They compare staged interpreters, metaprogramming and templating by comparing MetaOCaml, Template Haskell and C++ templates [Czarnecki et al. 2004]. Template Haskell has been used to implement related work. They all differ slightly in functionality from our domain and can be divided into several categories.

6.1.1 Generating extra code. Using Template Haskell or other metaprogramming systems it is possible to add extra code to your program. The original Template Haskell paper showed that it is possible to create variadic functions such as `printf` using Template Haskell that would be almost impossible to define without [Sheard and Jones 2002]. Hammond et al. used Template Haskell to generate parallel programming skeletons [Hammond et al. 2003]. In practise, this means that the programmer selects a skeleton and, at compile time, the code is massaged to suit the pattern and information about the environment is inlined for optimisation.

Polak et al. implemented automatic GUI generation using Template Haskell [Polak and Jarosz 2006]. Duregård et al. wrote a parser generator using Template Haskell and the custom quasiquoting facilities [Duregård and Jansson 2011]. From a specification of the grammar, given in verbatim using a custom quasiquoter, a parser is generated at compile time. Shioda et al. used metaprogramming in the D programming language to create a DSL toolkit [Shioda et al. 2014]. They also programmatically generate parsers and a backend for either compiling or interpreting the IR. Blanchette et al. use Template Haskell to simplify the development of Liquid Haskell proofs [Blanchette et al. 2022]. Folmer et al. used Template Haskell to synthesize CλaSH [Baaij 2015] abstract syntax trees to be processed [Folmer et al. 2022]. In similar fashion, Materzok used Template Haskell to translate YieldFSM programs to CλaSH [Materzok 2022].

6.1.2 Optimisation. Besides generating code, it is also possible to analyse existing code and perform optimisations. Yet, this is dangerous territory because unwantedly the semantics of the optimised program may be slightly different from the original program. For example, Lynagh implemented various optimisations in Template Haskell such as automatic loop unrolling [Lynagh 2003]. The compile-time executed functions analyse the recursive function and unroll the recursion to a fixed depth to trade execution speed for program space. Also, O'Donnell embedded Hydra, a hardware description language, in Haskell utilising Template Haskell [O'Donnell 2004]. Using intensional analysis of the AST, it detects cycles by labelling nodes automatically so that it can generate *netlists*. The authors mention that alternatively this could have be done using a monad but this hampers equational reasoning greatly, which is a key property of Hydra. Finally, Viera et al. present a way of embedding attribute grammars in Haskell in a staged fashion [Viera et al. 2018]. Checking several aspects of the grammar is done at compile time using Template Haskell while other safety checks are performed at runtime.

6.1.3 Compiler extension. Sometimes, expressing certain functionalities in the host languages requires a lot of boilerplate, syntax wrestling, or other pains. Metaprogramming can relieve some of this stress by performing this translation to core constructs automatically. For example, implementing generic—or polytypic— functions in the compiler is a major effort. Norell et al. used Template Haskell to implement the machinery required to implement generic functions at compile time [Norell and Jansson 2004]. Adams et al. also explores implementing generic programming using Template Haskell to speed things up considerably compared to regular generic programming [Adams and DuBuisson 2012]. Clifton et al. use Template Haskell with a custom quasiquoter to offer skeletons for workflows and embed foreign function interfaces in a DSL [Clifton-Everest et al. 2014]. Eisenberg et al. showed that it is possible to programmatically lift some functions from the function domain to the type domain at compile time, i.e. type families [Eisenberg and Stolarek 2014]. Furthermore, Seefried et al. argued that it is difficult to do some optimisations in eDSLs and that metaprogramming can be of use there [Seefried et al. 2004]. They use Template Haskell to change all types to unboxed types, unroll loops to a certain depth and replace some expressions by equivalent more efficient ones. Torrano et al. showed that it is possible to use Template Haskell to perform a strictness analysis and perform let-to-case translation [Torrano and Segura 2005]. Both applications are examples of compiler extensions that can be implemented using Template Haskell. Another example of such a compiler extension is shown by Gill et al. [Gill 2009]. They created a meta level DSL to describe

rewrite rules on Haskell syntax that are applied on the source code at compile time.

