Shallowly Embedded Functions

Mart Lubbers[0000−0002−4015−4878], Pieter Koopman[0000−0002−3688−0957], and Niek Janssen[0009−0003−7348−7788]

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands
mart@cs.ru.nl pieter@cs.ru.nl niek.janssen3@ru.nl

Abstract. Domain-specific languages, DSLs, enhance the productivity of programmers. The abstraction level of a DSL is tailored to an application domain to facilitate the production and maintenance of programs. Functions add an important abstraction and repetition mechanism to DSLs, just as for any other programming language. For the evaluation of embedded DSLs one can use functions in the host language for this purpose. However, for example in case of recursion, this use of host language functions may cause undesirable side effects for some views of the DSL, like pretty printing and code generation.
In this paper, we offer a systematic overview of the options for defining functions in an embedded DSL, in particular tagless-final, or class-based shallow embedding. These functions are type safe, require minimal syntactic overhead, and are suitable for multiple views of the DSL.

Keywords: Embedded Domain-Specific Languages· Static Typing· Functional Programming· Recursion

1 Introduction

A Domain-Specific Language, DSL, helps one to develop and maintain software for a particular application domain. An embedded DSL, eDSL, is implemented as a library in a host programming language. Preferably, an eDSL supports multiple interpretations, *views*, for example printing, optimization, evaluation, or compilation. It is desirable that the type safety of the host language is also available in the embedded DSL. Embedding a DSL saves us from making a complete standalone tool chain with parser, type checker, and so on for the DSL, as they are inherited from the host language. Typically, the host language becomes a powerful generator of eDSL programs [16]. Though some trade offs have to be made for the correct typing and sharing of identifiers, like function names, and when using them in function application.

Functional programming languages have shown to be well-suited host languages for embedded DSLs. This has been recognized long ago by Hudak [9] and others. Moreover, recent extensions of functional programming languages increase their quality as host languages for eDSLs. Among the properties contributing are the ability to define infix operators, generalized algebraic data types, multi-parameter type (constructor) classes and powerful function types.

In this paper, we focus on the definition of functions in eDSLs. Basically, functions can be handled like identifiers with a fancy type. In a naive implementation, identifiers are just a variable tag, either a constructor or a function, and some identifier. In that approach, the host language compiler cannot check that the variable is properly defined nor that the type is used correctly. On the other hand, the compiler can check variables if we represent them by functions or function arguments in the host language. This is an old idea known as Higher-Order Abstract Syntax, HOAS [26]. We use this concept to define functions in DSLs and thus reducing syntactic overhead and improving type safety.

1.1 Requirements for DSL Functions

For embedded functions, we have some requirements. Firstly, we want to have control over the types of function arguments. Even for polymorphic and overloaded functions, it should not be possible to use types that are not considered to be part of the DSL. For some applications, it is required to have an upper limit on the number of arguments of the function, e.g. when an evaluator of the eDSL needs to store thunks in some finite memory location or to disallow partial function applications. Similarly, we want to choose in the eDSL design between first-order and higher-order functions. Secondly, embedded functions should be able to call each other and themselves. We require at least recursion, and preferably mutual recursion. Thirdly, sharing in the host language should be maintained in the DSL [23]. For views that directly interpret terms in the DSL, sharing is automatically maintained, but for views such as printing or code generation, this could be made explicit. Finally, function application in the DSL resembles the function application in the host language and preferably does not require an explicit apply. That is, we prefer to write `f x` over `ap f x` in our eDSL to minimize the semantic friction and syntactic overhead.

1.2 Choice of Embedding

There are two main flavours for embedding DSLs, deep embedding and shallow embedding. In deep embedding, DSL constructs are represented as data types in the host language. Views are functions over these data types. In class-based shallow embedding, an extension of standard shallow embedding, the DSL constructs are type classes in the host language. Here, views are instances of these classes. Due to space restrictions, the deeply embedded version using generalized ADTs [25] version of our examples are only provided in the artefact and only the tagless-final implementation is shown.

1.3 Research Contribution

The research contribution of this paper is twofold. Firstly, it provides a systematic overview of the options for defining type-safe functions in shallowly embedded DSLs. All variants to obtain type-safety are based on the ability of

the host language to check functions and function arguments at compile time. We encode the embedded functions either as arguments or as functions in the host language. There are various ways to encode embedded functions, and our choices have a significant impact on what is allowed for our embedded functions.

Secondly, several novel ideas are described: how to deal with type annotations in a language without type ascriptions, encoding named arguments in embedded functions using record types, restrict functions to be only definable at the top level, and show how embedded functions can be made reusable without restricting their scope. All examples are given in the purely functional programming language Clean [4, 28] but can be applied to other languages as well.[1]

2 Class-based Shallow Embedding

In the coming sections, we show various versions of embedding functions. We use class-based embedding rather than a traditional shallow embedding to allow multiple views of the DSL. This class-based embedding is also known as tagless-final embedding [5].

We start with the class for the ordinary operators that is reused by all versions of the DSL. Infix operators are used with the usual binding power and associativity to beautify the syntax. We add a dot to the operator name whenever required to avoid name clashes with the host language.

```
class lit v :: a → v a | toString a
class arith v where
  (+.) infixl 6 :: (v a) (v a) → v a | + a
  (-.) infixl 6 :: (v a) (v a) → v a | - a
  (*.) infixl 7 :: (v a) (v a) → v a | * a
  (/.) infixl 7 :: (v a) (v a) → v a | / a
class bool v where
  (&&.) infixr 3 :: (v Bool) (v Bool) → v Bool
  (||.) infixr 3 :: (v Bool) (v Bool) → v Bool
class comp v where
  (==.) infix 4 :: (v a) (v a) → v Bool | == a
  (<.)  infix 4 :: (v a) (v a) → v Bool | < a
class If v :: (v Bool) (v a) (v a) → v a
```

We add type classes for local values (`cons`) and single argument functions (`fun1`).

```
class cons v ::  ((v a)      →In  (v a)       (v b)) → v b
class fun1 v :: (((v a)→v b)→In ((v a)→v b) (v c)) → v c

:: In a b = (In) infix 0 a b // a prettier tuple
```

The definition of the familiar factorial function serves as an example in `ex1_v2`.[2] There is no higher meaning to the trivial definition of `one`, it is just to illustrate the `cons` function. The argument of `ex1_v2` is the value of the argument of the factorial function, i.e. the value is inlined. This illustrates how the host language and the eDSL can mix. For recursive functions, like `fac`, it is essential that we combine the function definition and its application in a single

[1] A concise guide to Clean for Haskell programmers is found here [18].

construct. The defined function should be in scope of the application in its own body as well as the applications of the function.

```
ex1_v2 n =
  cons λone = lit 1 In
  fun1 λfac = (\n. If (n ==. one) one (n *. fac (n -. one))) In
  fac (lit n)
```

2.1 Evaluation

Evaluating these operators in the monad `E` is completely standard. This monad is an identity functor with a strict value but illustrates that it could work in any more complex monad. Some class instances are listed in Appendix A.

