Shallowly Embedded Functions

Benedikt M. Rips
Radboud University
Nijmegen, Netherlands
benedikt.rips@ru.nl

Niek Janssen
Radboud University
Nijmegen, Netherlands
niek.janssen3@ru.nl

Mart Lubbers
Radboud University
Nijmegen, Netherlands
mart@cs.ru.nl

Pieter Koopman
Radboud University
Nijmegen, Netherlands
pieter@cs.ru.nl

Abstract

A domain-specific language, DSL is tailored to a specific 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, the automatic replacement of host function calls by their body is undesirable in other interpretations of the DSL, like pretty printing and code generation, especially for recursive DSL functions.

In this paper, we offer an overview of the options for defining functions in an embedded DSL, in particular tagless-final, or class-based shallow embedding style. These functions are type safe, require minimal syntactic overhead, and are suitable for multiple interpretations of terms in the DSL.

CCS Concepts

• **Software and its engineering → Domain specific languages**; **Automated static analysis**; **Functional languages**; **Recursion**.

ACM Reference Format:
Benedikt M. Rips, Niek Janssen, Mart Lubbers, and Pieter Koopman. 2025. Shallowly Embedded Functions. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming (PPDP '25), September 10–11, 2025, Rende, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3756907.3756923

1 Introduction

Domain-Specific Languages, DSLs, help to develop and maintain software for particular application domains. Embedded DSLs, so-called eDSLs, are implemented as libraries in host programming languages. Preferably, an eDSL supports multiple interpretations, for example printing, optimization, or evaluation. It is desirable that the type safety of the host language is also available in the eDSL. Embedding saves us from making a standalone tool chain with a parser, type checker, etc. for the DSL, as they are inherited from the host language. Typically, the host language becomes a powerful generator of eDSL programs [19]. Though some trade-offs have to be made for the correct typing and sharing of identifiers such as function names and when using them in function application.

Functional programming languages have shown to be well-suited host languages for eDSLs. This has been recognized long ago by Hudak [12] and its large number of successors. Moreover, recent extensions of functional programming languages increase their quality as host languages for eDSLs. For example, the ability to define infix operators, generalized algebraic data types, multi-parameter type (constructor) classes, and higher-order/rank polymorphic function and data types.

In this paper, we focus on the definition of functions in eDSLs. Basically, functions are handled as 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, if we represent them by functions or function arguments in the host language, the compiler can check variables. This is an old idea known as Higher-Order Abstract Syntax, HOAS [30]. We use this concept to define functions in DSLs and thus reducing syntactic overhead and improving type safety. We parameterize the type of DSL expressions to mirror the type of the equivalent host language expression. Hereby, the host language's type system not only imposes type safety onto DSL expressions but also gives us type inference.

1.1 Requirements for DSL Functions

We would like embedded functions to fulfil certain requirements. First, we want to be able to control the types of the bound expressions if necessary, even for polymorphic, overloaded, and higher-order functions. Also, we would like to have the possibility to set bounds on the number of arguments of the function. This comes in handy when e.g. a compiler 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. Second, function application in the DSL should naturally resemble function application in the host language. It preferably does not require an explicit application construct. That is, we prefer to write `f x` over `ap f x` in our eDSL to minimize the semantic friction and syntactic overhead without losing control over the definition and callsites. Third, expressions shall be sharable in a manner that is accessible to the interpretations such that the implementation of sharing can be specific to the interpretation at hand. Fourth, embedded functions should be able to call themselves and each other. We require at least recursion, but preferably mutual recursion. Finally, we prefer to have single abstraction for binding, sharing, *and* recursion to make DSL expressions concise and uniform.

1.2 Research Contribution

Our work regards DSL binders represented in HOAS. The overall contribution is fourfold.

First, we present a DSL binder that is capable of defining functions of arbitrary arity (Section 3.2). We do so by providing a binder that is polymorphic in the type of its bound expression while still allowing DSL-level functions only and suitable type class instances for the arrow type. For the sake of completeness, we put our ideas in the context of the well-known technique for DSL binders for functions of fixed arity.

