Parametric Higher-Order Abstract Syntax in Agda: A Case Study

Renate Eilers
Department of Information and Computing Sciences, Utrecht University
r.r.eilers@students.uu.nl

1 Introduction

One common application of computer science is to solve real world problems through computation. Modeling such a problem as a computational one requires a way of representing the real world domain inside a computer. One approach to accomplishing this is to use a general purpose language for modeling, but it is often the case that problems can be explored more easily in a language created specifically for the given domain: dedicated types and functions can be easier to interpret and transform than the laborious constructions sometimes needed to represent the same structure in a general purpose language.

A single-purpose language is called a *domain-specific language* (DSL), and when it comes to DSLs there are generally two approaches: one can either design a dedicated compiler for it, or choose to embed it into a host language. The second option has the benefit of letting the programmer reuse the host language's features, such as its type checker and syntax. It is because of this advantage that a lot of research has been done in the area of *embedded DSLs* (EDSLs).

Generally, an EDSL is represented by an *abstract syntax tree* (AST) in some host language. Unfortunately, sharing is not always straightforward in these structures. Generally, some representation of an environment must be made explicit to enable sharing. The downside of this is that programmers working with the language have to concern themselves with labeling or other measures to ensure well-scopedness[OL13]. This calls for cumbersome, low level work and distracts from the actual problem.

From the issues of sharing within ASTs (among other things) the notion of *higher-order abstract syntax* (HOAS) arose[PE88]. HOAS is a technique that uses higher order elements inside ASTs to represent sharing: rather than using an explicit representation, terms in an EDSL can be represented by variables in the host language. The programmer does not have to worry about the particulars of scoping and explicit environments any longer, as this is solved by the host language's native constructs.

But while HOAS solves one set of problems, it comes with complications of its own. HOAS terms can encompass too wide a function space —*any* function mapping an expressions to an expression will do, mala fide pattern matches included— and even worse: they can produce non-terminating definitions. This is a problem for languages—such as Agda— that require strict positivity. Fortunately, there is an alternative that retains most of the advantages that HOAS offers: *parametric HOAS* (PHOAS). It differs from HOAS by extending terms with a polymorphic type parameter. Rather than binding terms, PHOAS binds variables of this polymorphic type. By parametricity the variables can not be inspected or used for anything other than being pushing around[WW], solving both of the objections to HOAS mentioned above.

As PHOAS appears to offer a promising solution to the problem of sharing in DSLs, this work[1] aims to discover some of the benefits and difficulties of using PHOAS in the dependently-typed programming language and theorem prover Agda[Nor09]. It shows how to translate a simple DSL for arithmetic expressions into PHOAS terms, and demonstrates techniques for proving basic properties of this DSL. Finally, it records some unsuccessful attempts at verifying a compiler for PHOAS terms. In failing to prove compiler correctness, this work demonstrates some of the limits of using PHOAS in Agda.

2 Related work

Adam Chlipala has researched the use of PHOAS for DSLs in Coq[Chl08], and successfully created a library for the automatic construction proofs on PHOAS terms. In [Chl10] he uses this library, Lambda Tamer[2] for the automatic verification of a compiler. Since this work relies on Coq's highly customizable tactics for proof automation a direct translation to Agda is not possible.

More recently Oliveira and Löh have worked on

[1] All code for this project can be found at https://github.com/RenateEilers/Agda-PHOAS

[2] Free to download at http://ltamer.sourceforge.net/

the implementation of a PHOAS-based DSL[OL13] in Haskell. As Haskell does not support full-blown dependent types, their work focuses on how to ensure that PHOAS terms are well-typed. This is achieved by using *typed lists*; a "generalization of heterogeneous lists of statically known length". Fortunately, well-typedness is something we get more-or-less for free in Agda.

In [WW], Washburn and Weirich note the difficulty of folding over PHOAS terms, and provide a basic Haskell library to tackle this issue. They have implemented expressions as recursive types, which are more abstract and more complex than the expressions in our implementation. Our work follows Oliveira and Löh's design because it is easier to comprehend and reason about. The downside of this approach is that makes generalized functions such as folds difficult to implement.