For the evaluation of the `fun1` and `cons` class, we uniformly substitute the body of the definition to all applied occurrences using a cyclic definition. Thanks to lazy evaluation, this works flawlessly.

```
instance lit E where lit a = pure a
instance arith E where
  (+.) x y = (+) <$> x <*> y
  (-.) x y = (-) <$> x <*> y
  ...
instance fun1 E where fun1 f = let (val In body) = f val in body
instance cons E where fun1 f = let (val In body) = f val in body
```

The effect of this circular use of the body at each applied occurrence of the function is very similar to the use of an explicit fixed-point combinator (see Section 2.3).

In shallow embedding, the view is already embedded in the data type. So evaluating expressions in our DSL is a matter of unpacking the datatype `E`, as seen in `eval2`.

```
eval2 :: (E a) → a
eval2 (E a) = a
```

The main function `eval (ex1_v2 4)` produces the value 24.

2.2 Printing

The printing view for this DSL is slightly more complex. We use the tooling shown in Appendix B that uses a reader writer state monad, RWS. Here, the writer is a list of strings, the state contains fresh identifiers and the reader is not used. Printing these language constructs in this monad is fairly straightforward. To generate concise prints we skip the keyword `lit` in the `Print` instance of the class `lit` and the suffixes of the infix operators. For brevity in presentation, we do not attempt to print minimal parentheses. Printing of the other basic classes is similar and shown in Appendix B.

[2] In Clean, →, = and . can be used for lambdas, we pick according to our taste.

```
instance lit Print where lit a = P (tell [toString a])
instance arith Print where
  (+.) x y = printBin x y "+"
  (-.) x y = printBin x y "-"
  ...
printBin x y op
  = P (tell ["("] ≫| runPrint x ≫| tell [op] ≫| runPrint y ≫| tell [")"])
```

For printing the definitions, we use the same idea as in the previous section. A fresh identifier is used for all applied occurrences of the defined DSL object.[3] However, we prefer to use an instance of the more general definitions introduced in Section 3 and 4 below. Reuse is more concise and shows that this is indeed a special case of the more general definitions.[4]

```
instance cons Print where cons f = def  f
instance fun1 Print where fun1 f = funa f
```

Evaluating `printMain (ex1_v2 4)` produces:

```
def v0 = 1 In
def v1 = λv2 → (If (v2==.v0)
    v0
    (v2*(v1 (v2-v0)))) In
(v1 4)
```

Although we use a single `cons` and `fun1` definition in this example, it is important to realize that these constructs can be nested arbitrarily. Each of these definitions is a valid value of type `v a`.

2.3 Why no Fixed-Point Combinator

Our DSL primitives for local- and function definitions abstract the definition itself from its body and its applications. This resembles the common way to define recursive functions in the λ-calculus using the fixed-point combinator [3]. It is possible to implement recursion in our DSLs with such a fixed-point combinator. However, the direct approach used here has a number of advantages. First, we need only a single abstraction of the function name instead of a separate abstraction from the function body and another abstraction from the application expression. Second, the cyclic implementation of the evaluation view is more direct and efficient than the approach based on an explicit fixed-point combinator. Third, the generalization to more flexible DSL definitions as shown below is easier in our direct approach, e.g. each type of mutual recursion requires a separate fixed-point combinator. Finally, in our experience the approach based on fixed-point combinators is slightly more error prone both for the DSL user and the DSL implementer.

[3] Unfortunately, the names used by the user of the DSL are not accessible without template metaprogramming magic, see also Section 5.

[4] The only consequence is that the printed name of all definitions becomes the name of the most general case. We do not consider that a problem. Whenever required, we can make this name an additional parameter of a helper function that is called in the actual print class.

3 Controlled Function Arguments

The `cons` definitions in the DSL above cannot have an argument. The functions in `fun1` have exactly one argument, which is a single value in the DSL. By replacing this argument of type `v a` by the type variable `a`, making it a multi-parameter type class [7], we generalize this.

```
class funa a v :: ((a→v b)→In (a→(v b)) (v c)) → v c
```

With this definition we can allow more argument types. We can for instance make instances of this class for various tuples to allow multiple arguments. By making an instance for unit, `()`, we allow functions without arguments in C style. Note that this does not imply that tuples become part of the types in the DSL. The tuples in the host language are just a way to denote multiple arguments in the DSL. This is very useful if you want to limit the number of arguments, for example in low memory environments.

The Ackermann function is a famous example of a function with two arguments. It is defined as `ex2`.

```
ex2 =
  cons λzero = lit 0 In
  cons λone  = lit 1 In
  funa λack = (λ(m,n)→If (m ==. zero) (n +. one)
                      (If (n ==. zero) (ack (m -. one, one))
                          (ack (m -. one, ack (m, n -. one))))) In
  ack (lit 2, lit 2)
```

3.1 Evaluation

The evaluation instances is identical to the definitions shown above. We implement it in terms of the most general combinator (`def`, Section 4) as it is just a specialisation. Executing `eval ex2` produces the desired value 7.

```
instance funa a E where funa f = def f
```

3.2 Printing

The printing of this language construct is again more work. The instance for `(Print a) Print` is equal to the instance of `fun1` for `Print`, only the name `fun1` should be changed to `funa`. In the same style, we can make an instance for `() Print` to allow functions with zero arguments, or a unit argument to be more precise. We show the instance with a tuple of two arguments, as used in the example `ex2` above. The difference with the single argument function is that we generate two symbolic arguments for the body and handle a tuple of arguments in the applications.

```
instance funa (Print a,Print b) Print where
  funa f = P (fresh >>= λv → fresh >>= λa → fresh >>= λb →
    let (body In main) = f (λ(c,d).P (tell ["(",v," ("] >>| runPrint c >>|
```

```
      tell [", "] ≫| runPrint d ≫| tell [")"])) in
    tell ["funa ",v," = \\(",a,",",b,") → "] ≫|
    runPrint (body (P (tell [a]),P (tell [b]))) ≫|
    tell [" In\n"] ≫| runPrint main)
```

This prints our example `ex2` as:

```
def v0 = 0 In
def v1 = 1 In
def v2 = λ(v3,v4) → (If (v3==.v0)
    (v4+v1)
    (If (v4==.v0)
      (v2 ((v3-v1), v1))
      (v2 ((v3-v1), (v2 (v3, (v4-v1)))))))) In
(v2 (2, 2))
```

4 Arbitrary Number of Arguments

By changing the class for definitions a little more, we can handle literals as well as functions with an arbitrary number of arguments in the style of the host language. The function can be used uncurried, i.e. without the need to pack the arguments in a tuple. We just replace `a→v b` in the definition by `a` and make again appropriate instances. The class `def` defines these more general definitions in the DSL.

```
class def a v :: (a → In a (v b)) → v b
```

We illustrate the power of this approach by the algorithm to compute powers in logarithmic time.

```
ex3 = def λone = lit 1 In
      def λtwo = lit 2 In
      def λodd = (\n.If (n ==. one) (lit True)
                    (If (n <. one) (lit False) (odd (n -. two)))) In
      def λpow = (λx n.If (n ==. lit 0)
                        one
                        (If (odd n)
                           (x *. pow x (n -. one))
                           (def λy.pow x (n /. two) In y *. y))) In
      pow (lit 3) (lit 5)
```

The definitions `one` and `two` have no arguments, `odd` has a single argument, and `pow` has two arguments. The functions with arguments are recursive.