Second, we discuss DSL binders for mutually-recursive expressions (Section 3.3). We start out with an illustration of a naive approach that does not require additional syntax but only enables exposure of one of the multiple mutually-recursive expressions. To remedy this disadvantage, we extend the existing binders to tuples such that multiple expressions are bound simultaneously.

Third, we demonstrate a technique to define standalone DSL functions (Section 4). That is, instead of writing a closed DSL expression in which every DSL function is defined locally, we want to write a standalone DSL function that may be reused across multiple DSL expressions. This technique enables code deduplication and the implementation of libraries containing reusable DSL functions.

Last, to round things up, we list auxiliary techniques of which some are known from folklore and come in handy for certain DSLs. This includes type annotations for DSL expressions, named function arguments through records, and the ability to prohibit nesting of binders (Section 5).

Most code is written in the pure functional programming language Clean [5, 32].[1] Yet, it can be translated to Haskell in a straightforward manner. In Section 3.2.3 and 5.3, we used Haskell instead of Clean due to some language features that are available in GHC but not in Clean.

2 The Example DSL

We demonstrate our ideas with an example DSL that is iteratively enriched by new constructs. The DSL contains basic functionality for lifting of values, arithmetics, boolean operations, comparisons, and conditionals.

To implement such a DSL, there are two main flavours of embedding techniques, namely deep embedding and shallow embedding. They are closely related and can be transformed isomorphically into each other [1, 10]. In deep embedding, DSL constructs are represented as data types in the host language and interpretations are functions over these data types. In shallow embedding, it is exactly reversed: interpretations on the language are data types and language constructs are functions over these data types.

This paper focusses on a class-based shallow embedding. However, in the source code artefact, we also provide a deep embedded variant using generalized ADTs [29].

[1]All code is available as an artefact [23]. A concise guide to Clean for Haskell programmers is found in [22].

We start with classes for basic operators. As in any class-based embedding, the class variable `v` indicates the *interpretation* of the DSL. The argument of `v` indicates the type of the DSL expression, like in a GADT. 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.[2]

```
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
```

The class variable `v` and its argument are also used to make the embedding sound. The latter guarantees that the type of DSL operators and its arguments match, e.g. that a conjunction is only used on Booleans. By making the type ascribed to a DSL term polymorphic in the interpretation `v`, one prevents inspection of terms, hereby ruling out exotic terms.

For code deduplication, sharing, and recursion, it seems natural to use the host language's recursive let binder. Yet, since it sits on the host level, we only reap its benefits during evaluation. For interpretations like printing, whose output leaves the host language, the bound expressions are not shared, and recursive expressions would lead to an infinite output. Thus, we add DSL-level binders to delegate sharing and fixation to the interpretation at hand [27]. Specifically, we add a `cons` binder for constants and a `fun1` binder for single argument functions.

```
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 `fun1` binder enables defining DSL unary functions. Compared to HOAS, in which one typically has functions of type `v (a → b)`, in our approach functions have the type `(v a) → v b`, hence have the advantage that we need neither an explicit apply nor an explicit variable construct. The `fun1` binder also permits recursion due to the function name being in scope inside its body. While it is possible to implement recursion with a fixed-point combinator [3], we chose a different approach here since we want our binder to support binding, sharing, *and* recursion, as laid out in Section 1.1.

The following definition of the factorial function `ex1_v2` serves as an example for a recursive function.[3] There is no higher meaning to the trivial definition of `one`, it simply serves as an illustration of the `cons` function. The argument of `ex1_v2` is the value of the argument

[2]Functions in Clean have a fixed arity, so arguments are parenthesized when needed [32, §3.7].

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

of the factorial function, i.e. the value is inlined. This illustrates how the host language and the eDSL mix. For recursive functions, like `fac`, it is essential that we combine the function definition and its application in a single construct. The defined function is in scope of its own body as well as its applications.

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

Currently, the `fun1` construct allows first-order functions with one argument only. In Section 3, we get rid of this limitation through a generalization over the type of the bound expressions.

2.1 Evaluation

Evaluation of DSL expressions takes place in a strict identity monad `E` and is completely standard. The definition of `E` and its relevant class instances are listed in Appendix A.

For the evaluation of the `cons` and `fun1` class, we uniformly substitute the term's body for all applications using a cyclic definition. The effectiveness of this technique depends crucially on lazy evaluation.