3 A simple DSL and sharing

To demonstrate the problems with sharing, we introduce a simple DSL for representing arithmetic expressions. Initially, this language consists only of values and binary addition. In Agda, such a language could look like this:

```
1  data Expr : Set where
2    Val : Nat  → Expr
3    Add : Expr  → Expr  → Expr
```

The data constructor `Val` lifts natural numbers to `Expr`s, and `Add` represents the binary addition of two expressions. Below we have defined a function to evaluate expressions, `eval`, and one to generate binary trees of a given depth. This second function, `tree`, is implemented using Agda's native binding construct `let`. Both functions are direct translations of the work done in Haskell by Oliveira and Löh in [OL13].

```
4  eval : Expr  → Nat
5  eval (Val x)     = x
6  eval (Add e₁ e₂) = eval e₁ + eval e₂
7
8  tree : Nat  → Expr
9  tree Zero     = Val (Succ Zero)
10 tree (Succ n) = let s = tree n in Add s s
```

The problem with sharing in DSLs is exposed when we consider, for instance, the evaluation of `tree (Succ (Succ Zero))` below, as in figure 1.

This breakdown demonstrates how traversing a tree destroys sharing. Identical subterms that were supposed to be shared are all evaluated individually, resulting in an exponential time computation.

To speed things up, we need a way to make sharing explicit within the DSL. As stated earlier, the work of defining a system for sharing is time-consuming and error-prone. It means spending time on lower-level work, stealing focus from the actual problem. Hence, it is preferable to let Agda's native binding mechanism handle the work.

This is where HOAS comes in. A first attempt has us make sharing explicit by adding a `Let` constructor to the previously designed language. We can imagine it looking something like this:

```
1  data Expr : Set where
2    Val : Nat  → Expr
3    Add : Expr  → Expr  → Expr
4    Let : Expr  → (Expr  → Expr)  → Expr
```

The `Let` constructor above takes two arguments: the expression that is to be shared, and a function from one expression to another, which, when semantics are added, we pass the expression that is to be shared as an argument. A type declaration such as the above causes no problems in most regular functional languages, but for various theorem provers, Agda amongst them, the negative occurrence of `Expr` in the second argument is a no-go. As such, HOAS as defined above can not be implemented in Agda.

In 2008 Adam Chlipala introduced a technique to solve this problem: PHOAS[Chl08]. To convert the previously defined expression language to PHOAS we have to parameterize its data type. Now, instead of expressions, the `Let` constructor's function argument binds variables of the parameter type. For the purpose of lifting objects of the parameter type to expressions an additional constructor is required. This is the function of `Var` in the code below:

```
1  data Expr (a : Set) : Set where
2    Val : Nat  → Expr a
3    Add : Expr a  → Expr a  → Expr a
4    Var : a  → Expr a
5    Let : Expr a  → (a  → Expr a)  → Expr a
```

By requiring that the type parameter is polymorphic we can represent closed expressions. By parametricity the type argument can only be used in an abstact way, and we will not be able to construct malformed terms like `Let (Val Zero) (λx → Var (Succ Zero))`. Closed expressions are defined below:

```
6  ClosedExpr : Set₁
7  ClosedExpr =  ∀ {a : Set}  → Expr a
```

To get a feel of what PHOAS terms look like, let us return to the `tree` function we defined earlier. A translation of this function to one that produces PHOAS terms would look like this:

```
8  tree : Nat  → ClosedExpr
9  tree Zero = Val Zero
10 tree (Succ n) = Let (tree n)
11    (λ s  → Add (Var s) (Var s))
```

Below it is shown what the `eval` function would look like for PHOAS terms. Note that this function works on `Expr Nat`s rather than on `ClosedExpr`s. The parameter type is specialized to whatever result type we want. This is something we shall see in a lot of functions working on PHOAS terms.

Figure 1: Stepwise evaluation of `eval` (`Tree` (`Succ` (`Succ Zero`)))

```
  eval (tree (Succ (Succ Zero)))
= eval (let s =  tree (Succ Zero) in Add s s)
= let s = (tree (Succ Zero)) in eval s + eval s
= let s = (let s' = tree Zero in Add s' s') in eval s + eval s
= let s' = tree Zero in (eval s' + eval s') + (eval s' + eval s')
= let s' = Val (Succ Zero) in (eval s' + eval s') + (eval s' + eval s')
= (Val (Succ Zero) + Val (Succ Zero)) + (Val (Succ Zero) + Val (Succ Zero))
= Succ (Succ (Succ (Succ Zero)))
```

Figure 2: `inline` function for PHOAS terms

```
17 inline : {a : Set} → Expr (Expr a) → Expr a
18 inline (Val i) = Val i
19 inline (Add e₁ e₂) =
20   Add (inline e₁) (inline e₂)
21 inline (Var x) = x
22 inline (Let e e) = inline (e₂ (inline e₁))
```

Also notable is how the case for `Let` e_1 e_2 is evaluated. The shared expression is evaluated once, and the result is propagated using Agda's own lambda binding. Just what we wanted!

```
12 eval : Expr Nat → Nat
13 eval (Val x) = x
14 eval (Add e₁ e₂) = eval e₁ + eval e₂
15 eval (Var x) = x
16 eval (Let e₁ e₂) = eval (e₂ (eval e₁))
```

4 Proof techniques for PHOAS

By using PHOAS it appears that we have solved the problem of sharing in DSLs, and while PHOAS terms look a little different from what we started with, otherwise simple functions such as `eval` are still easily implemented.

