Embedding Functions
– extended abstract –

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

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

Niek Janssen
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
Niek.Janssen@science.ru.nl

ABSTRACT

This paper discusses implementation techniques for functions in embedded Domain Specific Languages, DSLs. The challenge is threefold. Firstly, the type system of the host language should ensure that the functions are properly defined and applied. Secondly, the function definitions and application should flexible and require minimal syntactical overhead. Finally, we want to allow multiple interpretations like evaluation, pretty printing and code generation of the DSLs.

This paper offers a systematic overview of the options to define type-safe functions in an embedded DSL. 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.

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Niek Janssen. 2023. Embedding Functions – extended abstract –. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 some host language. This implies that the host language constructs can be reused in the eDSL. Moreover, it saves us from making a complete tool chain with parser, type checker and so on for the eDSL. Typically, the host language becomes a powerful generator of eDSl programs. Functional programming languages have shown to be very suited host languages for embedded DSLs. This has been recognized long ago by Hudak [Hudak 1998] and others. Recent additions increase their quality as host language for eDSLs. Some properties contributing are the ability to define new infix operators, generalized algebraic data types, multi-parameter type constructor classes and powerful function types in almost any combination.

In this paper, we focus on the definition of functions in eDSLs. Basically, functions can be handled like variables with a fancy type. In a naive implementation, such variables are just a variable tag, either a constructor or a function, and some identifier. The host language compiler is not able to check that the variable is properly defined nor that the type is used correctly. The compiler can check variables, and hence function, definitions when we represent them by function or function arguments in the host language. This is an old idea known as Higher Order Abstract Syntax, HOAS [Pfenning and Elliott 1988].

For embedded functions, we have some additional requirements over variables. Firstly, we want to have control over the type of function arguments. Even for polymorphic and overloaded functions, it should not be allowed to use types that are not considered to be part of the eDSL. For some applications it is required to have an upper limit on the number of arguments of the embedded function, e.g. when an evaluator of the eDSL needs to store thunks in some finite memory location. Secondly, embedded functions should be able to call each other and themselves. We require at least recursion, and preferably mutual recursion. Finally, function application in the DSL should resemble function application is the host language and not require an explicit apply construct. That is, we prefer to write `f x` over `ap f x` in our DSL.

For an eDSL with a single evaluation view this is usually not too complicated. Each applied occurrence of the embedded function is replaced by a fresh copy of its body. This body is evaluated like any other simple expression in the eDSL. For views like printing and code generation, the implementation is slightly more challenging. Since the quests for code generation are basically very similar to printing, we limit us to displaying the eDSL in some text format. The optimization of the eDSL requires the dynamic generation of eDSl constructs. Typing this correctly imposes some new constraints that are not covered in this paper.

Organization of the paper.

The main contributions of this paper is the supplied systematic overview of the options to define type-safe functions in an embedded DSL. 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.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DEEP EMBEDDING

In a deep embedding, we represent the DSL by some Algebraic Data Type, ADT. When we do not want to define a new data for each type in the DSL or want to allow overloading in the DSL we use an ADT with a type argument. This argument indicates the type represented by the language construct. The standard Hindley-Milner type system of functional languages like Haskell and Clean is not always able to derive these types. There are two well-known solutions for this problem; Generalized ADTs, GADTs, and adding bimaps [?]. In both solutions, the designer of the eDSL indicates the value of the type arguments explicitly. For instance, the bimap in `Eq` indicates that result type is `Eq Bool`. In GADT one would write a function type for this constructor: `Eq (DSL b) (DSL b) → DSL Bool | type b`.

```
:: BM a b = {ab::a→b, ba::b→a}

bm :: BM a a
bm = {ab = id, ba = id}
```

For embedded functions, the interesting constructors are `Let` to define local values and `Fun` to define single argument functions. To generate proper typing and binding of these constructs, we use Higher Order Abstract Syntax, HOAS [Pfenning and Elliott 1988]. In both cases we use a function to ensure that the defined value is used properly. To allow recursion, the body of the function has to be in the scope of the name definition. The body of the function basically is a tuple of the function body and the main expression. We use to single constructor datatype `In` instead of an ordinary tuple, since it limits the amount of parentheses and is syntactically clearer.

```
:: DSL a
  = Lit a
  | ∃ b: Eq (BM a Bool) (DSL b) (DSL b) & type b
  | Mul (BM a Int) (DSL Int) (DSL Int)
  | Sub (BM a Int) (DSL Int) (DSL Int)
  | Not (BM a Bool) (DSL Bool)
  | IF (DSL Bool) (DSL a) (DSL a)
  | ∃ b: Let (DSL b) ((DSL b)→DSL a) & type b
  | ∃ b.c: Fun (((DSL b)→(DSL c))→In ((DSL b)→(DSL c)) (DSL a))
    & type b & type c
  | Var String // for printing names

:: In a b = In infix 0 a b

class type a | toString, == a
```

The variable constructor `Var` should not be used by the user of the DSL. It is used internally in the print view of this DSL.