The definition of the value `y` is local inside the else branch of a conditional in the function `pow`. It is convenient that such arbitrary nested definitions are possible.

4.1 Evaluation

The evaluation can again be general for any definition of an arbitrary type `a`.

```
instance def a E where def f = let (body In exp) = f body in exp
```

Defining such an instance of this class can allow more than we want. It works for any type `a`. By defining more specific instances for various instance of `a` instead of this very general instance, we can control the allowed arguments in detail. In the mTask DSL for microcontrollers, we use this to limit the amount of function arguments, and forbid currying in such a way that we can use fixed size heap nodes [20]. For instance, by making only the instances `def (E a) E` and `def (E a, E b) E` we allow only functions with one and two arguments. Allowing more function arguments in definitions is just a matter of making instances for longer tuples.

Evaluating expression `eval ex3` yields the desired value `243`.

4.2 Printing

Printing of the `lit` and `arith` classes is already defined in Section 2.2. That implementation is also used with the more general definitions. Here we define the `Print` instance of our most general definition class `def`.

The class `def` works for definitions with an arbitrary number of arguments. In the applied function occurrences, we print a generated name for the functions as well as the actual arguments. To print the body of the expression, we supply functions that print variable names as arguments to the defined function. The helper class `defType` does exactly that for various types of arguments. The Boolean argument of `actArg` indicated whether a closing parenthesis is needed.

```
class defType a where
  actArg  :: Bool (Print c) → a
  formArg :: a → Print c
```

Printing definitions becomes straightforward with this tooling. First, we generate a fresh name `n` for the definition. Next, the defining function `f` is applied to the function that prints the actual arguments. Initially, this prints just the name of the function by `P (tell [n]))`. The `actArg` adds the actual arguments one by one. To print the function body, we supply the formal arguments one by one with `formArg`. Finally, we only have to add some bookkeeping code to announce that there is definition and print the definition to finally print the body.

```
instance def a Print | defType a where
  def f = P (fresh >>= \n.
    let (a In b) = f (actArg False (P (tell [n]))) in
    incr >>| tell ["def ",n," = "] >>| runPrint (formArg a) >>|
    tell [" In"] >>| decr >>| nl >>| runPrint b)
```

The basic case for the class `defType` covers the case that there are no more arguments. In this situation, the object to be printed has type `Print a`. For the actual argument `actArg`, we just run the given printer `p` and add a closing parenthesis whenever the Boolean argument `b` indicates that this is required. For the formal argument `formArg`, we just run the given printer.[5]

[5] In both cases the body is `P (runPrint p)` instead of just `p` to make the required type transition from `Print a` to `Print c`.

```
instance defType (Print a) where
  actArg b p | b         = P (runPrint p >>| tell [")"])
             | otherwise = P (runPrint p)
  formArg p = P (runPrint p)
```

The instance of `defType` for `(Print a)→b` handles the case for a single function argument. Recursive calls handle multiple arguments one by one. The instance for an actual argument takes that argument as argument and prints it after the accumulator `f`. The function `paren` prints an open parenthesis whenever needed and calls `actArg` recursively. For the formal argument, a fresh argument variable `v` is generated. The function yields a printer that produces the corresponding lambda definition and provides the printer `P (tell[v])` as argument to the function.

```
instance defType ((Print a)→b) | defType b where
 actArg b f = λx.paren b (runPrint f >>| tell [" "] >>| runPrint x)
 formArg  f = P (fresh >>= λv → tell ["\\",v," → "] >>|
   runPrint (formArg (f (P (tell [v])))))

paren :: Bool (PrintM ()) → a | defType a
paren b f = actArg True (P (if b f (tell ["("] >>| f)))
```

This prints our power function example `ex3` as:

```
def v0 = 1 In
def v1 = 2 In
def v2 = λv3 → (If (v3==.v0)
    True
    (If (v3<.v0)
      False
      (v2 (v3-v1)))) In
def v4 = λv5 → λv6 → (If (v6==.0)
    v0
    (If (v2 v6)
      (v5*(v4 v5 (v6-v0)))
      def v7 = (v4 v5 (v6/v1)) In
      (v7*v7))) In
(v4 3 5)
```

4.3 Indicating Types

The host language compiler is able to derive types for almost all DSL expressions though there are some exceptions. For these exceptions or software engineering reasons, it is convenient to specify types inside our DSL [27, §11.4]. Our host language Clean has no syntax for adding type ascriptions to expressions.[6] With the definition of type witnesses and two combinators we can mimic the effect for the monomorphic definitions in our DSL. The type witnesses are values inside our DSL with suggestive names. In these definitions, we hide on purpose that these are types in our DSL rather than plain values.

```
Bool :: v Bool | lit v
Bool = lit False
```

[6] Something that is available in e.g. Haskell98 [24, §3.16].

```
Int :: v Int | lit v
Int = lit 42
```

We define two combinators. The first infix combinator is used to indicate that an expression `e` has type `T` as `e ::: T`. The second infix operator constructs function types as `(λx. x +. lit 1) ::: Int →. Int`.

```
(:::) infix 1 :: a a → a
(:::) a t = a

(→.) infixr 2 :: a b → a→b
(→.) a b = λa → b
```

This approach is very similar to the use of `Proxy` and native type annotations. The advantage of our approach with type witnesses is that we never need to write `Proxy` in the types and are not building on language extensions. The disadvantage is that it must be possible to create a value of the type, something that is always the case, e.g. a value of a `World` [2] or `Void` type. Though `undef` can be used as the value is never evaluated anyway.