```
instance lit E where lit a = pure a
instance arith E where
  (+.) x y = (+) <$> x <*> y
  (-.) x y = (-) <$> x <*> y
  ...

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

In shallow embedding, the interpretation is embedded in the data type. Hence, evaluating expressions in our DSL is a matter of unpacking the data type `E`, as seen in `eval`.

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

For example, the evaluation `eval (ex1_v2 4)` of a call to the factorial function produces the value 24.

2.2 Printing

The printing interpretation for this DSL is slightly more complex. We use the tooling shown in Appendix B that uses a reader writer state monad, called `RWS`. Here, the log of the writer is a list of strings, the state contains fresh identifiers and the environment of the reader is not used. To generate concise prints we print literals without `lit` keyword and omit the dot suffix of infix operators. For brevity in presentation, we do not attempt to print minimal parentheses nor strive for an efficient implementation. Printing of the other basic classes is similar and shown in Appendix B.

```
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 [")"])
```

To print the binders, we use an idea similar to the previous section. For every bound expression, we generate a fresh identifier.[4] For brevity, we reuse the printing instance for the more general binders that we introduce in Section 3 and 3.2 below. In addition, it shows that this is indeed a special case of the more general binders.[5]

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

Evaluating `printMain (ex1_v2 4)` to print an application of the factorial function produces:

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

Note that, although we use a single `cons` and `fun1` definition in this example, these constructs can be nested arbitrarily. Each of these binders yields a valid value of type `v a`.

3 Handling Arity and Recursion

We now extend the above binders in two directions: the arity of functions and mutual recursion. Regarding arity, we demonstrate a technique that limits the functions to particular arities and another technique that enables functions of arbitrary arity.

3.1 Fixed Arity

The `cons` and `fun1` binders in the DSL above have no argument or a single argument respectively. We generalize both binders to a single unified binder by replacing the argument type `v a` of `fun1` by a more general `a` type variable which is exposed as a type class argument.

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

This allows us to make instances of this class for tuples of varying size to resemble functions of varying arity. By making a `funa ()` `v` instance for the unit type, we enable C-style functions without arguments. Note that this does not imply that tuples become part of the types of the DSL. The tuples sit in the host language and are just a way to denote multiple arguments in the DSL. This technique is useful to limit the number of arguments.

Consider for example the Ackermann function, 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)
```

[4]Unfortunately, the names given by the user of the DSL are not accessible without template metaprogramming, see also Section 4.
[5]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 since, whenever required, we make this name an additional parameter of a helper function that is called in the actual print class.

The downside of this approach is that one needs to list `funa a v` instances for all types `a` that are bound inside the expression. To reduce the verbosity, you may use the quantified constraints [4] type system extension to aggregate these constraints.

3.1.1 Evaluation. Evaluation is identical to the instances shown above. We implement it in terms of the most general binder `def` (Section 3.2) as it is just a specialization.

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

Evaluating the Ackermann program shown above produces 7 as expected.

3.1.2 Printing. Again, printing the binder is more work. The instance for `(Print a) Print` is equal to the `fun1 Print` instance. In the same style, we make an instance for `() Print` to allow functions with zero arguments, or a unit argument to be more precise. We showcase the instance for a pair, as used in the example `ex2` above. The difference to the single argument function is that we generate two symbolic arguments for the body and handle a tuple of arguments in 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 λn"] ≫| runPrint main)
```

This prints our Ackermann function 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))
```

3.2 Arbitrary Arity

By yet another generalization, we define a binder for functions of arbitrary arity that can also be curried, i.e. there is no need to pack the arguments in a tuple. Furthermore, this binder binds not only functions but also constants. Again, we replace `a → v b` in the definition by a more general `a` type variable and supply appropriate instances.

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

We illustrate the power of this approach by an implementation of the algorithm that computes powers in logarithmic time.