Unfortunately, working with PHOAS is not as straightforward as one might initially expect. This will become clear as we try our hand at some proofs. What makes proofs on PHOAS terms difficult is the fact that terms (`ClosedExpr`s) are actually functions. Agda has no mechanisms supporting induction on function types. This notion is best explained using an example. Consider a function that removes sharing for a given term. We will call this function `inline`. It is defined in figure 2.

It seems sensible that evaluating a given term yields the same result as evaluating the same term with sharing removed, so let us try and prove it. That is, we would like to prove for all terms `e` that `eval e` ≡ `eval (inline e)`.

Generally, the way to go about such a proof is by induction on the structure of the term `e`. By showing that the stated property holds for every possible constructor of `e`, we show that is must hold for all possible expressions. But this turns out to be problematic, because, as we stated above, `e` is actually a *function*.

To understand this, we have to take a closer look at the types occurring in the above. In what we aim to prove, `e` is applied to two functions: on the left hand side of the equation it is applied to `eval`, which expects an `Expr Nat`. On the right hand side, we apply `e` to `inline`, which expects an `Expr (Expr Nat)`. Agda automatically solves these implicit arguments for us, and so even though we have not written it down explicitly, we are actually trying to prove that `eval (e Nat)` ≡ `eval (inline (e Expr Nat))`. Instead induction on `e`, what we actually need is induction on `e {Nat}` and `e {Expr Nat}` in parallel. Agda offers no built-in mechanisms for this, but it turns out we can approach this notion by construction of ourselves.

Parallel induction is realized through the definition of a data type representing the equivalence of two expressions. It can be found in figure 3. This approach is a generalization of the mechanism described by Chlipala in [Chl08], which was used to tackle a very similar problem encountered in a more specific application of PHOAS in Coq.

The data type that we have defined in the current work has constructors matching every one of the `Expr A` date type. In addition to being parameterized over both expression's parameter types, it abstracts over relations on these types. This allows a user to construct custom relations for different proofs and work with whatever is most suitable for the problem under consideration. The way to go about defining the proper relation should become clear when we go through an example later. First, however, let us walk through the definition of the data type. As with the `Expr A` data type, we distinguish between four cases:

`Val` has a very straightforward definition, stating that two `Val`s are equivalent if they hold the same value.

`Add` states that two `Add`s are equivalent if both their left and right subexpressions are.

`Var` is where we see why we need a relation `P` on the expressions types. Two expressions `Var x` and `Var y` are equal within the context of `P` if both `x` and `y` are in the relation `P`. Without such a relation, there would be no way to relate `x` and `y`. They could be any value of the

Figure 3: Data type representing the equivalence of two expressions

```
data _ ⊢ _≡ _ {A B : Set} : (P : A → B → Set) → Expr A → Expr B → Set₁ where
  Val : ∀ {P} {x : Nat} → P ⊢ (Val x) ≡ (Val x)
  Add : ∀ {P} {l₁ r₁ l₂ r₂} → P ⊢ l₁ ≡ l₂ → P ⊢ r₁ ≡ r₂ → P ⊢ Add l₁ r₁ ≡ Add l₂ r₂
  Var : ∀ {P x y} → P x y → P ⊢ Var x ≡ Var y
  Let : ∀ {P b₁ b₂ e₁ e₂} → P ⊢ b₁ ≡ b₂ → ( ∀ {x₁} {x₂} → (P x₁ x₂ → P ⊢ (e₁ x₁) ≡
(e₂ x₂))) → P ⊢ (Let b₁ e₁) ≡ (Let b₂ e₂)
```

Figure 4: Postulate stating that two instantiations of the same term are equivalent

```
postulate
  e≡e: ∀ {A B P} {e : ClosedExpr} → P ⊢
e {A} ≡ e {B}
```

required type, which would make it very difficult to prove anything concrete about them.

