Normalization-by-evaluation and Metaprogramming with PHOAS

ELEFTHERIOS IOANNIDIS, PHD STUDENT, ACM: 4333776, University of Pennsylvania, Advised by Steve Zdancewic

Parametric Higher-Order Abstract Syntax (PHOAS) skips the binder bureaucracy and leverages the meta-language's variable capturing facilities in the object-language. Using the Coq proof-assistant, we take the PHOAS approach to the next logical conclusion and show how to use the meta-language's evaluation mechanism for object-language Normalization-by-evaluation. We also show how the polymorphic nature of PHOAS allows us to do typed metaprogramming with an infinite a tower of embedded languages, while maintaining type-safety across levels. Those novel uses of PHOAS inspire hope that there are more promising applications to be discovered.

ACM Reference Format:
Eleftherios Ioannidis, PhD student, ACM: 4333776. 2022. Normalization-by-evaluation and Metaprogramming with PHOAS. 1, 1, Article 1 (November 2022), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

PROBLEM STATEMENT

Compiler verification is a fundamental problem in formal methods. For any computer scientist a bug in the compiler leads to arbitrary program behavior, will require arduous debugging and a deep understanding of the compiler's inner workings to solve. Formally verified compilers inspire confidence and a solid foundation for creating new software.

There have been several successful compiler verification projects; the CompCert certified C compiler [7], the CertiCoq Gallina compiler [1] and more [6][9][3]. This work focuses on verified compilers for the lambda-calculus family. Specifically, we will show how Parametric Higher-Order Abstract Syntax (PHOAS) [2] can accelerate the development of verified functional compilers in more ways than originally intended. The reader can consult the original PHOAS paper by Adam Chlipala on how PHOAS can achieve that [2]. In this work we present two novel PHOAS applications

(1) Normalization of higher-order terms
(2) Quasiquotation and Metaprogramming

Readers should refresh their memory on PHOAS syntax and denotational semantics for the Simply-Typed Lambda Calculus ($\lambda^{\rightarrow}$)in Coq by reading Appendix A.

Higher-order terms

A functional compiler with a first-order product language like assembly must flatten higher-order terms to a first-order series of instructions. The usual approaches are Closure-conversion (CC) [8], Defunctionalization [4] or Normalization-by-evaluation (Nbe) [5]. Maintaining a HOAS syntax between CC and Defunctionalization transformations is difficult or impossible, due to the introduction of evaluation contexts which erase dependent types. Normalization-by-evaluation (Nbe) provides an attractive alternative.

In nominal forms of $\lambda^{\rightarrow}$ with explicit binders Nbe cannot be implemented in a single AST pass. Beta-reductions might produce additional beta-redexes. Roughly, one would need to traverse the AST at least as many times as the longest sequence of nested lambda abstractions. Verified

Author's address: Eleftherios Ioannidis, PhD student, ACM: 4333776, elefthei@seas.upenn.eduUniversity of Pennsylvania, Advised by Steve Zdancewic.

2022. XXXX-XXXX/2022/11-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

implementations of Nbe in nominal forms often use *fuel*, then prove an upper bound on the fuel to establish termination. That is not the case in HOAS, Danvy et al. [5] gave a two-line implementation of Nbe based on reify/reflect using typeclasses for structural induction on the type of terms. Their approach is promising and with a few adjustments to deal with the positivity restriction, results in the Nbe instances in fig. 1.

```
Class Nbe (t: type) := {
  reify: typeDenote t → Term typeDenote t;
  reflect: Term typeDenote t → typeDenote t
}.
Instance Nbe_lam {a b: type}`{ Nbe a}`{ Nbe b}: Nbe <{{ a → b }}> := {
  reify v := LAM (fun x ⇒ reify (v (reflect (VAR x))));
  reflect e := fun x ⇒ reflect (APP e (reify x))
}.
Instance Nbe_int : Nbe <{{ Num }}> := {
  reify v := NUM v;
  reflect v := termDenote v;
}.
Definition normalize {t: type} `{ Nbe t}( e: Term typeDenote t): Term typeDenote t :=
  @reify t _ (@reflect t _ e).
```