In this approach these witnesses are never materialized in any view of the DSL. The definition of `:::` discards them. An application is the factorial definition below. Definition `evenOdd` in Section 4.4 contains a bigger illustration. The examples show that we can type definitions with various arities as well as sub expressions.

```
ex1_typed n = def λone → lit 1 ::: Int In
              def λequ → (λx y → x ==. y) ::: Int →. Int →. Bool In
              def λfac → (\n. If (equ n one ::: Bool)
                      one
                      (n *. fac (n -. one) ::: Int)) ::: Int →. Int In
              fac (lit n) ::: Int
```

4.4 Mutual Recursion

By placing the function body inside the scope of the function name, we facilitate recursive functions in the DSL. The function name can be used inside its body as well as inside the expression after the `In` constructor. This is not good enough for mutual recursion since the ordering dictates that the first function is in scope of the second function, but not the other way around. We solve this problem by defining the functions at the same time in a tuple. A mutually recursive example is the definition of the functions `even` and `odd` in terms of each other.

```
evenOdd = def λone = lit 1 ::: Int In
          def λisZero = (\n.n ==. lit 0) ::: Int →. Bool In
          def λisOne = (==.) one ::: Int →. Bool In // Currying!
          def λ(odd,even) =
             (\n.If (isZero n) (lit False)
                    (If (isOne n) (lit True) (odd (n -. one) ::: Bool))
             ,\n.If (isZero n) (lit True)
                    (If (isOne n) (lit False) (even (n -. one)))
             ) ::: (Int →. Bool, Int →. Bool)
          In odd (lit 7) ::: Bool
```

To add tuples of definitions to our DSL we need an instance of `def` for evaluation for this.[7]

```
instance def (a,b) E where
  def f = let (pair In exp) = f (fst pair, snd pair) in exp
```

To print such mutual recursive definitions, a tuple which prints the names is given as argument to the defining function. The obtained bodies and the main expression are printed in order. We just have to add some text to distinguish the parts. In the same way we can add instances to define three or more mutually recursive functions simultaneously.

```
instance def (a,b) Print | defType a & defType b where
  def f = P (fresh >>= \n. fresh >>= λm.
    let ((e1a, e1b) In e2) =
      f (actArg False (P (tell [n])), actArg False (P (tell [m]))) in
    incr >>| tell ["def (",n,",",m,") ="] >>| incr >>| nl >>|
    tell ["("] >>| runPrint (formArg e1a) >>| nl >>|
    tell [","] >>| runPrint (formArg e1b) >>| tell [") In "] >>|
    decr >>| decr >>| nl >>| incr >>| runPrint e2 >>| decr)
```

4.5 Named Arguments

Sometimes it is convenient to indicate function arguments by a name rather than by their position. For instance when a function has multiple arguments of the same type. We can facilitate named arguments in the DSL by using a record type of the host language.

For instance, we calculate the compound interest of some loan based on four real numbers[8] The record `Loan` introduces names for these values. All values are `Real` numbers in our DSL. Hence, they are views `v` on such a value. This view `v` parameterizes the record.

```
:: Loan v = { sum  :: v Real // principal sum
            , rate :: v Real // nominal annual interest rate
            , freq :: v Real // compound frequency
            , time :: v Real // overall length of time
            }
```

Using this record we can use named function arguments and parameters as in the example `compoundInterest`. Note that the order of fields in the definition of the function `ci` differs from the argument `loan1`. Allowing this is exactly the purpose of the named arguments.

```
compoundInterest =
  def λci = (λ{sum, rate, freq, time} →
            sum *. (lit 1.0 +. rate /. freq) ^. (freq *. time)) In
  ci { freq = lit 4.0, time = lit 6.0
     , sum = lit 1500.0, rate = lit 4.3 /. lit 100.0}
```

[7] The standard implementation fails since it evaluates the tuple too soon, due to pattern matching being strict in the constructor.

[8] See https://en.wikipedia.org/wiki/Compound_interest for an explanation.

Without any changes to the given DSL implementation, this evaluates to `1938.8`.

We need to add an instance of the class `defType` from Section 4.2 when we want to print DSL expressions with instances of this type. We need to add argument identifiers to record fields to cope with multiple arguments of type `Loan`. Hence, it is more concise to print only this identifier as the formal argument of DSL definitions. Since records and record updates are part of the host language rather than the DSL, the only safe option is to print each record field explicitly in every application. This can become rather verbose. A direct implementation yields for our compound interest example:[9]

```
def v0 = λv1 → (v1.sum*((1+(v1.rate/v1.freq))^(v1.freq*v1.time)))
In (v0 {sum=1500,rate=(4.3/100),freq=4,time=6})
```

4.6 Only Top-Level Functions

The definitions introduced above can be nested arbitrarily. It is for the programmer convenient that arguments of a function are available in a nested definition. See for instance example `ex3` at the start of Section 4. The arguments `x` and `n` of function `pow` are used in the body of `y`. For some DSLs this is undesirable. Nested definitions require special attention in code generation to make the implicit arguments reachable. Well-known solutions are closure conversion and lambda lifting.

Using a slightly different function definition class, we can prevent nested definitions. The key is to replace the expression where the definition is applied by a more specific datatype. Here we use a record, but an algebraic datatype also works. By embedding the main expression in a record, we ensure that the type system enforces that the DSL user only writes definitions at the top level. To reuse as much as possible from the existing definitions, we use two arguments for the record `Main`.

```
:: Main v b = {main :: v b}
```

To avoid confusion, the class to make DSL definitions is called `defM`. Enforcing only top-level functions works of course only in a DSL where the ordinary class `def` is not available. We can still have multiple definitions, but only definitions at the outermost level. Any `defM` definition at a nested position causes the required type error. For brevity we do not implement `defM` separately but use `def`.

```
defM :: (a → In a (Main v b)) → Main v b | def a (Main v)
defM f = def f

instance def a (Main v) | def a v where
  def f = {main = def ((λ(a In {main = b}) = a In b) o f)}
```

Yet another variant of the factorial example is defined as `facM`.

```
facM n = defM λone = lit 1 ::: Int In
         defM λis0 = (==.) (lit 0) ::: Int →. Bool In
         defM λfac = (\n. If (is0 n) one (n *. fac (n -. one))) In
         {main = fac (lit n) ::: Int}
```

[9] We manually moved `In` to the next line to comply with the maximum line length.

5 Reuse of Embedded Functions

The embedded function definitions introduced above work fine. By choosing the appropriate variant, we determine what we want to allow in the DSL. Whenever desired, we extend the DSL with new operators or datatypes as well as functions with a different number or type of arguments.

There are two drawbacks for this approach. First, it requires that the entire DSL program is defined as a single block of code. This is fine for small programs, but for large programs this can hamper the readability. Second, it seems to obstruct the reuse of code. Finally, carefully designed function names are lost in showing the function definitions.

It is possible to make reusable functions in the current framework. We make a global function definition for each embedded definition we want to reuse. We start by adding an argument for each other definition used. Like the `one` in the function `fac` below. Next, we add an argument for the recursive calls of the function itself.[10] This is exactly equal to the use of a fixed-point combinator in λ-calculus (Section 2.3). Finally, we have the arguments of the embedded function, either as normal function arguments or as a lambda function.