`Let` expressions are equivalent only if several requirements hold. First of all, it should be the case that the bound subexpressions are equivalent. Additionally, it is demanded that for any pair consisting of exactly one element of each expression type, *if* those two elements are in the relation `P`, then it must hold that the expressions resulting from applying the function arguments to these elements should be equal as well.

But how would we construct such an equivalence proof for two given terms? Intuitively, we know that by parametericity any two concrete instantiations of a closed expression `e`, such as for instance `e {Nat}` and `e {Expr Nat}`, must be equivalent. Unfortunately, we have not found a way to prove this in Agda because we can not use induction on functions: as soon as we apply `e` to a type, we lose the connection between the resulting term and `e` itself. For this reason, we add a postulate, which we believe to be justified by the parametricity of closed expressions. The postulate states that two separate instantiations of one term are equivalent. It can be found in figure 4.

Now, we can simply call on the postulate to supply us with a proof of the equivalence of the separate instantiations of a term. Induction on the structure of this proof allows us to break down the two separate instantiations in parallel.

For an example, let us try to construct a proof for the property stated earlier, saying that the removal of sharing does not change the value of a term. This proof should be demonstrative for the techniques associated with proofs on PHOAS terms.

For the running example, the expression types are `Nat` and `Expr Nat`. In addition to calling the postulate with these parameters, we have to supply a relation on them as well. This relation is the only thing we can use to relate two variables in our proof. Keeping in mind that we will have to prove for the `Var` case that `Var x ≡ Var (inline y)`, we come up with the following:

```
data _ ≃ _ : Nat → Expr Nat → Set where
  Eq : ∀ {n e} → n ≡ eval e → n ≃ e
```

This relation is sufficient to construct a complete proof, which can be found in figure 5.

The function `inlineEquality` that we can find there is actually a wrapper function, which instantiates the postulate and the closed term to the correct types. With these, a function called `inlineEquality'` is called. This is where we find the actual proof. Most proofs on PHOAS terms rely on such a wrapper function: that which we want proven involves a closed term, but the proof relies on induction on the shape of the term. As stated earlier, induction can not be applied to closed terms directly in Agda, because they are actually functions.

In the proof of `inlineEquality'`, the `Val` branch is trivially given by reflexivity because there is no sharing within values. For the `Add` branch we use recursive calls of the theorem on both subexpressions to show that they evaluate to the same value, and we throw in an additional lemma stating that if two pairs of integers are equal, their sums will be equal. The `Var` branch looks almost like cheating in its simplicity— we simply use the relation between `Nat`s and `Expr Nat`s that we defined above, which gives us exactly what we wanted to prove. The `Let` branch is the most interesting. We have two expressions `Let` $\mathtt{x}_1$ $\mathtt{e}_1$ and `Let` $\mathtt{x}_2$ $\mathtt{e}_2$, a proof $\mathtt{p}_1$ saying that $\mathtt{x}_1$ and $\mathtt{x}_2$ are equivalent and a proof $\mathtt{p}_2$ saying that for all pairs of `Nat` × `Expr Nat`, if they are in ≃, then applying the function arguments in the `Let` expressions to them will result in equivalent expressions. This proof is then instantiated on two specific values: `eval` $\mathtt{x}_1$ and `inline` $\mathtt{x}_2$, resulting in a function `q` which, when given a proof that `eval` $\mathtt{x}_1 \simeq$ `inline` $\mathtt{x}_2$, will return a proof of $\_\simeq\_ \vdash(\mathtt{e}_1\mathtt{x}_1) \equiv (\mathtt{e}_2\mathtt{x}_2)$. Fortunately, the argument that `q` requires is easily constructed by recursively calling `inlineEquality'` on $\mathtt{p}_1$. This concludes the proof of `eval e ≡ eval (inline e)`.

In general, the way to go about writing proofs involving PHOAS terms is to first consider what relation is needed to relate the `Var` terms. The next step is to write a wrapper function stating