Fig. 1. The reify-reflect normalization by structural induction on types

The reify-reflect methods define an isomorphism from $\lambda^{\rightarrow}$ terms to Gallina terms that is strongly normalizing. Reflection tranforms object-language terms to meta-language terms and reification does the opposite, taking meta-language terms to the object-language. Gallina's beta-reduction mechanism is invoked when a term is reflected, then reification brings the normal form back to the object-language's domain. Our definition requires no fuel and is small and elegant. The proof of correctness for strong normalization is only a few lines (Appendix B).

Metaprogramming

A common subject in programming languages is metaprogramming, embedding languages within languages. We worked with object-language ($\lambda^{\rightarrow}$) and meta-language (Gallina) until this point, but now we generalize. As $\lambda^{\rightarrow}$ was embedded in Gallina, we will embed an object-language $\lambda_2^{\rightarrow}$ in $\lambda_1^{\rightarrow} = \lambda^{\rightarrow}$ by embedding meta-terms in place of the PHOAS binders. A compositional metaprogramming framework should allow embedding an infinite sequence of languages that each denote to the next one. Defining $\triangleright$ to mean "denotes to the right", for example $\lambda_1^{\rightarrow} \triangleright$ Gallina, we will show it is possible to get

$$\lambda_n^{\rightarrow} \triangleright \dots \lambda_2^{\rightarrow} \triangleright \lambda_1^{\rightarrow} \triangleright \text{Gallina}$$

By using typed PHOAS on each denotation, our use of metaprogramming has to be well-typed in Gallina and does not require a well-formedness predicate.

Let us make some observations. First, the type of terms for $\lambda_1^{\rightarrow}$ is `Term typeDenote t`. Due to polymorphism of `Type` any `type -> Type` function can be used instead of `typeDenote`, including `Term typeDenote: type -> Type`, to get `Term (Term typeDenote) t`, the type of $\lambda_2^{\rightarrow}$ terms which denote to $\lambda_1^{\rightarrow}$. Generalize `termDenote` to the `Denotation` typeclass to implement $\triangleright$, as seen in fig. 2. The base step is `termDenote` which implements ($\lambda_1^{\rightarrow} \triangleright$ Gallina) and the inductive step $\lambda_n^{\rightarrow} \triangleright \lambda_{n-1}^{\rightarrow}$ is `termFlatten`, producing an infinite chain of denotations.

Finally, a demonstration of metaprogramming. Consider the fixpoint function `add1` that bumps all numbers in a language by 1 and the `meta` Ltac annotation that assists with existential variable

```
Class Denotation (v: type → Type) := {
  denote{t}(e: Term v t): v t
                                        }.
Fixpoint termFlatten {t: type} {v: type → Type} (e: Term (Term v) t): Term v t :=
  match e with
  | VAR v ⇒ v
  | NUM f ⇒ NUM f
  | ADD l r ⇒ ADD (termFlatten l) (termFlatten r)
  | APP e1 e2 ⇒ APP (termFlatten e1) (termFlatten e2)
  | LAM e'  ⇒ LAM (fun x ⇒ termFlatten (e' (RET x)))
  end.

#[refine]
Instance baseDenotation: Denotation typeDenote := {}.
intro; exact (termDenote). Defined.
#[refine]
Instance stepDenotation v {Denotation v}: Denotation (Term v) := {}.
intro; exact (termFlatten). Defined.
```

