Using Circular Programs for Higher-Order Syntax

Functional pearl

Emil Axelsson Koen Claessen
Chalmers University of Technology
{emax,koen}@chalmers.se

Abstract

This pearl presents a novel technique for constructing a first-order syntax tree directly from a higher-order interface. We exploit circular programming to generate names for new variables, resulting in a simple yet efficient method. Our motivating application is the design of embedded languages supporting variable binding, where it is convenient to use higher-order syntax when constructing programs, but first-order syntax when processing or transforming programs.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Syntax; D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords higher-order syntax; embedded languages; circular programming

1. Introduction

Imagine a simple Haskell data type for expressions of the lambda calculus:

```
data Exp = Var Name      -- Variable
         | Lam Name Exp  -- Abstraction
         | App Exp Exp   -- Application
         deriving (Show)
```

The `Var` and `Lam` constructors use explicit names to refer to variables, where names belong to the abstract type `Name`. When constructing expressions in this representation, we have to keep track of the scope of bound variables. As an example, the term $\lambda x.(\lambda y.y)(\lambda z.z)x$ – a verbose definition of the identity function – can be represented as follows (assuming an integer representation of names):

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500614

```
app :: Exp → Exp → Exp
lam :: (Exp → Exp) → Exp
```

Figure 1: Higher-order interface for the lambda calculus

```
identity_FO :: Exp
identity_FO = Lam 1
  (App (App (Lam 2 (Var 2)) (Lam 3 (Var 3)))
       (Var 1)
  )
```

Because it is so easy to mix up variable names, it is common to instead use *higher-order syntax* (HOS) in embedded languages. A HOS interface for the lambda calculus is given in Fig. 1. HOS allows us to write the above example in a much more convenient form:

```
identity :: Exp
identity = lam (λx →
    app (app (lam (λy → y)) (lam (λz → z)))
        x
  )
```

By using binding in the host language to represent binding in the object language, it is impossible to refer to unbound variables. Note that the HOS interface does not include a constructor for variables. Those are implicitly introduced by `lam`.

How can we implement the interface in Fig. 1 without changing the original `Exp` data type? That is the question that this pearl will answer. As we shall see in Sec. 4, this problem is highly relevant to the implementation of embedded languages.

First attempt. The difficulty is in implementing `lam`. A first attempt may lead to the following code:

```
lam f = Lam n (f (Var n))
  where
    n = ???
```

Now, the problem is to choose a name `n` that does not interfere with other names in the expression. The name must be

```
bot   :: Name
prime :: Name → Name

-- prime law:
∀ a . prime a > a

-- Maximum:
(⊔) :: Name → Name → Name
m ⊔ n | m ≥ n = m
      | n > m = n
```

Figure 2: Creation and manipulation of names

chosen so that (1) the binding does not accidentally capture free variables in the body, and (2) uses of the new variable are not captured by other bindings in the body.

Abstract representation of names. In order to allow freedom in the representation of names, we will use the operations in Fig. 2 to create, manipulate and reason about names. A name can be any totally ordered type implementing the given interface. The law for `prime` states that this function always increases the order of a name. Since no operation decreases the order of a name, we can argue that `bot` is the smallest name, as long as we only create names using the given interface.

For the examples in this pearl, we will use the following implementation of names:

```
type Name = Integer
bot       = 0
prime     = succ
```

Note also that `(⊔)` is equivalent to the `max` function.

2. Alternatives

This section gives two alternative implementations of the HOS interface. These provide sufficient background to develop our new method in Sec. 3. Some additional alternatives are mentioned in the related work (Sec. 4).

2.1 Threading a name supply

As a reference, let us first consider a non-solution – one that does involve changing the `Exp` type. The idea is to prevent capturing by using a unique name in each binding. This can be done by threading a name supply through the `lam` and `app` functions, which requires us to change `Exp` to a state-passing function:[1]

```
type ExpNS = Name → (Exp, Name)
```

[1] We will use subscripts as a way to distinguish different implementations of similar functions and types throughout the paper.

```
fromExpNS :: ExpNS → Exp
fromExpNS e = fst (e (prime bot))
```

The implementation of application will just thread the state, first through the function and then through the argument. Abstraction is a bit more involved.

```
appNS :: ExpNS → ExpNS → ExpNS
appNS f a = λn →
    let (f',o) = f n
        (a',p) = a o
    in  (App f' a', p)

lamNS :: (ExpNS → ExpNS) → ExpNS
lamNS f = λn →
    let var   = λo → (Var n, o)
        (a,p) = f var (prime n)
    in  (Lam n a, p)
```

The incoming name `n` is used for the new variable. The variable expression `var` just passes its name supply through unchanged. The body `(f var)` is given `(prime n)` as the incoming name, which ensures that all its bindings will use names that are different from `n`.