With a few additional function definitions, we can free the user of the DSL from writing explicit bimaps. Whenever desired, we can even define infix operators with the appropriate binding power instead of these functions.

```
eq = Eq bm
sub = Sub bm
mul = Mul bm
```

The definition of the familiar factorial function serves as an example.

```
e1 n =
  Let (Lit 1) λone →
  Fun λfac →
    (λx → IF (eq x (Lit 0)) one (mul x (fac (sub x one)))) In
  (fac (Lit n))
```

2.1 Evaluation

Evaluation is a straightforward descent of the syntax tree of the DSL. For the `Let` and `Fun` cases, we use the functions of the host language for uniform substitution; any occurrence of the function argument is replaced by the corresponding definition. The lazy evaluation of the host language ensures that we avoid unnecessary endless recursive computations.

```
eval :: (DSL a) → a
eval e = case e of
  Lit a = a
  Eq bm a b = bm.ba (eval a == eval b)
  Mul bm a b = bm.ba (eval a * eval b)
  Sub bm a b = bm.ba (eval a - eval b)
  Not bm a = bm.ba (not (eval a))
  IF c t e = eval (if (eval c) t e)
  Let e f = eval (f e)
  Fun f = let (b In e) = f b in eval e
  Var name = abort ("Var " + name + " should not occur in evaluation")
```

As announced by the introduction of the datatype, the `Var` constructor is only for internal use in the print view. We cannot prevent that it is used here, but any occurrence is erroneous.

The value of `eval (e1 4)` is the desired value 24.

2.2 Printing

The printing of this DSL is slightly more complex. We use the tooling of Appendix B that introduces a Reader Writer State monad, `RWS`. Here we use only the writer as a list of strings and the state to generate fresh identifiers.

For the basic operators `Lit`, `Eq`, `Mul`, `Sub`, `Not`, and `IF` we simply follow the tree structure of the ADT. For the definitions `Let` and `Fun` we replace all applied occurrences with a fresh variable. We need the strange constructor `Var` to make the required occurrences of these names part of the ADT. For both constructs, we print the value of the definition before printing the main expression. For the embedded functions, we have to generate a symbolic argument to print its definition. For its applied occurrences, we add the value of the actual argument to this generated variable. We show a `Var` by just adding the given string to the writer monad.

```
show e = case e of
  Lit a = P (prnt a)
  Eq  _ a b = P (tell ["(","Eq "]  ≫| show` a ≫|
                prnt " " ≫| show` b ≫| prnt ")")
  Mul _ a b = P (tell ["(","Mul "] ≫| show` a ≫|
                prnt " " ≫| show` b ≫| prnt ")")
  Sub _ a b = P (tell ["(","Sub "] ≫| show` a ≫|
                prnt " " ≫| show` b ≫| prnt ")")
  Not _ a   = P (tell ["(","Not "] ≫| show` a ≫| prnt ")")
  IF  c t e = P (tell ["(","IF "] ≫| show` c ≫| prnt " " ≫|
                show` t ≫| prnt " " ≫| show` e ≫| prnt ")")
  Let e f
  = P (fresh ≫= λv→nl ≫| incr ≫| tell ["(","Var ",v," = "] ≫|
       show` e ≫| prnt " In " ≫|
       decr ≫| show` (f (Var v)) ≫| prnt ")")
  Fun f
  = P (fresh ≫= λv → fresh ≫= λa → let (b In e) = f (λx →
      Var ("(" + v + " " + printMain (show x) + ")")) in
      nl ≫| incr ≫| tell ["(","Fun ",v," ",a," = "] ≫|
      show` (b (Var a)) ≫| prnt " In " ≫| decr ≫| show` e)
  Var n     = P (prnt n)
```

The expression `show (e1 4)` yields

```
(Var v0 = 1 In
(Fun v1 v2 = (IF (Eq v2 0) v0 (Mul v2 (v1 (Sub v2 v0)))) In (v1 4))
```

3 SHALLOW EMBEDDING

In the previous section, we used functions to represent local variables and functions in GADT based DSL. In the DSL jargon, this is called a deep embedding. One can argue that these components are more like a function based embedding than a true deep embedding. In the coming sections, we show various versions of such a shallow embedding. We use a class based embedding rather than a function based embedding to allow multiple interpretations of the DSL. This class-based embedding is known as finally tagless [Carette et al. 2009].