```
ex3 = def λone = lit 1 In
      def λtwo = lit 2 In
      def λodd = (λn.
          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 symbols one and two denote constants, odd is function of a single argument, and pow is a function of two arguments. Also observe that the functions are recursive.

The `def` construct is in principle also general enough to allow binding higher-order functions. However, their availability depends on the interpretation `v`. Evaluation for example handles higher-order functions without any additional machinery since they are a core feature of the host language. In contrast, the printing of higher-order functions remains future work.

3.2.1 Evaluation. The evaluation again is 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 allows 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 control the allowed arguments in detail.

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

3.2.2 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
```

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 ≫= λn.
    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.[6]

```
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)
```

3.2.3 Restricting the Bindable Expressions. While the liberal def allows binding functions of arbitrary arity, we have no guarantees that these functions are from the object language. So one can use it to define host language constants and functions like in the following example:

```
one = def λone = 1 In lit one
```

Instead, we would like to restrict the type of the bindable expressions to liftable types, i.e., types with a shape like v a, (v a) (v b) → v c, and ((v a) → v b) (v a) → v b. To do so, we use an empty type class with selected instances to enumerate all types that we consider lifted.[7]

```
class Lift v a
instance {-# INCOHERENT #-} (Lift v a, Lift v b) ⇒ Lift v (a → b)
instance {-# INCOHERENT #-} Lift v (v a)
```

Since the instance resolution only works as soon as the type variable v is instantiated, we get overlapping instances, regardless of any overlap pragmas. Hence, we have to mark these instances as incoherent and mention the recursive case first so that the instance resolution matches the arrow type whenever applicable.[8]

With small changes to the instances given above, we are also able to alter the set of lifted types. For example, to not allow any type but only integers and Booleans to be lifted, we would have to replace the Lift v (v a) instance by a Lift v (v Int) and a Lift v (v Bool) instance. Or, to disallow higher-order functions, the (Lift v a, Lift v b) ⇒ Lift v (a → b) instance would have to be replaced by a Lift v b ⇒ Lift v (v a → b) instance.

With this machinery in place, we restrict the bindable expressions of the def combinator by constraining the type of the bound expressions to the Lift type class.

```
class Def v a where
  def :: Lift v a ⇒ (a → a `In` v b) → v b
```

Note that this idea is that it does not require any changes on the use site because it is merely an artificial restriction onto the types. Furthermore, declaring types as lifted is as easy as adding another instance of the Lift class.

3.3 Mutual Recursion

With the current eDSL binders, we can define mutually-recursive expressions as long as we only need to expose one of them. The trick is to define the expression that shall not be exposed inside the body of the expression that shall be exposed. For example, the even function is defined mutually-recursively through the odd function.

```
isSevenEven =
    def λone = lit 1 In
    def λisZero = (==.) (lit 0) In
    def λisOne = (==.) one In
    def λeven =
            def λodd = λn. If (isZero n)
                    (lit False)
                    (If (isOne n) (lit True) (even (n -. one)))
            In λn. If (isZero n)
                    (lit True)
                    (If (isOne n) (lit False) (odd (n -. one)))
    In even (lit 7)
```

Compilation of such expressions requires advanced techniques like lambda lifting or closure conversion due to the nested binders. Also, being able to expose only one of the mutually-recursive expressions is a significant drawback.

[6]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.

[7]Since the type class that we want to define does not have any members and its instances are incoherent, an implementation in Clean would involve a significant amount of boilerplate code. Hence, we give a definition in Haskell.

[8]See §6.8.8.5 of the GHC documentation: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html.

Hence, we demonstrate a technique that allows to expose multiple mutually-recursive expressions on the same level. The basic idea is to define not only one expression but a tuple of expressions, so that our previous example of the mutually-recursive even and odd functions becomes:

```
evenOdd =
    def λone = lit 1 In
    def λisZero = (λn. n ==. lit 0) In
    def λisOne = (==.) one In
    def λ(odd,even) =
       ( λn. If (isZero n) (lit False)
               (If (isOne n) (lit True) (even (n -. one)))
       , λn. If (isZero n) (lit True)
               (If (isOne n) (lit False) (odd (n -. one)))
       )
    In odd (lit 7)
```

Intuitively, we expect the evaluation of a tuple of expressions to be already handled by the generic `def a E` instance. Even with the type of bound expressions restricted using the `Lift` type class, we expect it to work by declaring pairs as lifted types through a simple `(Lift v a, Lift v b) ⇒ Lift v (a, b)` instance. However, since the `(_, _) = ···` binder is strict in the tuple constructor, naive evaluation of a pair of binders leads to an infinite recursion.[9] Hence, we need to perform lazy pattern matching on the tuple, either at use site or at fixation site. To not put any burden onto the user, we change the fixation site by providing an explicit evaluation instance for tuples.

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

To print 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.

```
instance def (a,b) Print | defType a & defType b where
  def f = P (fresh ≫= λn. 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)
```

While we covered mutual recursion for two expressions only, it should be noted that this approach scales to three or more mutually-recursive expressions analogously.

4 Standalone Functions

Embedded function definitions with multiple interpretations as 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 data types as well as functions with a different number or type of arguments.

There are three 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 definition 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. 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 as if we 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 continuation passing style. This method and the method before result in very complex types become more complex and all library functions still need to be declared explicitly.

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


ex5 = withFac λfac → fac (lit 4)
```

4.1 Named Functions

A more radical approach uses only named definitions. These definitions have a user-defined ID that must be unique. This ID solves the lost names' problem in printing and is an 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.

```
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 interpretation 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 even and odd example below shows that this allows mutual recursion without the need to define the functions simultaneously as in Section 3.3.

[9]To delve into the problem, transform the HOAS binder into an open recursion style and then expand its fixation by means of equational reasoning.

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

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

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

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

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

4.2 Evaluation

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

4.3 Printing

For the printing interpretation, we have to work a little harder. Here, we require 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 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. After we are done with printing, we collect all definitions from the mapping with `printAll`. Like in Section 3, we 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 3.2.2 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 use 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 could 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 [36] 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 [19]. 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 is mitigated by incorporating the arguments in the identifier, as done below.

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

5 Auxiliary Techniques

5.1 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 [31, §11.4]. Our host language Clean has no syntax for adding type ascriptions to expressions.[11] With the definition of type witnesses and two combinators we 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

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

We define two combinators named `:::` and `→.` to ascribe the type. 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 `(\\x. x +. lit 1) ::: Int →. Int`.

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

[11] Something that is available in e.g. Haskell98 [28, §3.16].

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

This approach is very similar to the use of `Proxy` and type ascriptions. 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 the `World` [2] or the `Void` type. Though `undef` can be used as the value is never evaluated anyway.

In this approach these witnesses are never materialized in any interpretation of the DSL. The definition of `:::` discards them. An application is the factorial definition below. 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 → (λn. If (equ n one ::: Bool)
            one
            (n *. fac (n -. one) ::: Int)) ::: Int →. Int In
    fac (lit n) ::: Int
```

5.2 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.[12] The record `Loan` introduces names for these values. All values are `Real` numbers in our DSL. Hence, they are interpretations `v` on such a value. This interpretation `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}
```

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 3.2.2 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 becomes rather verbose. A direct implementation yields for our compound interest example:

```
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 })
```

5.3 Prohibiting Nested Definitions

The DSL binders introduced above can be nested arbitrarily. It is convenient that arguments of a function are available in a nested definition. See for instance example `ex3` (Section 3.2). The arguments `x` and `n` of function `pow` are used in the body of `y`. However, for some DSLs this is undesirable. Nested definitions require special attention in code generation to make the implicit arguments reachable. Well-known solutions to handle nested definitions are closure conversion and lambda lifting.

Using a slightly different function definition class, we prevent nested definitions. The key is to wrap the expression in which the bound term is used in a data type. Here we use a record, but an algebraic data type works equally well. Also note that we provide Haskell code here because the `Lift` type class from Section 3.2.3 which is implemented in Haskell is used.

```
data Main v b = Main { runMain :: v b }
```

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 avoid name clashes with the DSL binder `def` that was introduced earlier, the binder for top-level terms only is called `defM`.