An example shows how the names are chosen:

```
*Main> fromExpNS identityNS
Lam 1 (App (App (Lam 2 (Var 2)) (Lam 3 (Var 3)))
(Var 1))
```

The expression $\texttt{identity}_{\texttt{NS}}$ is defined as `identity` but with $_{\texttt{NS}}$ subscripts on `app` and `lam`. We will use the same convention for $\texttt{identity}_{\texttt{SPEC}}$ below.

The name supply method does not solve the original problem, as it uses a different representation of expressions. Also, the tedious state threading in $\texttt{app}_{\texttt{NS}}$ and $\texttt{lam}_{\texttt{NS}}$ leaves a bad taste in the mouth. On the more practical side, the fact that $\texttt{Exp}_{\texttt{NS}}$ is a function leads to some additional problems:

- It is not directly possible to pattern match on expressions. Pattern matching is commonly used to define smart constructors that simplify expressions on the fly [6].
- It is not possible to observe any implicit sharing [5, 7] in the expression. After all, a shared sub-expression can appear in many contexts with different name supplies.

For all these reasons, we leave $\texttt{Exp}_{\texttt{NS}}$ behind and look for a better alternative.

2.2 Speculative naming

Recall, the problem is to implement `lam` without changing the `Exp` type, which means that there will not be any name supply available. Let us thus return to our original attempt at defining `lam`:

```
lam f = Lam n (f (Var n)) where n = ???
```

We have no name supply, yet we need to pick a name that does not interfere with the body of the expression. One way to solve this puzzle is to speculatively evaluate the function `f` to find out which names are used in the body, then pick a different name for the variable, and apply `f` again:

```
lamSPEC :: (Exp → Exp) → Exp
lamSPEC f = Lam n′ (f (Var n′))
  where
    ph = Var bot     -- Placeholder
    n  = maxV (f ph) -- Speculation
    n′ = prime n
```

The placeholder `ph` used in the first application of `f` uses the smallest name `bot`, which is assumed only to be used for speculative evaluation, not for bound variables. The $\mathtt{max}_\mathsf{V}$ function simply traverses the body to find the greatest occurring variable name:

```
maxV :: Exp → Name
maxV (Var n)   = n
maxV (App f a) = maxV f ⊔ maxV a
maxV (Lam _ a) = maxV a
```

Selecting `n′ = prime n` ensures absence of capturing: there could be other variables of that name in scope, but they are anyway not used in the body.

Since we are now constructing the `Exp` type directly, the $\mathtt{app}_\mathsf{SPEC}$ constructor is identical to `App`:

```
appSPEC = App
```

Our running example shows how the names are chosen:

```
*Main> identitySPEC
Lam 2 (App (App (Lam 1 (Var 1)) (Lam 1 (Var 1)))
(Var 2))
```

So, the method works, but can you spot the problems with the implementation of $\mathtt{lam}_\mathsf{SPEC}$?

One problem is that $\mathtt{max}_\mathsf{V}$ has to traverse the whole body, leading to quadratic complexity in expressions with nested lambdas. However, there is a much more severe problem: The function `f` is applied twice in each lambda, which means that an expression with n nested lambdas requires 2^n applications!

Deeply nested lambdas are not uncommon in embedded languages where variable binding is used to represent shared sub-expressions (as, for example, in reference [6]). Thus, the exponential complexity renders the above method unusable in practice.

3. Our method: circular speculation

The speculative application in the previous method is used to resolve the circular dependency arising from the fact that we need to examine the body before constructing it. In a classic paper, Richard Bird poses the Repmin problem that has a similar circular dependency [2]. The Repmin problem is to define a function that converts a tree into a tree of identical shape, but where all leaves have been replaced by the minimal leaf in the original tree. A naive solution would traverse the tree twice – once to find the minimal leaf, and once to construct the new tree. Bird's solution uses circular programming to collapse the two traversals into one.

For the Repmin problem, circular programming reduces two traversals into one more complicated traversal, which makes it unclear if the approach saves any computation at all. However, in case of nested traversals, cutting the number of recursive calls in each step can reduce the complexity class!