Figure 5: Proof that the removal of sharing in a PHOAS term does not change its value

```
inlineEquality : {e : ClosedExpr} → eval e ≡ eval (inline e)
inlineEquality {e} = inlineEquality' ( e≡e{Nat} {Expr Nat} {_≃_} {e})

inlineEquality' : ∀ {eA : Expr Nat} {eB : Expr (Expr Nat)} → _≃_ ⊢ eA ≡ eB → (eval eA) ≡
(eval (inline eB))
inlineEquality' Val = refl
inlineEquality' (Add p₁ p₂) = lem (inlineEquality' p₁) (inlineEquality' p₂)
  where lem : {a b a' b' : Nat} → a ≡ a' → b ≡ b' → a + b ≡ a' + b'
        lem refl refl = refl
inlineEquality' (Var (Eq x)) = x
inlineEquality' {Let x₁ e₁} {Let x₂ e₂} (Let p₁ p₂) with (p₂ {eval x₁} {inline (x₂)})
...  | q = inlineEquality' (q (Eq (inlineEquality' p₁)))
```

the property at hand in terms of closed expressions. From here, a function specialized to the relevant expression types can be called and supplied with an equivalence proof initialized on the relation needed to prove the `Var` case. Using this approach, various properties on PHOAS terms can be proven. For example, the code base provided with this work contains a proof showing that constant folding does not change the value of an expression. The next section demonstrates some attempts at a more intricate proof on the correctness of a compiler for PHOAS terms.

5 Compiler correctness

This section showcases the efforts made to construct a provably correct compiler for the expression language into instructions for a simple, abstract stack machine implemented in Agda. While the constructed definitions for the functions and the data types central to this aim seem sensible, none of the approaches that were explored have led to a correctness proof for the compiler. In fact, they have given rive to the suspicion that such a proof can not exist for PHOAS terms as implemented in this work. Regardless of this unsatisfying result, the efforts shed a light on the nature of PHOAS and should prove insightful to the less obvious problems that arise from using this technique. Hence, after delving into the design of the stack machine and the compiler this section continues by showing the attempts at proving compiler correctness, thereby revealing the problems prohibiting it.

Our aim is to be able to compile expressions to a basic set of executable instructions operating on stacks (our implementation of stacks can be found in figure 9). These instructions, five in total, are represented by the `Op` data type which can be found in figure 6. Their design was inspired by the work of James Mckinna and Joel Wright in [MW06], which implements a provably correct expression compiler for a simple expression language in the dependently-typed programming language Epigram.

A minor difference is that, in contrast to the work by Mckinna and Wright, the expression language described in the current work does not support booleans. As a result, any boolean-related operations have been left out here.

More notable is the fact that the PHOAS language supports variable binding, which the language defined by Wright and Mckinna does not. Hence the `Op` data type must provide functionality for the storing and loading of variables.

The operation data type described in the work by Mckinna and Wright is indexed by two numbers, signifying the effect that performing the associated operation would have on a stack. The first number represents the size of the stack before performing the operation. It is used to prevent operations being performed on stacks of insufficient size. An example of this would be attempting to perform bindary addition on a single-element stack. By demanding addition is only performed on stacks containing at least two elements, such errors can be prevented.

The second number signifies the size of the stack upon performing the operation. The `Op` data type that is used in the current work is indexed by an additional number to represent the heap size, which is where variables are stored. This index is used to prevent users from looking up variables that have not yet been stored.

Below, the function of each operation is explained briefly. The semantics given below are implemented by the `exec` function (see figure 6). For every operation and pair of stacks, the second of which represents the variable heap, `exec` returns the updated stack pair that is the result of performing that operation on the pair that was supplied.

`Stop` signifies a non-operation, and simply returns the current stack. It is used to end chained operations.

`Push` takes a natural number and pushes it on the stack.

`Add` requires that the current stack size is at least two, removes the two topmost elements and

Figure 6: Data type representing operations and a function showing the semantics of executing an operation on a pair of stacks

```
data Op : Nat → Nat → Nat → Set where
  Stop  : ∀ {n m} → Op m n n
  Push  : ∀ {n m v} → Nat → Op v (Succ n) m → Op v n m
  Add   : ∀ {n m v} → Op v (Succ n) m → Op v (Succ (Succ n)) m
  Load  : ∀ {n m v} → Fin v → Op v (Succ n) m → Op v n m
  Store : ∀ {n m v} → Fin v → Op v n m → Op v (Succ n) m

exec : ∀ {n m v } → Op v n m → Stack n × Stack v → Stack m × Stack v
exec Stop s = s
exec (Push x o) (x₁ , x₂ = exec o ((x ▷ x₁) , x₂)
exec (Add o) ((x ▷ (x₁ ▷ x₂)) , x3) = exec o (((x + x₁ ▷ x₂) , x3)
exec (Load x o) (x₁ , x₂) = exec o (((lookup x x₂) ▷ x₁ ), x₂ )
exec (Store x o) ((x₁ ▷ x₂) , x) = exec o (x₂ , update x x₁ x3)
```