```
class DefM a v where
    defM :: Lift v a ⇒ (a → a `In` Main v b) → Main v b
instance DefM a E where
    defM f = let (body `In` expr) = f body in expr
```

Obviously, enforcing only top-level terms works only in a DSL where the `def` binder 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. Note that for this to work, it is important to restrict the type `a` of bindable terms by the `Lift` type class as laid out in Section 3.2.3. Otherwise, nesting in the form of

```
defM λt1 → (defM λt2 → ··· `In` Main ···) `In` t2
```

is still possible. Apart from the wrapping `Main` type, `defM` definitions are identical to `def` definitions. Also, all instances are similar. To prevent that the user of the DSL fools the system, the `runMain` accessor is only available inside the implementation of the DSL.

Yet another variant of the factorial example, this time with top-level terms only, is given below.

```
facM n =
  defM λone → lit 1 `In`
  defM λis0 → (==.) (lit 0) `In`
  defM λfac → (λn. If (is0 n) one (n *. fac (n -. one))) `In`
  Main $ fac (lit n)
```

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

6 Code Generation for Nested Functions

To show that this approach is not limited to simpler interpretations, we introduce code generation as a new interpretation for the example DSL. For this illustration, we use the standalone function variant (Section 4). To handle nested functions, we adjust the function definition by adding arguments for the captured variables and all applications are extended with the captured variables. This obtains the same effect as lambda lifting [14].

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. Extensible ADTs [32, §5.1.5], data types *à la carte* [37], or classy deep embedding [21] 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 all basic data types are converted to an integer (using `toInt`) 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 `compOrRetrFunction`. The third argument of this function is a function that, given a label, provides a representation of the arguments to the function definition. The `compOrRetrFunction` 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 (compOrRetrFunction 1 name
       (λlbl → f (C (tell [Arg lbl 0])))
    ≫= callFunction (runCompile x))
```

6.2.1 Generating the Definition. The `compOrRetrFunction` function first checks if the function has been encountered before by looking it up in the `funMap`. If this is the case, we return the label immediately. Otherwise, we generate a fresh label and store it with the name in the `funMap`. Then we execute the body using the provided function `f` while capturing the output using `censor` and `listen` — similarly to what was done in the printing interpretation. The instructions are placed after performing the lambda lifting. Finally, the instructions for the function call preparation is stored in the `funcalls` field. Later, the `Marker` referencing this label is replaced by this sequence of instructions.

```
compOrRetrFunction :: Int String (Int → Compile a) → CompileM Int
compOrRetrFunction 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 call graph [26]. 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
```

6.2.2 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

The idea of embedding DSLs is due to Landin [20]. Hudak [12] introduced the notion of embedded modular DSLs in functional programming. Pfenning and Elliott [30] introduce, with HOAS, a simply typed λ-calculus enriched with products and polymorphism to express syntax trees. This in turn is an extension of the second-order term language of Huet and Lang [13].

Using a type parameter is known from Parametric HOAS [7] and Boxes Go Bananas [38]. While those techniques use the type parameter to prevent exotic terms, we use the type parameter to indicate the type of the DSL construct. This leverages the host language's type checker to rule out undesirable terms and to infer the type. In that sense, it is similar to the type parameter in GADTs. Yet, in contrast to a deep based embedding, our shallow embedding is simpler in that it omits some quantifiers.

A consequence of defining identifiers as function arguments and imposing the normal type constraints is that we can define polymorphic functions in the DSL, but we can only use them with a single type instance. Serrano et al. [35] introduce a type extension to allow the polymorphic use of such a definition at the cost of an explicit type definition. In Koopman and Lubbers [18] we show that we can achieve the same effect at the cost of an additional constructor in function definitions and hence an explicit function application operator. Oliveira and Löh [27] introduces techniques for expression sharing and recursion that we reuse here.