Can we use circular programming to avoid the exponential blowup in $\mathtt{lam}_\mathsf{SPEC}$? Let us try:

```
lamCIRC :: (Exp → Exp) → Exp
lamCIRC f = Lam n′ body
  where
    body = f (Var n′)
    n    = maxV body
    n′   = prime n
```

This version avoids the separate speculation by using the correct name right away. Although this function type checks, unfortunately it does not work.

Why?

The problem is that $\mathtt{max}_\mathsf{V}$ can no longer distinguish the new variable from other variables in the body. Thus, $\mathtt{max}_\mathsf{V}$ returns a name that is at least as high as `n′`, giving $\mathtt{n} \geq \mathtt{n'}$. At the same time, we have `n′ = prime n` which gives us $\mathtt{n'} > \mathtt{n}$. This contradiction manifests itself as an infinite loop in $\mathtt{lam}_\mathsf{CIRC}$.

3.1 A different perspective

The simple speculative method involved finding a name that is not used in the body of a binding. As previously said, this ensures absence of capturing. However, another way to avoid capturing is to only look at the variables that are bound in the body, and pick a name that is not bound. Then there is still a risk of capturing a free variable, but as long as all bindings and variables are created using the same method, this will never happen (see Sec. 3.2).

The function that finds the greatest bound variable is a slight variation of $\mathtt{max}_\mathsf{V}$:

```
maxBV :: Exp → Name
maxBV (Var _)   = bot
maxBV (App f a) = maxBV f ⊔ maxBV a
maxBV (Lam n a) = n ⊔ maxBV a
```

```
app :: Exp → Exp → Exp
app = App

lam :: (Exp → Exp) → Exp
lam f = Lam n body
  where
    body = f (Var n)
    n    = prime (max_BV body)
```

Figure 3: Implementation of the higher order interface using circular programming

Fig. 3 gives a complete definition of the HOS interface using circular programming. The trick is that since $\mathsf{max}_{\mathsf{BV}}$ does not look at `Var` constructors, it can produce a value without forcing evaluation of the new variable. Thus, the infinite loop is broken.

For the `identity` example (but not in general), our method chooses the same names as the simple speculative method:

```
*Main> identity
Lam 2 (App (App (Lam 1 (Var 1)) (Lam 1 (Var 1)))
(Var 2))
```

3.2 The law of the jungle: to capture or to be captured

When choosing a name for a new binding, there are two problems we want to avoid: (1) the binding captures a free variable in the body, and (2) uses of the new variable are captured by other bindings in the body. For closed terms, capturing can only happen when a binder shadows a variable in scope. Thus, to check for absence of capturing, it is enough to check for absence of shadowing:

```
safe :: Exp → Bool
safe (Var _)   = True
safe (App f a) = safe f && safe a
safe (Lam n a) = n > max_BV a && safe a
```

The above function checks that no binding is shadowed by another binding in its body. The requirement that each binding introduces a variable that is greater than all bound variables in the body is overly conservative (it is enough that the new variable is distinct from the bound variables in the body), but suffices for our purposes. Note that by assuming closed terms and only considering shadowing, we can reason about capture-avoidance purely in terms of binders, ignoring any uses of the variables. We trust that our HOS implementation only produces closed expressions.

We will argue for the correctness of our method by showing that any term constructed using the HOS interface – `app` and `lam` – is safe. To simplify reasoning, we only consider Haskell terms t built using direct application of those functions and variables.

Definition 1. A HOS term is defined by the following grammar:

$$\begin{array}{rcl} t & ::= & v \\ & | & \mathtt{app}\ t\ t \\ & | & \mathtt{lam}\ (\lambda v\ .\ t) \end{array}$$

Definition 2. We use $c \vdash t \Downarrow e$ to denote evaluation of the term t to value e (of type `Exp`) in context c. A context is a mapping from Haskell variables to expressions of type `Exp`. We omit the definition of evaluation from the paper.

Definition 3. We extend the notion of safety to contexts: $\mathsf{safe}_{\mathsf{CXT}}\ c$ holds if all variables in c map to safe expressions.