The example `ex4` shows how this looks for a factorial definition.

```
one _ = lit 1
fac one f n = If (n ==. lit 0) one (n *. f (n -. one))

ex4 = def λoneF = one oneF In
      def λf = fac oneF f In f (lit 4)
```

This evaluates and prints like if we would have had the definitions inlined as above. This approach enables the reuse of the functions `one` and `fac`. An alternative approach is to define the functions as HOAS. This method and the method before result in very complex types become more complex and all library functions still need to be declared explicitely.

```
withOne f = def λone→lit 1 In f one
withFac f = withOne λone→
    def λfac→(\n.If (n ==. lit 0) one (n *. fac (n -. one)))
    In f fac

ex5 = withFac λfac→fac (lit 4)
```

5.1 Named Functions

A more radical approach uses only named definitions. These definitions have a used-defined `ID` that must be unique. This `ID` solves the lost names' problem in printing and is a unique identifier to spot whether we have encountered this definition before. The equivalent of the definitions from Section 3 becomes the classes `fun` for function with an arbitrary argument and `def` for constant definitions.

[10] Technically, this is not required for non-recursive definitions like constants. For uniformity and simplicity, we add this argument always in our examples.

```
class fun a v :: ID (a→v b) → a→v b
class term  v :: ID (v b) → v b

:: ID :== String
```

To specify concise type class constraints, we gather all relevant type classes in `funDef`. The type parameters are the view `v` and the argument type `a` in function definitions.

```
class funDef v a | lit, arith, bool, comp, If v & term v & fun (v a) v
```

The example with the functions `even` and `odd` below shows that this allows mutual recursion without the need to define the functions simultaneously as in Section 4.4.

```
Zero :: (v Int) | lit, term v
Zero = term "zero" (lit 0) ::: Int

One :: (v Int) | lit, term v
One = term "one" (lit 1)

even :: ((v Int)→v Bool) | funDef v Int
even = fun "even" \n.If (n ==. Zero) (lit True)  (odd  (n -. One))

odd :: ((v Int)→v Bool) | funDef v Int
odd  = fun "odd " \n.If (n ==. Zero) (lit False) (even (n -. One))
```

5.2 Evaluation

The evaluation is again straightforward, we just replace each definition by the body. The `ID` is not needed in this view and stripped.

```
instance fun a E where fun  i a = a
instance term  E where term i a = a
```

The expression `eval (even (lit 5))` evaluates to `False`.

5.3 Printing

For the printing view, we have to work a little harder. Here, we need the full tooling introduced in Appendix B. The state contains a mapping from `ID` to output of type `[String]`. Each time we encounter a new definition, we check this mapping for occurrence of the ID. When we have seen the definition before, we can just use the `ID` to indicate a call to this definition. When the definition is not known, we print it like before and store the output of the writer monad at the position of the `ID` in the mapping. When we are done with printing, we can just collect all definitions from the mapping with `printAll`. Like in Section 3, we can make instances for functions with a single argument as well as for tuples containing multiple arguments.

For simplicity, we assume that functions are not nested. One can handle the printing of nested functions for instance like their code generation by lifting all functions to the top level (see Section 6). We reuse the class `defType` from Section 4.2 for the type specific printing details. We only show the instances for

definitions without argument, functions with a single argument and a tuple as argument.

```
instance term Print where
  term name f = P (printDef "term" name f >>| tell [name])
instance fun (Print a) Print where
  fun name f = actArg False (P (printDef "fun" name (formArg f) >>| tell [name]))
instance fun (Print a, Print b) Print where
  fun name f = actArg False (P (printDef "fun" name (formArg f) >>| tell [name]))
```

Both instances uses the same helper function to add a definition to the mapping when this is needed.

```
printDef :: String ID a → PrintM () | defType a
printDef kind name f = gets (λs→'M'.get name s.defs) >>= λmd→case md of
  ?Just _ = pure () // definition found
  ?None   = censor (λ_→[]) (listen runDefinition) >>= λ(_,def)→
            modify (λs→{s & defs = 'M'.put name def s.defs})
where runDefinition = modify (λs→{s & defs = 'M'.put name [] s.defs})
        >>| enter name // enter to context
        >>| tell [kind," ",name," = "] >>| runPrint (formArg f)
        >>| leave
```

To add some resilience to the system, we can add the compiler-generated name of the current function to the `ID`. This yields unique names when the user of the DSL accidentally reuses an identifier, but generates ugly function names. If available, a template metaprogramming system such as Template Haskell [30] can be used to generate unique identifiers for functions. Printing our example `printAll (even (lit 5))` produces:

```
fun even = λv0 → (If (v0==.zero) True  (odd (v0-one)))
fun odd  = λv1 → (If (v1==.zero) False (even (v1-one)))
term one = 1
term zero = 0
main = (even 5)
```

It is no silver bullet because DSL functions can be constructed on the fly using the host language as a macro language [16]. For example, the `times` function below unrolls a multiplication function in a sequence of additions by using the host language. Using just the name, location or generated identifier per function in the host language is not enough. This can overcome by incorporating the arguments in the identifier, as done below.

```
times x = def ("times" +++ toString x) λy→foldr (+.) (lit 0) (repeatn x y)
```

6 Code for Nested Functions

To show that this approach is not limited to simpler views, we introduce code generation as a new view for the example DSL. To handle nested functions, we adjust the function definition and all applications with the lifted arguments. This obtains the same effect as lambda lifting [11].

The generated code is represented by the algebraic data type `Instruction` containing instructions for a stack machine. There are two instructions that contain data only used during compilation. Firstly, `Marker` is used to place markers in the code, and allow easy later patching the code. Secondly, `Arg` contains two data fields; the scope, and the argument number. The scope number is only used internally to implement lambda lifting. Built-in extensible ADTs [28, §5.1.5], *à la carte* style ADTs [31], or classy deep embedding [17] can be used to hide these constructors.

```
:: Instruction = Push Int | Arg Int Int
  | Add | Sub | Mul | Div | Le | Eq | And | Or | Not | Neg
  | Lbl Int | Jmp Int | JmpF Int | Jsr Int | Ret Int | Halt
  | Marker Int
```

6.1 Compilation

The compilation of our DSL follows the same schema as printing and uses a reader writer state monad. The reader part is not used. The state contains a counter for fresh identifiers, a map from function identifiers to the code of the body, a map from function names to identifiers and a map containing the required metadata used when calling a function. During execution of the monad, instructions are emitted through the writer part of the monad.

```
:: Compile a = C (CompileM ())
:: CompileM a :== RWS () [Instruction] CompileState a
:: CompileState =
  { fresh     :: Int
  , functions :: Map Int [Instruction] // Maps labels to instructions
  , funmap    :: Map String Int        // Maps names to labels
  , funcalls  :: Map Int [Instruction] // Maps labels to function calls
  }

runCompile :: (Compile a) → CompileM ()
runCompile (C a) = a
```