Fig. 2. A chain of denotations defined by structural induction on denotation

instantiation (Appendix C). Calling add1 on $\lambda_3^{\rightarrow}$ only affects the terms of $\lambda_3^{\rightarrow}$ and does not change the terms of $\lambda_1^{\rightarrow}, \lambda_2^{\rightarrow}$. The inverse is also possible, a depth-first approach to modify $\lambda_1^{\rightarrow}$ terms, as well as fine-grained control of which level a transformation will affect.

```
Tactic Notation "meta" uconstr(x) := refine x; exact typeDenote.
Definition l3 :=
  ltac:(meta <{ \_, #1 + (@ (#3 + (@( #1))) }>).
Definition l2 := denote (add1 l2).

Compute add1 l3.                 (* = <{ \_, #2 + @ (#3) + @ (#1) }> *)
Compute l2.                      (* = <{ \_, #2 + (#3 + @ (#1)) }> *)
Compute add1 l2.                 (* = <{ \_, #3 + #4 + @ (#1) }> *)
Compute denote (denote (add1 l2)) (* = <{ \_, 8 }> *)
```

PHOAS enables programming across levels of meta-languages without binder bureaucracy, a pain that would only be amplified by an infinite tower of embedded languages, each with their own binders. Notice we maintain dependent types all the way down and can not get ill-typed terms at any level. We are very hopeful more interesting properties of PHOAS will come up, as it is a natural representation in the edge of meta and object language.

REFERENCES

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

[2] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 143–156.

[3] Adam Chlipala. 2010. A verified compiler for an impure functional language. *ACM Sigplan Notices* 45, 1 (2010), 93–106.

[4] Olivier Danvy and Lasse R Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. 162–174.

[5] Olivier Danvy, Morten Rhiger, and Kristoffer H Rose. 2001. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming* 11, 6 (2001), 673–680.

[6] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.
[7] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.
[8] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 271–283.
[9] LUCAS SILVER, IRENE YOON, YANNICK ZAKOWSKI, and STEVE ZDANCEWIC. [n. d.]. Equational Proofs of Optimizations with Interaction Trees. ([n. d.]).

APPENDIX A: PHOAS SYNTAX

Object-language types in PHOAS and type denotations in the meta-language.

```
Inductive type : Type :=
| TNum: type
| TArrow : type → type → type.

Declare Custom Entry stlc_ty.
Notation "<{{ e }}>" := e (e custom stlc_ty at level 99).
Notation "( x )" := x (in custom stlc_ty, x at level 99).
Notation "x" := x (in custom stlc_ty at level 0, x constr at level 0).
Notation "S → T" := (TArrow S T) (in custom stlc_ty at level 2, right associativity).
Notation "'Num'" := TNum (in custom stlc_ty at level 0).

Fixpoint typeDenote (t : type) : Set :=
  match t with
  | <{{ Num }}> ⇒ nat
  | <{{ t1 → t2 }}> ⇒ typeDenote t1 → typeDenote t2
  end.
```

Fig. 3. Types of $\lambda^{\rightarrow}$ language denote to Gallina types

The Simply-Typed Lambda Calculus ($\lambda^{\rightarrow}$) PHOAS syntax and denotational semantics.

```
Section vars.
  Variable var :  type ⟶ Type.
  Inductive Term: type ⟶ Type :=
  | NUM:  nat ⟶ Term <{{ Num }}>
  | ADD:  Term <{{  Num }}>  ⟶ Term <{{ Num }}>  ⟶ Term <{{ Num }}>
  | APP:  ∀ a b,  Term <{{  a ⟶ b }}>  ⟶ Term a ⟶ Term b
  | VAR:  ∀ a,  var a  ⟶ Term a
  | LAM:  ∀ a b,  (var a  ⟶ Term b) ⟶ Term <{{ a ⟶ b }}>.
End vars.

Fixpoint termDenote {t: type} (e :  Term typeDenote t) :  typeDenote t :=
  match e in (Term _ t)  return (typeDenote t) with
  | VAR v  ⇒ v
  | NUM f  ⇒ f
  | ADD l r  ⇒ (termDenote l) + (termDenote r)
  | APP e1 e2  ⇒ (termDenote e1) (termDenote e2)
  | LAM e'   ⇒ fun x ⇒ termDenote (e' x)
  end.

Declare Custom Entry stlc.
Notation "<{  e  }>" := e (e  custom stlc at level 99).
Notation "(  x  )" := x (in custom stlc,  x  at level 99).
Notation "x" :=  x  (in custom stlc at level 0,   x  constr at level 0).
Notation "x  y" := (APP  x  y)  (in custom stlc at level 2,   left associativity).
Notation "x  +  y" := (ADD  x  y)  (in custom stlc at level 2,   left associativity).
Notation "\  x  ,  y" :=
  (LAM (fun x  ⇒ y)) (in custom stlc at level 90,
                      x  constr,
                      y  custom stlc at level 80,
                      left associativity).
Notation "\_  ,  y" :=
  (LAM (fun _  ⇒ y)) (in custom stlc at level 90,
                      y  custom stlc at level 80,
                      left associativity).

Notation "#  n" := (NUM n)  (in custom stlc at level 0).
Notation "@  n" := (VAR n)  (in custom stlc at level 0,   n  custom stlc at level 1).
Notation "{  x  }" := x (in custom stlc at level 1,   x  constr).
```