Figure 7: The `compile` function

```
compile : ∀ {v n} → Expr (Fin (Succ v)) → Op (Succ v) n (Succ n)
compile {v} e = compile' (fromNat v) e

compile' : ∀ {v n} (t : Fin v) → Expr (Fin v) → Op v n (Succ n)
compile' t (Val m)     = Push m Stop
compile' t (Add e₁ e₂) = compile' t e₁ ++ (compile' t e₂ + + Add Stop)
compile' t (Var i)     = Load i Stop
compile' t (Let x e)   = compile' t x ++ Store t (compile' (predFin t) (e t))
```

Figure 8: The `Fin` data type

```
data Fin : Nat → Set where
  FZero : ∀ {n} → Fin (Succ n)
  FSucc : ∀ {n} → Fin n → Fin (Succ n)
```

Figure 9: The `Stack` data type

```
data Stack : Nat → Set where
  Empty : Stack Zero
  _ ▷ _ : {n : Nat} → Nat → Stack n →
Stack (Succ n)
```

pushes their sum back on the stack.

`Load` takes a number no higher than the heap size, represented by an element of the `Fin v` data type, where `v` is bound by the operation's first index. The `Fin v` data type represents a set containing exactly `v` elements, and is used here to make sure we will not try to access a heap element that is out of bounds. The definition of `Fin v` can be found in figure 8. `Load` looks up the heap element at the given location, and pushes it on the current stack.

`Store` takes a heap address, also represented by an element of `Fin v`, removes the topmost value from the current stack and saves it to the given address.

In addition to any arguments mentioned in the above, all operations but `Stop` receive an operation argument as well. This allows the user to chain operations together into longer sequences. Alternatively, operations can be concatenated using the function ++, defined in the code base.

As an example, consider an operation for giving the sum of the natural numbers one and two. This could be represented as `Push (Succ Zero)(Push (Succ (Succ Zero))(Add Stop))`, or alternatively, using ++, as `Push (Succ Zero)Stop ++ (Push (Succ (Succ Zero))Stop ++ Add Stop)`.

With the definitions for operations in place, we can construct a compiler function that translates expressions to operations. The result is found in figure 7. Here, we see that the function `compile` takes an `Expr (Fin (Succ v))` and returns an `Op (Succ v) n (Succ n)` for any n. Interesting to note here is that the expression type is `Fin (Succ v)` rather than just `Fin v`. The reason for this is that `compile` is a wrapper function. It calls the function `compile'`, and supplies it with the number `v` converted to a `Fin (Succ v)` by the function `fromNat`. `fromNat` takes a natural number `n` and returns a `Fin (Succ n)`— not a `Fin n` because `n` could be any natural number, zero included, and there is no way of creating a `Fin Zero`. Hence, the resulting `Fin` type has to be incremented by one. This `Fin (Succ v)` functions as an address for storing variables on the heap. It is decremented with every binding term that is encountered, thus ensuring that no two variables will be stored in the same place.

The design of the `compile'` function should not be very surprising, bar maybe that of the more in-

tricate `Let` branch. It works as follows: first, the shared expression is compiled and stored on the location of the `Fin v` argument. A new expression is created by applying the `Let`'s function argument to this same `Fin v`, ensuring that all bound variables refer to this heap location. The expression resulting from this function application is then compiled by the `compile`' function, taking not the `Fin v` element but its predecessor, calculated by the `predFin` function. This prevents variables in nested expressions from being stored in the same location.

Testing the compiler shows us that it works as expected as long as the supplied `v` is at least as great as the number of distinct variables in the expression (if not, we would end up repeatedly storing values to the zero index, which could lead to incorrect results).

Ideally, the compiler's correctness would be shown by the existence of a proof in Agda, but unfortunately we have not succeed in constructing such a proof. In the following, we demonstrate two failed approaches at proving compiler correctness to help us see where the pitfalls lie.

First, let us consider what it is that we want to prove exactly. The compiler is considered correct if executing the instructions it generates results in the same value as evaluating our expression would. As stated above, the compiler can fail to do so if the heap is too small to hold all variables. With all this in mind, we come up with the following goal: `∀ {e : ClosedExpr} {n : Nat} {s : Stack n} {h : Stack (Succ (#vars e))} → (fst (exec (compile e)(s , h))) ≡ eval e ▷ s`. Informally: compiling an expression and executing the result on some stack of size n and some heap the size of the number of variables in the given expression plus one *should* give us the same result as evaluating said expression.