We start with the class for the ordinary operators that is reused by all versions of the DSL. To allow more variation in the examples, we have added some operators. We use infix operators with the usual binding power to beautify the syntax. Whenever appropriate, we add a dot to the operator name to avoid name clashes.

```
class lit v :: a → v a | toString, toInt 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 start with defining local values and single argument functions in the same style as in the previous section. The corresponding classes are

```
class Let  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
```

Note that we moved the body of `Let` definitions to the right-hand side of the function for consistency with the function definitions. Later we will unify those definitions.

The example from the previous section in this embedding becomes the new `e1`.

```
e1 =
  Let  λone → lit 1 In
  fun1 λfac → (\n. If (n ==. one) one (n *. fac (n -. one))) In
  fac (lit 5)
```

Note that `Let` and `In` are definitions in the host language that are part of the DSL. Do not confuse them with the syntax element **let** and **in** of the host language.

3.1 Evaluation

Evaluating these operators in the monad `E` is completely standard. Only for the and- and or-operator, we need some additional work to make them lazy in our DSL.

The evaluation of the `Let` and `fun1` classes is pretty simple. 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
  (*.) x y = (*) <$> x <*> y
  (/.) x y = (/) <$> x <*> y
instance bool E where
  (&&.) x y = x ≫= λb → if b y (pure False)
  (||.) x y = x ≫= λb → if b (pure True) y
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

instance Let  E where Let  f = let (val In body) = f val in body
instance fun1 E where fun1 f = let (val In body) = f val in body
```

The main function `Start = eval e1` will produce the value 120.

3.2 Printing

Also printing these operators in the writer monad is pretty standard.

```
instance lit Print where lit a = P (tell [toString a])
instance arith Print where
  (+.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["+"] ≫|
                runPrint y ≫| tell [")"])
  (-.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["-"] ≫|
                runPrint y ≫| tell [")"])
  (*.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["*"] ≫|
                runPrint y ≫| tell [")"])
  (/.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["/"] ≫|
                runPrint y ≫| tell [")"])
instance bool Print where
  (&&.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["&&."] ≫|
                 runPrint y ≫| tell [")"])
  (||.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["||."] ≫|
                 runPrint y ≫| tell [")"])
instance comp Print where
  (==.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["==."] ≫|
                 runPrint y ≫| tell [")"])
  (<.) x y = P (tell ["("] ≫| runPrint x ≫| tell ["<."] ≫|
                runPrint y ≫| tell [")"])
instance If Print where
  If c t e = P (tell ["(If "] ≫| runPrint c  ≫| tell [" "] ≫|
  runPrint t ≫| tell [" "] ≫| runPrint e ≫| 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. When this object is a function, the instance for `fun1`, we add a symbolic argument in printing the body and the actual argument when printing the applications.

```
instance Let Print where
  Let f =
    P (fresh ≫= λv → let (val In main) = f (P (prnt v)) in
      tell ["Let ",v," = "] ≫| runPrint val ≫| prnt " In\n" ≫|
      runPrint main)
instance fun1 Print where
  fun1 f =
    P (fresh ≫= λv → fresh ≫= λa → let (body In main) =
        f (λb.P (tell ["(",v," "] ≫| runPrint b ≫| prnt ")")) in
      tell ["fun1 ",v," = \\",a," → "] ≫|
      runPrint (body (P (prnt a))) ≫| prnt " In\n" ≫|
      runPrint main)
```

Evaluating `printMain e1` will produce

```
Let v0 = 1 In
fun1 v1 = λv2 → (If (v2==.v0) v0 (v2*(v1 (v2-v0)))) In
(v1 5)
```

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

In our mTask language, we do not want this to avoid the need of sophisticated lambda-lifting. By embedding the main expression in a record, we can ensure that the type system enforces that the DSL user only writes definitions at the top level.

4 FUNCTIONS WITH CONTROLLED ARGUMENTS

The `Let` 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 a type variable `a` we can generalize this to various types.

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

We can for instance make instances of this class for various tuples to allow multiple arguments. Note that this does not imply that tuple 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.

The Ackermann function is a famous example with two arguments. It can be defined now as shown by `e2`.

```
e2 =
  Let λzero = (lit 0) In
  Let 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)
```

4.1 Evaluation

The evaluation can again be identical to the definitions seen above.