6.1.4 Quasiquotation. By means of quasiquotation, the host language syntax that usually seeps through the embedding can be hidden. The original Template Haskell quasiquotation paper [Mainland 2007] shows how this can be done for regular expressions, not only resulting in a nicer syntax but syntax errors are also lifted to compile time instead of run time. Also, Kariotis et al. used Template Haskell to automatically construct monad stacks without having to resort to the monad transformers library which requires advanced type system extensions [Kariotis et al. 2008].

Najd uses the compile time to be able to do normalisation for a DSL, dubbing it QDSLs [Najd et al. 2016]. They utilise the quasiquation facilities of Template Haskell to convert Haskell DSL code to constructs in the DSL, applying optimisations such as eliminating lambda abstractions and function applications along the way. Egi et al. extended Haskell to support non-free data type pattern matching—i.e. data type with no standard form, e.g. sets, graphs—using Template Haskell [Egi et al. 2022]. Using quasiquotation, they make a complicated embedding of non-linear pattern matching available through a simple lens.

6.1.5 Typed Template Haskell. Typed Template Haskell is a very recent extension/alternative to normal Template Haskell [Pickering et al. 2019; Xie et al. 2022]. Where in Template Haskell you can manipulate arbitrary parts of the syntax tree, add top-level splices of data types, definitions and functions, in Typed Template Haskell the programmer can only splice expressions but the abstract syntax tree fragments representing the expressions are well-typed by construction instead of untyped.

Pickering et al. implemented staged compilation for the *generics-sop* [de Vries and Löh 2014] generics library to improve the efficiency of the code using Typed Template Haskell [Pickering et al. 2020]. Willis et al. used Typed Template Haskell to remove the overhead of parsing combinators [Willis et al. 2020].

7 DISCUSSION

This paper aims to be twofold, first, it shows how to inherit data types in a DSL as first-class citizens by generating the boilerplate at compile time using Template Haskell. Secondly, it introduces the reader to Template Haskell by giving an overview of the literature in which Template Haskell is used and provides a gentle introduction by explaining the case study.

Functional programming languages are especially suitable for embedding DSLs but adding user-defined data types is still an issue. The tagless-final style of embedding offers great modularity, extensibility and flexibility. However, user-defined data types are awkward to handle because the built-in operations on them—construction, deconstruction and constructor tests—are not inherited from the host language. We showed how to create a Template Haskell function that will splice the required class definitions and view instances. The code dataset also contains an implementation for defining field selectors and provides an implementation for a compiler.[14] Furthermore, by writing a custom quasiquoter, pattern matches in natural syntax can be automatically converted to the internal representation of the DSL, thus removing the syntax burden of the facilities. The use of a custom quasiquoter does require the DSL programmer to write a parser for their DSL, i.e. the parser is not inherited from the host language as is often the case in an embedded DSL. However, by making use of modern parser combinator libraries, this overhead is limited and errors are already caught at compilation.

The fact that Template Haskell is deemed *unsafe* or even *scary*, and that the learning curve is often perceived makes people hesitant to employ this powerful and useful tool. We found that, when familiar with the intricacies, implementing the non-trivial functionality went quite well. The error messages are reasonable and using quasiquotation, not a lot of AST data types have to be created.

7.1 Future work

For future work, it would be interesting to see how generating boilerplate for user-defined data types translates from shallow embedding to deep embedding. In deep embedding, the language constructs are expressed as data types in the host language. Adding new constructs, e.g. constructors, deconstructors, and constructor tests, for the user-defined data type therefore requires extending the data type. Techniques such as data types à la carte [Swierstra 2008] and open data types [Löh and Hinze 2006] show that it is possible to extend data types orthogonally but whether metaprogramming can still readily be used is something that needs to be researched. It may also be possible to implemented (parts) of the boilerplate generation using Typed Template Haskell (See Section 6.1.5) to achieve more confidence in the type correctness of the implementation.

Another venue of research is to try to find the limits of this technique regarding richer data type definitions. It would be interesting to see whether it is possible to apply the technique on data types with existentially quantified type variables or full-fledged generalised ADTs [Hinze 2003]. It is not possible to straightforwardly lift the deconstructors to type classes because existentially quantified type variables will escape. Rank-2 polymorphism offers tools to define the types in such a way that this is not the case anymore. However, implementing compiling views on the DSL is complicated because it would require inventing values of an existentially quantified type variable to satisfy the type system which is difficult.