Theorem 1. Evaluation of a term t in a safe context c results in a safe expression:

$$\forall\, c\, t\, e\ .\ \ \mathsf{safe}_{\mathsf{CXT}}\ c\ \ \&\ \ c \vdash t \Downarrow e \quad \Rightarrow \quad \mathsf{safe}\ e$$

The proof is by induction on terms. The base case, for variables, is proved by noting that looking up a variable in a safe context must result in a safe expression. The case for `app` is shown by a straightforward use of the induction hypothesis. For `lam`, we see in Fig. 3 that it evaluates to `Lam n body`. This expression is safe if `n` is greater than all bound variables in the body and the body is safe. The first requirement is trivially fulfilled by the definition of `lam`. To show that `body` is safe, we expand it to `f (Var n)`, where `f` is equal to $\lambda v\ .\ t$ for some variable v and term t. Thus, the evaluation of the body in context c must be equal to the evaluation of t in context $(v \mapsto \mathtt{Var\ n} : c)$. Assuming that c is safe, this extended context is also safe; hence the induction hypothesis implies that the result of evaluating the body is safe. □

3.3 Achieving linear complexity

So far, we have prevented the exponential complexity in the simple speculative solution by only computing the body once in `lam`. However, since `lam` has to traverse the whole body to find the greatest bound variable, we still have quadratic complexity in the number of nested lambdas. Fortunately, the reasoning in Sec. 3.2 shows us that `lam` actually traverses most of the body in vain!

The `safe` property states that each binding introduces a variable that is greater than all bound variables in the body. This means that we can make an improved version of $\mathsf{max}_{\mathsf{BV}}$ that only looks at the closest binders:

```
max_BV+ :: Exp → Name
max_BV+ (Var _)   = bot
max_BV+ (App f a) = max_BV+ f ⊔ max_BV+ a
max_BV+ (Lam n _) = n
```

Lemma 1. For safe expressions, $\texttt{max}_{\texttt{BV}+}$ gives the same result as $\texttt{max}_{\texttt{BV}}$:

$$\texttt{safe}\ e \quad \Rightarrow \quad \texttt{max}_{\texttt{BV}+}\ e\ =\ \texttt{max}_{\texttt{BV}}\ e$$

Proof by induction on expressions. □

Swapping in $\texttt{max}_{\texttt{BV}+}$ in the definition of `lam` gives us a more efficient implementation:

```
lam₊ f = Lam n body
  where
    body = f (Var n)
    n    = prime (maxBV₊ body)
```

Here, $\texttt{max}_{\texttt{BV}+}$ traverses the body down to the closest binders, which in the worst case means traversing most of the expression. However, since the result is a `Lam` expression, the body will never have to be traversed again by uses of $\texttt{max}_{\texttt{BV}+}$ from lambdas higher up in the expression. Thus, the total effect of all uses of $\texttt{max}_{\texttt{BV}+}$ is *one extra traversal* of the expression. This means that the complexity of building an expression is linear in the size of the expression, giving an amortized complexity of $O(1)$ for each `lam` and `app`.

Theorem 2. Let t_+ range over terms built using `app` and $\texttt{lam}_+$. Evaluation of a term t_+ in a safe context c results in a safe expression:

$$\forall\, c\, t_+\, e\ .\ \texttt{safe}_{\texttt{CXT}}\ c\ \ \&\ \ c \vdash t_+ \Downarrow e \quad \Rightarrow \quad \texttt{safe}\ e$$

Proof using induction on terms t_+ and lemma 1. □

4. Discussion and related work

The problem solved in this pearl is not just a theoretical exercise – it is of great interest to the implementation of embedded domain-specific languages (EDSLs). There are many EDSLs that rely on a higher-order interface towards the user and a first-order representation for analysis and code generation: Lava [3], Pan [6], Nikola [8], Accelerate [10], Obsidian [12] and Feldspar [1], to name some. All of these EDSLs employ some kind of higher-order to first-order conversion.

The simple speculative method in Sec. 2.2 was originally suggested by Lennart Augustsson in a private communication with the authors. One of the anonymous referees also brought to our attention that a similar method is used in reference [4] to construct a first-order term with de Bruijn indexes from a higher-order interface.

A common method for implementing higher-order language constructs is to use *higher-order abstract syntax* (HOAS) [11]. A HOAS version of the lambda calculus would be like our `Exp` but where the `Lam` constructor mimics the type of `lam`:

```
data Exp
    = Var Name
    | App Exp Exp
    | Lam (Exp → Exp)
```

The advantage of this representation is that the constructors have a direct correspondence to the HOS interface in Fig. 1. However, working with this type is not convenient. As soon as we need to look inside a lambda, we need to come up with a variable name to pass to the binding function, which means that the name generation problem we have battled in this paper will reappear in each analysis. Another problem is HOAS to HOAS transformations where the binding functions have to be reconstructed after transforming under a lambda.