As mentioned in section 4, we can not build inductive proofs from `ClosedExpr`s directly: they have to be instantiated to the desired types. Such instances can be shown to be equivalent by the postulate `e≡e`. Since we are working with `Expr (Fin (Succ v))`s for `compile` and `Expr Nat`s for `eval`, we have to come up with a relation on these types that can help proving the `Var` case. When this relation is defined the postulate can be instantiated and used for constructing an inductive correctness proof. The design of this relation is where our two approaches to the proof differ.

Let `eN` and `eF` be the `Nat` and `Fin (Succ v)` instances of a `ClosedExpr`, respectively. The aim is to prove something along the lines of `??? ⊢ eF ≡ eN → fst (exec (compile eF)(s , h)) ≡ eval eN ▷ s`, where ??? should be some relation on the target types. In case of the `Var` branch, we would have to prove that `fst (exec (compile (Var xF))(s , h)) ≡ eval (Var xN)` which, after some rewriting, translates to `lookup xF h ≡ xN`. What kind of relation on `Nat`s and `Fin`s could help build this proof?

Figure 10: A relation on `Fin`s and `Nat`s

```
70 data _≏_ : ∀ {n} → Fin n → Nat →
   Set where
71   Eq : ∀ {n m} {f : Fin n} →
   ( ∀ {h : Stack n} → lookup f h ≡ m) → f ≏ m
```

Figure 11: Another relation on `Fin`s and `Nat`s

```
72 _≗_ : ∀ {n} → {s : Stack n} → Fin n →
   Nat → Set
73 _≗_ {n} {s} f m = lookup f s ≡ m
```

Our first attempt introduces a very lenient relation on the target types. Let this relation, defined in figure 10, be called `≏`. It states that from some `f : Fin n` and `m : Nat`, `f≏m` if it holds for all heaps `h` that `lookup f h ≡ m`.

With this relation, the case for `Var`s is easily proven. `Val` is straightforward as well, and `Add`, even though it requires some shuffling around of operations for the recursive calls to work out, is manageable as well. The problems arise when we try to prove the `Let` case. Remember that this branch of our equivalence data type looks like this: `∀ {P b₁ b₂ e₁ e₂} → P ⊢ b₁ ≡ b₂ → ( ∀ {x₁} {x₂} → (P x₁x₂ → P ⊢ (e₁x₁) ≡ (e₂x₂))) → P ⊢ (Let b₁e₁) ≡ (Let b₂e₂)`.

Initialized on the required types it lends us the following building blocks for our proof: `_≏_ ⊢ bF ≡ bN` and `∀ {xF} {xN} → (xF≏xN → _≏_ ⊢ (eF xF) ≡ (eN xN))`. The first of these is easily plugged into a recursive call; the second not so much. We need the equivalence of the applied function terms in order to show by a recursive call that the compiler is correct for both subterms. If we have that, both parts can be glued together into a correctness proof for the whole term. But in order to obtain that equivalence proof, we must first supply an argument saying that `xF≏xN`. This requires us to prove that for *all* heaps `h` and some `f` and `m`, `lookup f h ≡ m`. Needless to say, no such proof exists. The relation we defined, though helpful for the `Var` instances, was too strong. Disappointed, but not discouraged, we return to the drawing table for a second attempt.

This time we try to construct a weaker relation on the expression types. Instead of letting the heap argument be universally quantified, this relation concerns one that is set. It is defined by the function `_≗_` in figure 11.

In order for this relation to be of any help in the `Var` branch, the heap we supply it with should be the same as the one the compiled ex-

pression is executed on, because that is where it will do the lookup. Filling in the new relation and its implicit arguments we get the following proof goal: `(_≈_ {Succ v} {h}) ⊢ eF ≡ eN → fst (exec (compile eF)(s , h)) ≡ eval eN ▷ s`. Unsurprisingly, `Val` and `Var` remain easy to prove by reflexivity and calling on the relation respectively. But this time, we run into trouble while attempting to prove the `Add` branch.