Finally, having to write a parser for the DSL is extra work. Future research could determine whether it is possible to generate this using Template Haskell as well.

ACKNOWLEDGMENTS

This research is partly funded by the Royal Netherlands Navy. Furthermore, we would like to thank the anonymous reviewers for their invaluable comments.

REFERENCES

Michael Adams and Thomas DuBuisson. 2012. Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. Association for Computing Machinery, New York,

[14] Lubbers, M.; Koopman, P.; Plasmeijer, R. (2022): Code for the paper First-Class Data Types in Shallow Embedded Domain-Specific Languages using Metaprogramming. Zenodo. 10.5281/zenodo.6416747.

NY, USA, 13–24. https://doi.org/10.1145/2364506.2364509 event-place: Copenhagen, Denmark.

Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1596638.1596644 event-place: Edinburgh, Scotland.

Christiaan Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. PhD Thesis. University of Twente, Netherlands. https://doi.org/10.3990/1.9789036538039 ISBN: 978-90-365-3803-9.

Alan Bawden. 1999. Quasiquotation in Lisp. In *Olivier Danvy, Ed., University of Aarhus, Dept. of Computer Science (BRICS Notes Series, Vol. NS-99-1)*. BRICS, Aarhus, Denmark, 88–99. https://doi.org/10.1.1.22.1290

Henry Blanchette, Niki Vazou, and Leonidas Lampropoulos. 2022. Liquid Proof Macros. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3546189.3549921 event-place: Ljubljana, Slovenia.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205 Publisher: Cambridge University Press.

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226 event-place: Victoria, BC, Canada.

Robert Clifton-Everest, Trevor McDonell, Manuel Chakravarty, and Gabriele Keller. 2014. Embedding Foreign Code. In *Practical Aspects of Declarative Languages*, Matthew Flatt and Hai-Feng Guo (Eds.). Springer International Publishing, Cham, 136–151.

Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–72. https://doi.org/10.1007/978-3-540-25935-0_4

Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/2633628.2633634 event-place: Gothenburg, Sweden.

Jonas Duregård and Patrik Jansson. 2011. Embedded Parser Generators. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 107–117. https://doi.org/10.1145/2034675.2034689 event-place: Tokyo, Japan.

Satoshi Egi, Akira Kawata, Mayuko Kori, and Hiromi Ogawa. 2022. Embedding Non-linear Pattern Matching with Backtracking for Non-free Data Types into Haskell. *New Generation Computing* 40, 2 (July 2022), 481–506. https://doi.org/10.1007/s00354-022-00177-z

Richard Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361 event-place: Gothenburg, Sweden.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. 2003. Compiling embedded languages. *Journal of Functional Programming* 13, 3 (2003), 455–481. https://doi.org/10.1017/S0956796802004574 Publisher: Cambridge University Press.

Hendrik Folmer, Robert de Groote, and Marco Bekooij. 2022. High-Level Synthesis of Digital Circuits from Template Haskell and SDF-AP. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Marc Reichenbach, and Matthias Jung (Eds.). Springer International Publishing, Cham, 3–27.

Andy Gill. 2009. A Haskell Hosted DSL for Writing Transformation Systems. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–309.

Kevin Hammond, Jost Berthold, and Rita Loogen. 2003. Automatic Skeletons in Template Haskell. *Parallel Processing Letters* 13, 03 (2003), 413–424. https://doi.org/10.1142/S0129626403001380 _eprint: https://doi.org/10.1142/S0129626403001380.

Ralf Hinze. 2003. Fun With Phantom Types. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Bloomsbury Publishing, Palgrave, 245–262.

Pericles Kariotis, Adam Procter, and William Harrison. 2008. Making Monads First-Class with Template Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/1411286.1411300 event-place: Victoria, BC, Canada.

Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Jeremy Gibbons (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/319838.319859 event-place: Cambridge, Massachusetts, USA.

Daan Leijen and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/331960.331977 event-place: Austin, Texas, USA.