The instances of the DSL components regarding expressions follow the pattern familiar from printing them. For simplicity, we assume that there all basic data types can be converted to an integer to keep the stack representation homogeneous. This is added as a class constraint to the `lit` function. Only in the conditional expression, the state is used to generate fresh labels in order to implement the jumps between conditional branches.

```
instance lit Compile where
  lit a = C (tell [Push (toInt a)])
instance arith Compile where
  (+.) x y = C (runCompile x >>| runCompile y >>| tell [Add])
  ...
instance bool Compile where ...
instance comp Compile where ...
instance If Compile where
  If c t e = C (
    fresh >>= λelselabel→fresh >>= λendiflabel→
```

```
    runCompile c ≫| tell [JmpF elselabel] ≫|
    runCompile t ≫| tell [Jmp endiflabel, Lbl elselabel] ≫|
    runCompile e ≫| tell [Lbl endiflabel])
```

6.2 Functions

Functions are implemented in this compiler similar to the printer. We show the `fun` instance for single argument functions, the implementation for other arities is more of the same. In the body, the arity and the name of the function are passed to `compileOrRetrieveFunction`. The third argument of this function is a function that, given a label, provides a representation of the arguments to the function definition. The `compileOrRetrieveFunction` produces a label used for calling the function. If the function was already encountered, only the label is returned, if the function is seen for the first time, the definition is generated. Using the label, `callFunction` is called with the code to evaluate the arguments and the label.

```
instance term Compile where
  term name f = fun name (λ()→f) ()
instance fun (Compile a) Compile where
  fun name f =
    λx→C (compileOrRetrieveFunction 1 name (λlbl→f (C (tell [Arg lbl 0]))))
        ≫= callFunction (runCompile x))
```

Generating the Definition The `compileOrRetrieveFunction` function first checks if the function has been encountered before by looking it up in the `funMap`. If this is the case, the label is immediately be returned. Otherwise, a fresh label is generated and stored with the name in the `funMap`. Then the body is executed using the provided function `f`, but the output is captured using `censor` and `listen` similarly to what was done in the printing view. The instructions are placed after performing the lambda lifting. Finally, the context, what to execute before calling the function and evaluating the arguments, is stored in the `funcalls` field. Later, the `Marker` referencing this label is replaced by this sequence of instructions.

```
compileOrRetrieveFunction :: Int String (Int → Compile a) → CompileM Int
compileOrRetrieveFunction arity n f =
  gets (λs→'M'.get n s.funmap) ≫= λv→case v of
    ?Just i = pure i
    ?None = fresh // Generate a fresh label
      ≫= λlbl→modify (λs→{s & funmap='M'.put n lbl s.funmap})
      ≫| censor (λ_→[]) (listen (runCompile (f lbl)))
      ≫= λ(_, def)→let la = findLiftedArguments lbl def in
          modify (λs→{s & functions=
            'M'.put lbl (lift arity lbl def la) s.functions})
      ≫| modify (λs→{s & funcalls =
            'M'.put lbl [Arg lbl i \\ (_, i)←la] s.funcalls})
      ≫| pure lbl
```

Lambda lifting finds the captured arguments of nested functions and adds them to the definition and applications. In general, lifting all functions to top-level definitions requires an intensional analysis of the callgraph [22]. For the

sake of simplicity and brevity, this implementation only handles simple cases, one level deep, where an argument of an outer function is used in an inner function. Lifted arguments are identified by having a different label than the current function and assigned a number. Lifting them is done by replacing the `Arg` instructions by `Arg` instructions with the label of the current function and with a patched number.

```
findLiftedArguments :: Int [Instruction] → [((Int, Int), Int)]
findLiftedArguments lbl def = zip2 (sort (removeDup
  [(f, i) \\ (Arg f i)←def | f ≠ lbl])) [0..]

lift :: Int Int [Instruction] [((Int, Int), Int)] → [Instruction]
lift arity lbl def la = map replaceLift def ++ [Ret (arity + length la)]
where replaceLift :: Instruction → Instruction
      replaceLift a=:(Arg f i) = case lookup (f, i) la of
        ?None   = a
        ?Just a = Arg lbl (arity + a)
      replaceLift i = i
```

Calling Functions The second part of the `fun` implementation is generating the code for calling the function. In case of a recursive call, it is not yet known what data from the context, i.e. lifted arguments, need to be inserted as well. Therefore, a marker is included that represents the context, this is later be replaced by the code for the context. After pushing the marker, the arguments are evaluated and a `Jsr` instruction is written.

```
callFunction :: (CompileM ()) Int → CompileM ()
callFunction args i = tell [Marker i] >>| args >>| tell [Jsr i]
```

6.3 Running the Compiler

Compilation is a matter of running the monad stack. This results in code for the main expression in the writer output and the code and metadata for the functions in the resulting state. The main expression decorated with a `Halt` instruction is concatenated with functions decorated with labels. The resulting code still contains markers, they are then replaced by the corresponding instructions to push the lifted arguments.

```
compile :: (Compile a) → [Instruction]
compile (C f) = foldr replaceMarkers [] (main ++ [Halt
  : flatten [[Lbl l:is] \\ (l, is)←'M'.toList st.functions]])
where
  (st, main) = execRWS f ()
    { fresh=0,           functions='M'.newMap
    , funmap='M'.newMap, funcalls='M'.newMap
    }
  replaceMarkers (Marker i) acc = 'M'.find i st.funcalls ++ acc
  replaceMarkers i acc = [i:acc]
```

Using the following nested definition, `llift (lit 40)` evaluates to 42.

```
llift = fun "plustwo" λx→
  let local = fun "fplus" λy→y +. x
  in local (lit 2)
```

Compiling this code produces the instructions below. Function label `0` is the `plustwo` function, label `1` is the `fplus` function. Before jumping to `fplus`, `Arg 1 0`, the context, is pushed, followed by the argument `Push 2`.

```
   Push 40
   Jsr 0
   Halt
0: Arg 1 0
   Push 2
   Jsr 1
   Ret 1
1: Arg 1 0
   Arg 1 1
   Add
   Ret 2
```

7 Related Work

Hudak [9] introduce the notion of embedded modular DSLs. Pfenning and Elliott [26] introduce, with higher order abstract syntax, a simply typed λ-calculus enriched with products and polymorphism to express syntax trees.g This is an extension of the second-order term language of Huet and Lang [10]. Extensions of HOAS exist, such as parametric HOAS or PHOAS [6].g For the methods described in this paper, basic HOAS suffices. We reuse these ideas to construct embedded DSL terms using type classes by recursive functions, as well as techniques to express sharing and recursion introduced by Oliveira and Löh [23].

The original HOAS based on ADTs suffers from problems with exotic or junk terms [29]. The type system of the host language prevents those problems for HOAS based on GADTs [1]. In our shallow embedding, the type system of the host language also eliminates those undesirable terms.