Atkey et al. [1], Matsuda et al. [24] converts the finally-tagless representation of lambda-terms to de Bruijn-indexed terms to fix the mismatch between host and guest language function semantics. This allows open terms as well other interpretations of function application. Our DSL functions are well-typed, implying that undesired terms are prevented, and can be recursive. By selecting the desired version of the definition, we can impose additional constraints, like a controlled number of arguments or first-order functions.

The original HOAS idea based on ADTs suffers from problems with exotic or junk terms [7]. Our class-based HOAS approach exposes the issue if the interpretation is known since it allows the user to exploit internals of the interpretation. Making an expression parametric in the interpretation, i.e. quantifying over the interpretation v, resolves it.

Intentional analysis, like program optimization and partial evaluation, is generally hard with HOAS and shallow embeddings because we cannot inspect functions. This is solved for HOAS [1] and class-based shallow embeddings [6, 16] by converting to-and-from a deep representation or by using an intermediate data type capturing just enough state. Gill [11] introduced an IO-based solution to observable sharing that uses type functions to provide type-safe observable sharing. McDonell et al. [25] expand on this for a typed syntax tree.

Carette et al. [6], Kiselyov [16] 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. The direct approach for evaluation is similar to Fegaras and Sheard [8] for a DSL that is always evaluated. Since we control what is done with applied function occurrences, we have fewer problems with intentional analysis and can easily make interpretations of the DSL like code generators and pretty printers.

Kiselyov [17] explicates the challenges in generating safe low-level code. These techniques should integrate with our approach.

Every class constraint needed in some interpretations is part of the type classes constructing our DSL. Jones et al. [15] shows how these constraints can be moved to the interpretations that require them.

Naming components of the language is not new. For example, Frost et al. [9] used this technique to detect left recursion in parser combinators. We show that it can be used for DSL functions as well that can be recursive themselves.

Implementing named arguments by means of a record, as described in Section 5.2 has been discussed extensively in white and grey literature. To the best of our knowledge, it has never been published in a DSL context before, although it is known from folklore.

8 Conclusion

This paper shows that it is possible to define strongly-typed functions as part of an embedded DSL with multiple interpretations in a strongly-typed host language. Key to these type-safe definitions are two observations. First, the type parameter of DSL expressions mirrors its type. Second, identifiers for functions are provided in HOAS style, hence giving the identifier itself a type. The type checker of the host language then unifies the identifier's type with the type of the expression that the identifier is bound to. The only syntactical overhead is a DSL binding construct and a lambda for defining the identifier.

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

We presented gradual additions to the binder to allow more general types of functions. In the first iteration, functions accept exactly one argument that is a single DSL value. In the second iteration, functions accept multiple arguments, encoded as a tuple of DSL values. This binder also showed how to restrict the arity of functions. In the third iteration, the most general case, DSL functions support any number of arguments, just like functions in the host language. In addition, we demonstrated a technique to restrict the types allowed in a DSL by enumerating these types through an empty type class. Hereby, the liberality of the DSL binder is decoupled from a restriction on the DSL types.

Besides enabling functions of arbitrary arity, our binder also handles recursive functions, or, more general, recursive terms. Furthermore, we illustrated a technique to extend the binder to mutually-recursive terms: Instead of binding a single expression only, we bind a tuple of expressions simultaneously.

These ideas were complemented with an approach for defining standalone functions to facilitate reuse. This enables us to implement libraries of DSL expressions.

Like with many eDSLs, the type errors that a user faces are difficult to comprehend. They are intertwined with the machinery which is not meant to be shown to the user, like e.g. the type classes for the restriction of eDSL types and the printing of arbitrary arity functions. There are well-known techniques to improve error messages of eDSLs [33, 34].

The printing of arbitrary arity functions shows another problem: To pattern-match on arrow types, we need another type class that is not related to the DSL itself. This implies that, in order to print expressions, printing-specific type class constraints have to be listed in the expression's typing context. Yet, the nature of the printing implementation suggests that it underlies a generic concept that remains to be uncovered in forthcoming work.

An interesting property of the DSL binder is that, from the perspective of the type system, it allows higher-order functions. However, we do not know of an implementation of higher-order functions for non-trivial interpretations. Investigating their implementation seems like another exciting opportunity for future work.