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Technical Report UU-CS-2001-27. Universiteit Utrecht, Utrecht. 22 pages.

Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6 (Oct. 2019). https://doi.org/10.1145/3354584 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Ian Lynagh. 2003. Unrolling and Simplifying Expressions with Template Haskell. http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179 event-place: New Orleans, Louisiana, USA.

Andres Löh and Ralf Hinze. 2006. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1140335.1140352 event-place: Venice, Italy.

Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1291201.1291211 event-place: Freiburg, Germany.

Marek Materzok. 2022. Generating Circuits with Generators. *Proc. ACM Program. Lang.* 6, ICFP (Aug. 2022). https://doi.org/10.1145/3549821 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Trevor McDonell, Joshua Meredith, and Gabriele Keller. 2021. Embedded Pattern Matching. *CoRR* abs/2108.13114 (2021). https://arxiv.org/abs/2108.13114 arXiv: 2108.13114.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-Specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2847538.2847541 event-place: St. Petersburg, FL, USA.

Ulf Norell and Patrik Jansson. 2004. Prototyping Generic Programming in Template Haskell. In *Mathematics of Program Construction*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 314–333.

John O'Donnell. 2004. Embedding a Hardware Description Language in Template Haskell. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 143–164. https://doi.org/10.1007/978-3-540-25935-0_9

Simon Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall, Hertfordshire. https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/

Simon Peyton Jones (Ed.). 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, Cambridge.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010 event-place: Atlanta, Georgia, USA.

Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged Sums of Products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell (Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3406088.3409021 event-place: Virtual Event, USA.

Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. Multi-Stage Programs in Context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/3331545.3342597 event-place: Berlin, Germany.

Gracjan Polak and Janusz Jarosz. 2006. Automatic Graphical User Interface Form Generation Using Template Haskell. In *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, United Kingdom, 19-21 April 2006 (Trends in Functional Programming, Vol. 7)*, Henrik Nilsson (Ed.). Intellect, Bristol, UK, 1–11. event-place: Nottingham, UK.

Morten Rhiger. 2009. Type-safe pattern combinators. *Journal of Functional Programming* 19, 2 (2009), 145–156. https://doi.org/10.1017/S0956796808007089 Publisher: Cambridge University Press.

Sean Seefried, Manuel Chakravarty, and Gabriele Keller. 2004. Optimising Embedded DSLs Using Template Haskell. In *Generative Programming and Component Engineering*, Gabor Karsai and Eelco Visser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–205.

Tim Sheard. 2001. Accomplishments and Research Challenges in Meta-programming. In *Semantics, Applications, and Implementation of Program Generation*, Walid Taha

(Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–44.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691 event-place: Pittsburgh, Pennsylvania.

Masato Shioda, Hideya Iwasaki, and Shigeyuki Sato. 2014. LibDSL: A Library for Developing Embedded Domain Specific Languages in d via Template Metaprogramming. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/2658761.2658770 event-place: Västerås, Sweden.

Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

GHC Team. 2021. GHC User's Guide Documentation. https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524 event-place: Copenhagen, Denmark.

Carmen Torrano and Clara Segura. 2005. Strictness Analysis and let-to-case Transformation using Template Haskell. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005 (Trends in Functional Programming, Vol. 6)*, Marko van Eekelen (Ed.). Intellect, Bristol, UK, 429–442. event-place: Talinn, Estonia.

Marcos Viera, Florent Balestrieri, and Alberto Pardo. 2018. A Staged Embedding of Attribute Grammars in Haskell. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3310232.3310235 event-place: Lowell, MA, USA.

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP (Aug. 2020). https://doi.org/10.1145/3409002 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with Class: A Specification for Typed Template Haskell. *Proc. ACM Program. Lang.* 6, POPL (Jan. 2022). https://doi.org/10.1145/3498723 Place: New York, NY, USA Publisher: Association for Computing Machinery.

David Young, Mark Grebe, and Andy Gill. 2021. On Adding Pattern Matching to Haskell-Based Deeply Embedded Domain Specific Languages. In *Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 20–36. https://doi.org/10.1007/978-3-030-67438-0_2 event-place: Copenhagen, Denmark.