```
instance funa a E where funa f = let (val In body) = f val in body
```

This can allow more that we want for the in other views. By defining more specific instances for various instance of `a` instead of this very general instance, we can control the allowed arguments in detail.

The `eval e2` produces the desired value 7.

4.2 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 void argument to be more precise. We show the instance with a tuple of two arguments, as used in the example `e2` above. The difference with the single argument function is that we have to 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 `e2` (with a manually added newline) as:

```
Let v0 = 0 In
Let v1 = 1 In
funa 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)
```

5 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. We just replace `a→v b` in the definition by `a` and make again appropriate instances.

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

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

```
e3 =
  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)
```

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

5.1 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
```

The value of `eval e3` is the desired value 243.

5.2 Printing

For printing, we allow definitions with an arbitrary number of plain arguments of type `Print a` and a result of type `Print b`. By construction this rules out the tuple arguments we introduced above as well as higher order functions. This is a result of our choices, not a limitation of the approach.

For the implementation, we introduce two additional classes. The class app takes care of the proper application of the definitions. It inserts parentheses around the function application and shows the actual arguments of the application. The class body takes care of printing the body of the definition. It creates a fresh variable for each function argument and passes the printing of that variable as argument to the body. This makes the printing of definitions simpler. We just make a fresh variable for definitions, print the fact the we encountered a fresh definition and use the classes body and app to handle printing of the body and the applications.

```
instance def a Print | app, body a where
  def f =
    P (fresh ≫= \n.
    let (e1 In e2) = f (app (P (tell ["(",n]))) in
    nl ≫| incr ≫| tell ["def ",n," = "] ≫| runPrint (body e1)
     ≫| tell [" In "] ≫| decr ≫| runPrint e2)

class app a :: (Print c) → a
instance app ((Print a) → b) | app b where
  app s = λx → app (P (runPrint s ≫| tell [" "] ≫| runPrint x))
instance app (Print a) where app s = P (runPrint s ≫| tell [")"])

class body a :: a → (Print c)
instance body (Print a) where body s = P (runPrint s) //to fix the type
instance body ((Print a) → b) | body b where
  body f = P (fresh ≫= λv → tell ["\\",v," → "] ≫|
              runPrint (body (f (P (tell [v])))))
```

Whenever we want to restrict the type of arguments to the types in our DSL we simply add the class constraint type a to the instances of the classes app and body.

This prints our example e3 pretty well. We have manually introduced line breaks in the bodies of v2 and v4 to cope with the limited line length in this paper.

```
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)
```

Note that we have used here the function nl from the tooling in Appendix B to introduce a newline and a proper indentation for nested definitions like v7. The function incr increments this indentation and derc decrements it. We can make this mechanism more sophisticated by need.

Admittedly, the parentheses around constant definitions are superfluous. It is straightforward to prevent those parentheses by introducing an additional class for the application of definitions. For a simple Print a it prints the name of the definition, for all other definitions it prints the opening parenthesis and continues as above.

6 REUSE OF EMBEDDED FUNCTIONS

The embedded function definitions introduced above work fine. By choosing the appropriate variant, we can determine what we want to allow in the DSL. Whenever desired, we can extend the DSL by new operators or datatypes as well as functions with a different numbed 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. Moreover, it seems to obstruct the reuse of code. Second, the carefully designed function names are lost in showing the function definitions.

It is possible to make reusable functions in the current framework. The algorithm is pretty simple. We can 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[1]. This is exactly equal to the use of a Y-combinator. Finally, we have the arguments of the embedded function, either as normal function arguments or as a lambda function.

Example e4 shows how this looks for a simple factorial example.

```
e4 =
  def λo = one o In
  def λf = fac o f In f (lit 4)

one o = lit 1
fac one f n = If (n ==. lit 0) one (n *. f (n -. one)) // like for a Y
```

This will evaluate and print like we had the definitions inlined as above.

A more radical approach uses only global 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 4 becomes the classes fun for function with an arbitrary argument and def for constant definitions.

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

:: ID :== String
```

The example below shows that this allows mutual recursive functions.

```
oneDef :: (v Int) | lit, def v
oneDef = def "one" (lit 1)

even =
  fun "even" \n.If (n ==. lit 0) (lit True)  (odd  (n -. oneDef))
odd  =
  fun "odd " \n.If (n ==. lit 0) (lit False) (even (n -. oneDef))
```

6.1 Evaluation

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

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

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

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

6.2 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, `[string]`. Each time we encounter a new definition, we check this mapping. 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 4, we can make instances for functions with a single argument as well as for tuples containing multiple arguments. We only show the instances for definitions without argument and functions with a single argument.