Intentional analysis, like program optimization and partial evaluation, is hard with HOAS and shallow embeddings because we cannot inspect functions. This is solved in HOAS [1] and class based shallow embeddings [5, 13] by using an intermediate data type. Gill [8] focusses on detecting observable sharing in a deeply embedded DSL by using the *stable* names of the untyped graph representation. McDonell et al. [21] expand on this by performing it on a typed syntax tree.

Carette et al. [5] and Kiselyov [13] use λ-calculus with a fixed-point combinator instead of direct recursion. In this paper, we show that we can use also a fixed-point combinator, but that direct recursion is more elegant and direct.

Kiselyov [14] explicates the challenges in generating safe low-level code. These techniques should integrate with our approach.

Every class constraint needed in some view is part of the type classes constructing our DSL. Jones et al. [12] shows how these constraints can be moved to the views that need them.

Implementing named arguments by means of a record, as described in Section 4.5 has been discussed extensively in white and grey literature. To the best of our knowledge, it has never been used in a DSL context before.

8 Conclusion

This paper shows that it is possible to define strongly-typed functions as part of an embedded DSL in a strongly-typed host language. Key to these checked definitions are identifiers for functions in HOAS style, the DSL functions are (lambda) functions in the host language. Like any other function in the host language, such a lambda function is type checked. By checking the host language function, the compiler checks also the DSL function. The only syntactical overhead is the lambda for the defining identifier and a function serving as keyword before a definition.

To allow multiple views of the DSL we use a class-based embedding. Each view of the DSL is an instance of the classes constructing the DSL. As views we demonstrate evaluation, pretty printing and code generation of nested functions using lambda lifting for a first-order functional DSL.

In this paper we use various classes to define functions in the DSL. In the first class, functions have exactly one argument that is a single value in the DSL. In the next class, functions have a more general argument. This argument can for instance be a tuple containing zero or more normal arguments to control the allowed number of arguments in the DSL. In the most general case, DSL function have an arbitrary number of arguments, just like functions in the host language. By implementing instances of the classes, we control the type of arguments allowed in the DSL. By defining the function body and the expression where this is applied together, we allow recursion without needing a fixed-point combinator in the evaluation.

By defining slightly different classes, we ensure that the user defines only top-level functions. This makes code generation easier and can for instance statically control the maximum size of heap nodes needed in an implementation.

We introduced various ways to facilitate reuse of function definitions in the DSL. This enables us to make top-level libraries inside the DSL.

This paper uses Clean as host language, but the ideas can be implemented in any modern functional programming language. All code is available Lubbers et al. [19].

As future work we like to combine the code generation of such a DSL with type-safe stack manipulations [15]. The automatic type-safe transformation of embedded DSL programs brings new challenges. We plan to investigate how this can be done. This approach is by no means restricted to the simple first-order functional DSL we use as example. We think it is fruitful to do more case studies for this by creating larger and more complex DSLs.

Bibliography

[1] Atkey, R., Lindley, S., Yallop, J.: Unembedding Domain-Specific Languages. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, pp. 37–48, Haskell '09, ACM, New York, NY, USA (2009), ISBN 978-1-60558-508-6, https://doi.org/10.1145/1596638.1596644, event-place: Edinburgh, Scotland

[2] Backus, J., Williams, J.H., Wimmers, E.L.: An Introduction to the Programming Language FL. In: Research Topics in Functional Programming, pp. 219–247, Addison-Wesley Longman Publishing Co., Inc., USA (1990), ISBN 0-201-17236-4

[3] Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. No. 40 in Studies in Logic, College Publ, London (2012), ISBN 978-1-84890-066-0

[4] Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean — A language for functional graph rewriting. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, pp. 364–384, Springer Berlin Heidelberg, Berlin, Heidelberg (1987), ISBN 978-3-540-47879-9

[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 (sep 2009), ISSN 0956-7968, https://doi.org/10.1017/S0956796809007205

[6] Chlipala, A.: Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 143–156, ICFP '08, ACM, New York, NY, USA (2008), ISBN 978-1-59593-919-7, https://doi.org/10.1145/1411204.1411226, event-place: Victoria, BC, Canada

[7] DUGGAN, D., OPHEL, J.: Type-checking multi-parameter type classes. Journal of Functional Programming **12**(2), 133–158 (2002), https://doi.org/10.1017/S0956796801004233

[8] Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, pp. 117–128, Haskell '09, Association for Computing Machinery, New York, NY, USA (2009), ISBN 978-1-60558-508-6, https://doi.org/10.1145/1596638.1596653, event-place: Edinburgh, Scotland

[9] Hudak, P.: Modular domain specific languages and tools. In: Proceedings of the 5th International Conference on Software Reuse, p. 134, ICSR '98, IEEE Computer Society, USA (1998), ISBN 0818683775, https://doi.org/10.1109/ICSR.1998.685738, URL http://ieeexplore.ieee.org/document/685738/

[10] Huet, G.P., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Informatica **11**, 31–55 (1978), https://doi.org/10.1007/BF00264598

[11] Johnsson, T.: Efficient compilation of lazy evaluation. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, p. 58–69, SIGPLAN

'84, Association for Computing Machinery, New York, NY, USA (1984), ISBN 0897911393, https://doi.org/10.1145/502874.502880

[12] Jones, W., Field, T., Allwood, T.: Deconstraining DSLs. ACM SIGPLAN Notices **47**(9), 299–310 (Oct 2012), ISSN 0362-1340, 1558-1160, https://doi.org/10.1145/2398856.2364571

[13] Kiselyov, O.: Typed Tagless Final Interpreters. In: Gibbons, J. (ed.) Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures, pp. 130–174, Springer Berlin Heidelberg, Berlin, Heidelberg (2012), ISBN 978-3-642-32202-0, https://doi.org/10.1007/978-3-642-32202-0_3

[14] Kiselyov, O.: Generating C: Heterogeneous metaprogramming system description. Science of Computer Programming **231**, 103015 (2024), ISSN 0167-6423, https://doi.org/https://doi.org/10.1016/j.scico.2023.103015, URL https://www.sciencedirect.com/science/article/pii/S0167642323000977

[15] Koopman, P., Lubbers, M.: Strongly-Typed Multi-View Stack-Based Computations. In: Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming, pp. 1–12, PPDP '23, Association for Computing Machinery, New York, NY, USA (2023), ISBN 9798400708121, https://doi.org/10.1145/3610612.3610623, event-place: Lisboa, Portugal

[16] Krishnamurthi, S.: Linguistic reuse. PhD Thesis, Rice University, Houston, USA (2001)

[17] Lubbers, M.: Deep Embedding with Class. In: Swierstra, W., Wu, N. (eds.) Trends in Functional Programming, pp. 39–58, Springer International Publishing, Cham (2022), ISBN 978-3-031-21314-4, https://doi.org/10.1007/978-3-031-21314-4_3