References

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

[2] John Backus, John H. Williams, and Edward L. Wimmers. 1990. An Introduction to the Programming Language FL. In *Research Topics in Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 219–247.

[3] Henk P. Barendregt. 2012. *The Lambda Calculus Its Syntax and Semantics*. Number 40 in Studies in Logic. College Publ, London.

[4] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 148–161. doi:10.1145/3122955.3122967 event-place: Oxford, UK.

[5] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

[6] 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. doi:10.1017/S0956796809007205

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

[8] Leonidas Fegaras and Tim Sheard. 1996. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/237721.237792

[9] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Practical Aspects of Declarative Languages*, Paul Hudak and David S. Warren (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–181.

[10] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 339–347. doi:10.1145/2628136.2628138 event-place:

Gothenburg, Sweden.

[11] Andy Gill. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/1596638.1596653 event-place: Edinburgh, Scotland.

[12] Paul 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. doi:10.1109/ICSR.1998.685738

[13] Gérard P. Huet and Bernard Lang. 1978. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica* 11 (1978), 31–55. doi:10.1007/BF00264598

[14] Thomas Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 58–69. doi:10.1145/502874.502880

[15] Will Jones, Tony Field, and Tristan Allwood. 2012. Deconstraining DSLs. *ACM SIGPLAN Notices* 47, 9 (Oct. 2012), 299–310. doi:10.1145/2398856.2364571

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

[17] Oleg Kiselyov. 2024. Generating C: Heterogeneous metaprogramming system description. *Science of Computer Programming* 231 (2024), 103015. doi:10.1016/j.scico.2023.103015

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

[19] Shriram Krishnamurthi. 2001. *Linguistic reuse.* PhD Thesis. Rice University, Houston, USA.

[20] Peter J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (March 1966), 157–166. doi:10.1145/365230.365257

[21] Mart Lubbers. 2022. Deep Embedding with Class. In *Trends in Functional Programming*, Wouter Swierstra and Nicolas Wu (Eds.). Springer International Publishing, Cham, 39–58. doi:10.1007/978-3-031-21314-4_3

[22] Mart Lubbers and Peter Achten. 2024. Clean for Haskell Programmers. arXiv:2411.00037 [cs.PL] https://arxiv.org/abs/2411.00037

[23] Mart Lubbers, Pieter Koopman, and Niek Janssen. 2023. Source code for the paper Shallowly Embedded Functions. doi:10.5281/zenodo.10225278

[24] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023). doi:10.1145/3607830 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[25] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/2500365.2500595 event-place: Boston, Massachusetts, USA.

[26] Marco T. Morazán and Ulrik P. Schultz. 2008. Optimal Lambda Lifting in Quadratic Time. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 37–56.

[27] Bruno C. d. S. Oliveira and Andres Löh. 2013. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation* (Rome, Italy) *(PEPM '13)*. Association for Computing Machinery, New York, NY, USA, 87–96. doi:10.1145/2426890.2426909

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

[29] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 50–61. doi:10.1145/1159803.1159811

[30] Frank Pfenning and Conal 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. doi:10.1145/53990.54010

[31] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press, Cambridge, Massachusetts.

[32] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2021. *Clean Language Report version 3.1.* Technical Report. Institute for Computing and Information Sciences, Nijmegen. 127 pages.

[33] Alejandro Serrano. [n. d.]. *Type Error Customization for Embedded Domain-Specific Languages.* Ph. D. Dissertation.

[34] Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling expression-level type errors by type-level programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages* (Bristol, United Kingdom) *(IFL '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 15 pages. doi:10.1145/3205368.3205370

[35] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. doi:10.1145/3408971

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

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

[38] Geoffrey Washburn and Stephanie Weirich. 2003. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. *SIGPLAN Not.* 38, 9 (Aug. 2003), 249–262. doi:10.1145/944746.944728

A Evaluation Tooling

Throughout this paper we use evaluators and printing interpretations 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 interpretations.

For the evaluation of DSL variants, we use the simplest type possible that generalizes to all monads.[13] 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.

```
:: 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}
```

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

```
:: 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 [")"])
```