```
instance def Print where
  def name f =
    let n = name + thunk_name_to_string f in
    P (printDef ("def " + n) "" f ≫| tell [n])

instance fun (Print a) Print where
  fun name f =
    λx→let n = name + thunk_name_to_string f in
    P (fresh ≫= λv→printDef ("fun " + n) ("\\" + v + ".")
    (f (P (tell [v]))) ≫| tell ["(", n, " "] ≫| runPrint x ≫|
    tell [")"])
```

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

```
printDef :: ID String (Print a) → PrintM ()
printDef n v p =
  gets (λs→'M'.get n s.defs) ≫= λmd→case md of
    ?Just _ = pure () // definition found
    ?None   = censor (λ_→[]) (listen runDefinition)
              ≫= λ(_, def) →
                  modify (λs → {s & defs = 'M'.put n [v:def] s.defs})
where
  runDefinition
    = modify (λs→{s & defs = 'M'.put n [] s.defs}) // Add to state
      ≫| enter n      // enter to context
      ≫| runPrint p   // add definition to current context
      ≫| leave        // Pop from context
```

To add some redundancy to the system, we add the location of the current function to the `ID`. This makes the function `ID` unique when two different functions use the same `ID`. Printing our example `printAll (even (lit 5))` yields a correct result. We have omitted the added function locations for brevity.

```
def oneP = 1
fun even = λv0.(If (v0==.0) True  (odd  (v0-oneP)))
fun odd  = λv1.(If (v1==.0) False (even (v1-1)))
(even 5)
```

// fix the last 1 to one and oneP to one !!! // the current printTooling is wrong !!! // can be done after submission of the draft

7 CODE GENERATION WITH LAMBDA LIFTING

To show that we can also introduce more complex views and manipulations over our embedded functions, we introduce code generation for the definitions from the previous section. The generated code is represented by the algebraic data type `Instruction`.

```
:: 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
```

7.1 Compilation

The compilation of our DSL follows the same schema as printing. we have a very similar state.

```
:: Compile a = C (CompileM ())
:: 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
    }
:: CompileM a :== RWS () [Instruction] CompileState a
```

The instances of the DSL components follow the pattern familiar from printing them.

```
instance lit Compile where
  lit a = C (tell [Push (toInt a)])
instance arith Compile where
  (+.) x y = C (runCompile x ≫| runCompile y ≫| tell [Add])
  (-.) x y = C (runCompile x ≫| runCompile y ≫| tell [Sub])
  (*.) x y = C (runCompile x ≫| runCompile y ≫| tell [Mul])
  (/.) x y = C (runCompile x ≫| runCompile y ≫| tell [Div])
instance bool Compile where
  (&&.) x y = C (runCompile x ≫| runCompile y ≫| tell [And])
  (||.) x y = C (runCompile x ≫| runCompile y ≫| tell [Or])
instance comp Compile where
  (==.) x y = C (runCompile x ≫| runCompile y ≫| tell [Eq])
  (<.) x y = C (runCompile x ≫| runCompile y ≫| tell [Le])
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]
    )
instance def Compile where
  def name f = fun name (λ()→f) ()
instance fun (Compile a) Compile where
  fun name f =
   λx→C (compileOrRetrieveFunction name
           (λlbl→f (C (tell [Arg lbl 0])))
      ≫= callFunction (runCompile x))
```

7.2 Evaluation Compiled Code

Code works fine, but must be added to the paper.

7.3 Printing Compiled Code

Code works fine, but must be added to the paper.

8 RELATED WORK

To be done.

9 CONCLUSION

In this paper we have introduced various ways to add type-safe embedded functions to an embedded DSL. This can be done in a deep embedding with datatypes as well as in a shallow embedding based on classes. These classes are required to allow multiple views of the DSL. We have demonstrated the views evaluation, printing and code generation with lambda-lifting. All type-safe function are based on some kind of function in the host language to represent the embedded functions. We have demonstrated that there are several ways to define these embedded functions and that each of these has its own benefits.

REFERENCES

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (sep 2009), 509–543. https://doi.org/10.1017/S0956796809007205

P. Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98)*. IEEE Computer Society, USA, 134.

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

To be extended with related work.

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 Monadic 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. There is no need to carry a state around since 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)
```

B PRINT TOOLING

The print tooling is more sophisticated than the evaluation tooling. It is based on the Reader Writer State monadic instance. 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 equal to 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 will 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", k, " = ":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]
```