[18] Lubbers, M., Achten, P.: Clean for Haskell Programmers (2024), URL https://arxiv.org/abs/2411.00037, _eprint: 2411.00037

[19] Lubbers, M., Koopman, P., Janssen, N.: Source code for the paper "Embedding Functions" (Oct 2023), URL https://doi.org/10.5281/zenodo.10225278, dOI: 10.5281/zenodo.10225278

[20] Lubbers, M., Koopman, P., Plasmeijer, R.: Interpreting Task Oriented Programs on Tiny Computers. In: Stutterheim, J., Chin, W.N. (eds.) Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, p. 12, IFL '19, ACM, New York, NY, USA (2021), ISBN 978-1-4503-7562-7, https://doi.org/10.1145/3412932.3412936, event-place: Singapore, Singapore

[21] McDonell, T.L., Chakravarty, M.M., Keller, G., Lippmeier, B.: Optimising purely functional GPU programs. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, pp. 49–60, ICFP '13, Association for Computing Machinery, New York, NY, USA (2013), ISBN 978-1-4503-2326-0, https://doi.org/10.1145/2500365.2500595, event-place: Boston, Massachusetts, USA

[22] Morazán, M.T., Schultz, U.P.: Optimal Lambda Lifting in Quadratic Time. In: Implementation and Application of Functional Languages: 19th Inter-

national Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers, pp. 37–56, Springer-Verlag, Berlin, Heidelberg (2008), ISBN 978-3-540-85372-5

[23] Oliveira, B.C.d.S., Löh, A.: Abstract syntax graphs for domain specific languages. In: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, p. 87–96, PEPM '13, Association for Computing Machinery, New York, NY, USA (2013), ISBN 9781450318426, https://doi.org/10.1145/2426890.2426909

[24] Peyton Jones, S. (ed.): Haskell 98 language and libraries: the revised report. Cambridge University Press, Cambridge (2003), ISBN 0-521 826144

[25] Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, p. 50–61, ICFP '06, Association for Computing Machinery, New York, NY, USA (2006), ISBN 1595933093, https://doi.org/10.1145/1159803.1159811

[26] Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, p. 199–208, PLDI '88, Association for Computing Machinery, New York, NY, USA (1988), ISBN 0897912691, https://doi.org/10.1145/53990.54010

[27] Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002), ISBN 0-262-16209-1

[28] Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean Language Report version 3.1. Tech. rep., Institute for Computing and Information Sciences, Nijmegen (Dec 2021)

[29] Sheard, T.: Accomplishments and Research Challenges in Meta-programming. In: Taha, W. (ed.) Semantics, Applications, and Implementation of Program Generation, pp. 2–44, Springer Berlin Heidelberg, Berlin, Heidelberg (2001), ISBN 978-3-540-44806-8

[30] Sheard, T., Peyton Jones, S.: Template Meta-Programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, pp. 1–16, Haskell '02, ACM, New York, NY, USA (2002), ISBN 1-58113-605-6, https://doi.org/10.1145/581690.581691, event-place: Pittsburgh, Pennsylvania

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

A Evaluation Tooling

Throughout this paper we use evaluators and printing views of various eDSLs. We try to reuse the standard tooling, like instances for the monad classes, as much as useful. The required definitions for evaluation are given in this appendix. The next appendix contains the tooling for the print views.

For the evaluation of DSL variants, we use the simplest type possible that generalizes to all monads.[11] There is no need to carry a state around since

[11] In Clean, ! indicates strictness, so this is the strict identity functor.

all variables are represented by functions or function arguments. We use the substitution mechanism of the host language to ensure type-safe and efficient replacement of variables by the appropriate value.

```
eval :: (E a)→a
eval (E a) = a

:: E a = E !a

instance pure E where pure a = E a
instance Monad E where bind (E a) f = f a
instance Functor E where fmap f (E a) = E (f a)
instance <*> E where (<*>) (E f) (E a) = E (f a)
```

The evaluation of the basic classes not listed in Section 2.1 are listed here. The instances for the class `bool` ensure that the second argument is only evaluated when that is necessarily.

```
instance bool E where
  (&&.) x y = x >>= λb → if b y (pure False) // for lazy evaluation
  (||.) x y = x >>= λb → if b (pure True) y  // for lazy evaluation
instance comp E where
  (==.) x y = (==) <$> x <*> y
  (<.) x y = (<) <$> x <*> y
instance If E where If c t e = c >>= λb.if b t e
```

B Print Tooling

The print tooling is more sophisticated than the evaluation tooling. It is based on the reader writer state monad. The state `PS` is a record containing an integer `i` to generate fresh variables, a context that is a stack of function `ID`s, a mapping from `ID`s to output as a list of strings, and an indentation depth `ind`. Only the last DSL versions use all these fields. The writer monad is a list of strings, `[String]`, to denote the output of printing. The reader `PR` is not used and equals void, `()`.

```
:: PS = {i :: Int, context :: [ID], defs :: Map ID [String], ind :: Int}
:: PR :== ()
:: Print a = P (PrintM ())
:: PrintM a :== RWS PR [String] PS a
:: ID :== String
```

We use some convenience functions for this `RWS` monad. They are explained in context on their first use in this paper.

```
runPrint :: (Print a) → PrintM ()
runPrint (P a) = a

nl :: PrintM ()
nl = get >>= λs → tell ["\n":[" " \\ _ ← [1..s.ind]]]

incr :: PrintM ()
incr = modify λs → {s & ind = s.ind + 1}

decr :: PrintM ()
decr = modify λs → {s & ind = s.ind - 1}
```

```
fresh :: PrintM String
fresh = get >>= λs.put {s & i = s.i + 1} >>| pure ("v" + toString s.i)

printAll :: (Print a) → String
printAll (P f) = concat
  ('M'.foldrWithKey (λk v a.["\n":v] ++ a) ["\n":main] st.defs)
where
  (st, main) = execRWS f () {i=0, context=[], defs='M'.newMap, ind=0}

printMain :: (Print a) → String
printMain (P f) = concat main
where
  (st, main) = execRWS f () {i=0, context=[], defs='M'.newMap, ind = 0}

prnt :: a → PrintM () | toString a
prnt s = tell [toString s]

show‘ :: (DSL a) → PrintM () | toString a
show‘ e = let (P p) = show1 e in p
```

The print instances for basic classes not listed in Section 2.2 are listed here.

```
instance bool Print where
  (&&.) x y = printBin x y "&&"
  (||.) x y = printBin x y "||"
instance comp Print where
  (==.) x y = printBin x y "=="
  (<.) x y  = printBin x y "<"
instance If Print where
  If c t e = P (tell ["(If "] >>| runPrint c  >>| tell [" "] >>|
    runPrint t >>| tell [" "] >>| runPrint e >>| tell [")"])
```