Fig. 4. PHOAS syntax and denotational semantics for $\lambda^{\rightarrow}$

APPENDIX B: STRONG NORMALIZATION PROOF

Proof of strong-normalization for Nbe. The predicate `fof` indicates a first-order function, either a nullary function or an n-arry function whose arguments are all first-order terms. Then `value` indicates first-order values (does not include lambdas) and `hnff` is for *head-normal form*, a nullary function which is a value or a n-arry function which has all the binders in the top-level and the types of binders match the negative types in the type arrow. A function in in head-normal form is normal and has no undreduced beta-redexes.

```
Inductive fof: type → Prop :=
| fo_num: fof <{{ Num }}>
| fof_num: ∀ a,
    fof <{{ a }}> →
    fof <{{ Num → a }}>.

Inductive value: ∀ {t: type}, Term typeDenote t → Prop :=
| Value_var: ∀ x, @value <{{ Num }}> (@VAR typeDenote <{{ Num }}> x)
| Value_const: ∀ (x: nat), @value <{{ Num }}> (NUM x).

Inductive hnff: ∀ (t: type), Term typeDenote t → Prop :=
| HNF_num_ar: ∀ a f,
    (∀ (arg: typeDenote <{{ Num }}>), hnff <{{ a }}> (f arg)) →
    hnff <{{ Num → a }}> (LAM f)
| HNF_num: ∀ e,
    value e →
    hnff <{{ Num }}> e.

Theorem normalize_correct: ∀ (t: type) (e: Term typeDenote t),
    fof t →
    hnff t (normalize e).
Proof with eauto.
  induction t0;
    intros; dependent destruction e; cbn; try constructor;
      inversion H; clear H; subst; cbn; try constructor...
Defined.
```

APPENDIX C: METAPROGRAMMING DEMO

```
Fixpoint add1 {t: type} {v: type → Type} (e: Term v t): Term v t :=
  match e with
  | NUM f ⇒ NUM (f+1)
  | APP e1 e2 ⇒ APP (add1 e1) (add1 e2)
  | ADD e1 e2 ⇒ ADD (add1 e1) (add1 e2)
  | LAM e'  ⇒ LAM (fun x ⇒ add1 (e' x))
  | RET v ⇒ RET v
  end.

Tactic Notation "meta" uconstr(x) := refine x; exact typeDenote.
Definition l3 :=
  ltac:(meta <{ \_, #1 + (@ (#3 + (@( #1)))) }>).

Compute add1 l3.              (* = <{ \_, #2 + @ (#3) + @ (#1) }> *)
Compute denote (add1 l3).     (* = <{ \_, #2 + (#3 + @ (#1)) }> *)
Compute denote (add1 (denote (add1 l3))).
                              (* = <{ \_, #3 + #4 + #1 }> *)
Compute denote (denote (add1 (denote (add1 l3)))).
                              (* = <{ \_, 8 }> *)
```