Compiling an `Add` expression translates to compiling both subexpressions, chaining the resulting operations and appending an add operation. *Executing* this operation means that first the operation that results from compiling the left subterm is executed on some stack and heap. Then, the operation corresponding the right subterm is executed on the stack and heap that result from the previous execution. From the stack and heap that result from *this* execution the two topmost results are added and pushed back on the stack. To *evaluate* an `Add` expression, both subexpressions are evaluated and then added together. Thus, if we could show by recursively applying the correctness proof to both subterms we would be done (a lemma showing that the sum of two equals produces an equal is easily constructed). Unfortunately, these recursive proofs turns out to be problematic: a subterm may contain one or more binding terms. Compiling end executing such a term affects the heap, which means that the operation resulting from compiling the rightmost subterm will execute on a heap that is potentially different from the one that the relation is initialized on. For this reason we can not use the correctness proof obtained recursively from the right branch of our equivalence proof— it's initialized on a relation with a set heap. Additionally, there is no way to construct the required equivalence proof with a different relation on a potentially changed heap. The relation that was defined was too weak, and now we are stuck.

Unfortunately, there is no clear solution to this predicament: some relation on heaps, `Fin`s and `Nat`s is required for finishing the `Var` branch, but universal quantification on heaps is too strong and asks us to prove the impossible, while existential quantification is too rigid and can not deal with the evolution of heaps during execution. This leaves us to conclude that proving compiler correctness on the data types and functions as defined in this work, is not directly possible.

6 Conclusion

Section 3 showed that binding in DSLs is not always straightforward to implement. Shallow embeddings tend to lose the property of sharing when traversed, and deeper embeddings may require intricate constructions to keep track of the environment. For defining simple expressions or basic functions and proofs on such terms, PHOAS has shown itself to be a good solution to these issues. In section 3 we saw that it takes very little work to define a readable construction for let bindings using PHOAS. In section 4 it was demonstrated that simple proofs and functions on PHOAS terms, such as `eval` and `inline`, do not require very complex proof techniques. Most basic proofs tend to follow the exact same structure, making the author suspect that the idea of some form of proof automation is not far-fetched. To conclude: for simple types of programs, PHOAS proves itself to be a quick and elegant solution.

However, as soon as soon as functions turn more complex, working with PHOAS terms becomes challenging. As soon as we want to build proofs over expressions of different types, we need to supply an equivalence proof on these terms. This requires us to come up with a relation between the target types. Thinking of this is not necessarily trivial, and it is probably not automatable. As we can see from `inlineEquality`, figure 5, more complex functions deviate in structure from the basics, as they may require numerous additional lemmas.

Still, for certain problems the proofs remain small and elegant. The real issues with PHOAS surfaced in section 5, when we attempted to verify a compiler that was constructed there in figure 7. The necessity of a dedicated data type to allow for parallel induction on two different expression types prohibited us from completing the proof. This data type is too rigid for certain types of proofs, thereby limiting the PHOAS's abilities.

The above leads us to conclude that PHOAS, as implemented in this work, is applicable in Agda as a quick and elegant fix for the binding problem in DSLs if the work at hand remains limited to simple functions and proofs, preferably consisting of expressions of a single type. For more intricate purposes, alternative binding constructs may be preferable or even necessary.

7 Future work

The current work relies heavily on the postulate `e==e`, stating that two instances of a closed expression are equivalent. This postulate seems sensible enough, but we have not been able to prove it yet. Since it is vital to a number of proofs, we should look into ways to prove it to make the work more complete.

Additionally, it would be interesting to see what making the object language more complex would do to the complexity of proofs: would adding more constructors and supporting types make PHOAS terms much harder to work with? To study the scalability of this approach it would be interesting to see how expanding the language would affect the proofs.

Another topic for future research concerns abstraction. Currently, a number of basic functions follow the exact same structure, differing only in the body of the computation that takes place. It might be possible to be more efficient in this. As speculated earlier, automation may be within reach. Alternatively, we could consider implementing ideas from [Kme13]. Here, it is shown that PHOAS terms can be seen as profunctors. Implementing expressions as such would allow users to map over terms. This is closely related to the work done by Washburn and Weirich[WW], mentioned in section 2.

Finally, since we were unable to verify our compiler, the question whether it is actually correct is still open. It would be interesting to see if other approaches lead to success. One possible path to take here would be to compile to an intermediate language representing the same semantics, such as for instance one using De Bruijn indices, and prove correctness by transitivity.

References

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. *SIGPLAN Not.*, 45(1):93–106, January 2010.

[Kme13] Edward Kmett. Phoas for free, 2013. [Online; accessed 15-03-2017].

[MW06] James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler. In *in Epigram. Submitted to the Journal of Functional Programming*, 2006.

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[OL13] Bruno C. d. S. Oliveira and Andres Löh. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 87–96, New York, NY, USA, 2013. ACM.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. 23(7):199–208, July 1988.

[WW] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higherorder abstract syntax with parametric polymorphism. *J. Funct. Program*, page 2008.