Instead, a common approach is to have a separate data type for first-order abstract syntax (FOAS) and a function to convert from HOAS to FOAS. This technique is used, for example, in Accelerate and recent versions of Feldspar. Although the technique is quite useful, it has some practical concerns:

- It requires defining two separate but almost identical data types (or play tricks to merge the two into one).
- Care has to be taken not to destroy implicit sharing during conversion [10].

In a blog post, McBride [9] proposes an implementation of higher-order syntax that, like our solution, does not require a separate HOAS data type. His term representation uses typed de Bruijn indexes and a type class to compute the index of a variable depending on its use site. Since de Bruijn indexes depend on the nesting depth of binders, a value-level implementation would require passing an environment while building expressions (with problems similar to the ones in Sec. 2.1). McBride cleverly avoids the problem by lifting the environment to the type level. Unfortunately, this also leads to more complicated types in the user interface.

It should be noted that our technique using circular speculation assumes that all bindings and variables are created using the `lam` function. This restriction is fine in an embedded language front end where terms are constructed from scratch, but it makes the approach unsuitable for introducing new bindings in existing terms. Consider the following transformation:

```
trans (Lam n a) = Lam n (trans' a)
```

If `trans'` introduces a new binder using `lam`, this introduction will be unaware of the fact that `n` is in scope leading to potential capturing. To avoid capturing, we would have to use `lam` on the right-hand side to introduce a fresh variable `x`, and substitute `x` for `n` in the body (assuming the existence of a suitable substitution function `subst`):

```
trans (Lam n a) = lam (λx → trans' (subst n x a))
```

```
data Exp = Var Name      -- Variable
         | Lam Name Exp  -- Abstraction
         | App Exp Exp   -- Application
         deriving (Show)

app :: Exp → Exp → Exp
app = App

lam :: (Exp → Exp) → Exp
lam f = Lam n body
  where
    body = f (Var n)
    n    = prime (maxBV body)

type Name = Integer

bot   :: Name
prime :: Name → Name
(⊔)   :: Name → Name → Name

bot   = 0
prime = succ
(⊔)   = max

maxBV :: Exp → Name
maxBV (Var _)   = bot
maxBV (App f a) = maxBV f ⊔ maxBV a
maxBV (Lam n _) = n
```

Figure 4: Final solution

Although this version does avoid capturing, the whole approach is a bit fragile, and the need for renaming makes it quite inefficient.

5. Conclusion

We have presented a simple solution to the problem of generating first-order syntax with binders from a higher-order interface. The key is to use circular programming to be able to examine the body of a binding "before" deciding which name to bind. Despite its simplicity, our solution possesses characteristics that makes it quite suitable for practical EDSLs. In particular, our solution

- does not require a separate data type for higher-order abstract syntax,
- is efficient and implementable in plain Haskell 98.

Our technique also serves as a real example where circular programming is used to change the complexity class of an algorithm. Bird's circular solution to the Repmin problem uses one traversal instead of two, possibly leading to a smaller constant factor for the algorithmic complexity. In our case, circular programming allows us to build an expression in linear time instead of exponential.

The complete final solution is given in Fig. 4.

Acknowledgments

This work has been partially funded by the Swedish Foundation for Strategic Research, under grant RAWFP. We would like to thank Anders Persson, Jean-Philippe Bernady, Joel Svensson, Josef Svenningsson, Mary Sheeran and Michał Pałka for useful input on this pearl.

References

[1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar. In *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer Berlin Heidelberg, 2011.

[2] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 174–184. ACM, 1998.

[4] V. Capretta and A. P. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 63–77. Springer Berlin Heidelberg, 2007.

[5] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Advances in Computing Science, ASIAN'99*, volume 1742 of *LNCS*, pages 62–73. Springer, 1999.

[6] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003.

[7] A. Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128. ACM, 2009.

[8] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78. ACM, 2010.

[9] C. McBride. I am not a number, I am a classy hack. 2010. URL http://www.e-pig.org/epilogue/?p=773.

[10] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. Accepted for publication at ICFP 2013.

[11] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208. ACM, 1988.

[12] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Implementation and Application of Functional Languages*, volume 5836 of *LNCS*, pages 156–173. Springer Berlin Heidelberg, 2011.