Formalizing a Correctness Property of a Type-Directed Partial Evaluator

Noriko Hirota Kenichi Asai

Ochanomizu University
2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
{hirota.noriko,asai}@is.ocha.ac.jp

Abstract

This paper presents our experience of formalizing Danvy's type-directed partial evaluator (TDPE) for the call-by-name lambda calculus in the proof assistant Coq. Following the previous approach by Coquand and Ilik, we characterize TDPE as a composition of completeness and soundness theorems of typing rules with respect to the semantics. To show the correctness property of TDPE (i.e., TDPE preserves semantics), we further define a logical relation between residualizing and standard semantics, following Filinski. The use of parametric higher-order abstract syntax (PHOAS) leads to a simple formalization without being disturbed by fresh names created during TDPE. Because of the higher-order nature of PHOAS, it also requires us to prove manually a core property that corresponds to the main lemma of logical relations, which appears to be difficult to prove in Coq.

Categories and Subject Descriptors D.1.1 [*Software*]: Programming Techniques—applicative (functional) languages; D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—partial evaluation; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—lambda calculus and related systems

General Terms Languages, Theory, Verification

Keywords type-directed partial evaluator, parametric higher-order abstract syntax, proof assistant Coq

1. Introduction

Given a compiled representation of a program and its type, a type-directed partial evaluator (TDPE) [6] (also known as normalization by evaluation) returns the normal form of the source program. Unlike standard partial evaluators [13] that operate on the source program, TDPE does not inspect the internal structure of the program and thus is very fast. It simply uses the compiled representation of the input to extract its normal form using types as guidance. The efficiency of TDPE is attractive as a fast optimizer for a compiler. For example, Lindley [14] uses TDPE to optimize the SML.NET compiler.

Despite the simplicity of the definition of TDPE, its correctness property is not easy to establish. Filinski [7] showed the correctness property of call-by-name TDPE using denotational semantics and Kripke logical relations and extended it to arbitrary monadic effects [8]. Fiore [9] studied TDPE from a categorical and algebraic viewpoint. Similar approach is used to describe TDPE with sum types by Balat et al. [1]. However, it is not easy to apply these results to other languages. For example, the second author [15] proposed TDPE for a lambda calculus with delimited control operators, shift and reset [5] using shift and reset operators themselves, but its correctness property is still open; it is not at all clear how to apply denotational or categorical arguments here, because one would require denotational or categorical theory for control operators.

An interesting line of research on the property of TDPE is started by Coquand [4]. She showed a formalized proof of soundness and completeness of a simply-typed lambda calculus with respect to Kripke semantics. It turns out that the proof term of completeness coincides exactly to the reification function of TDPE. This work is followed by Ilik [10–12] who formalized various kinds of TDPE in the proof assistant Coq. The correspondence between TDPE and completeness clarifies the internal workings of TDPE, enabling easier formalization. Formalization of TDPE in a proof assistant has an advantage that every detail of the proof is spelled out and it can be basis for showing the correctness property of other TDPE. However, the correspondence alone does not prove the correctness property of TDPE.

As a first step towards formalizing and proving the correctness property of TDPE for shift and reset, this paper formalizes Filinski's proof for call-by-name TDPE in Coq. We characterize TDPE as completeness of a calculus with respect to standard semantics (rather than Kripke semantics), following the spirit of Coquand's work. We then extend it to cover the correctness property of TDPE, which states that TDPE does not change the semantics of its input.

To avoid the variable renaming problem, we employ parametric higher-order abstract syntax (PHOAS), proposed by Washburn and Weirich [16] and adapted to Coq by Chlipala [2, 3]. PHOAS enables us to formalize the correspondence elegantly. In particular, we do not have to worry about name generation required during TDPE. When proving the correctness property of TDPE, however, we are faced with a core property that corresponds to the main lemma of logical relations. It appears to be difficult to prove this property in Coq, as observed by Chlipala [2], due to the higher-order nature of PHOAS. Instead, we manually prove the property to establish the correctness property of TDPE.

The contributions of this paper are summarized as follows.

PLPV '14, January 21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2541568.2541572

- We formalize the correctness property of TDPE in Coq. The proof is simple. In particular, the generation of fresh variables is automatically handled thanks to the use of PHOAS.
- The development is a non-trivial case study of PHOAS. We point out its merit and weakness.

The paper is organized as follows. We introduce the PHOAS representation of terms, normal forms, and neutral terms in the next section. We then state soundness (Section 3) and completeness (Section 4) of the typing derivation with respect to the semantics. The correctness property of TDPE is shown in Section 5 and a pen-and-paper proof of the property needed to show the correctness property is presented in Section 6. We discuss related work in Section 7.

The proof script is available from: `http://pllab.is.ocha.ac.jp/~asai/papers/plpv14.v`

2. Terms

The terms we consider in this paper are the call-by-name simply-typed lambda calculus. The types are defined by:

```
Inductive typ : Set := base : typ
                     | arrow : typ -> typ -> typ.
```

We will use metavariables `A` and `B` for values of type `typ`.

We represent terms over `typ` using the parametric higher-order abstract syntax (PHOAS), proposed by Washburn and Weirich [16] and adapted to Coq by Chlipala [2, 3]. We first define `tm` that is parameterized over the representation of variables:

```
Inductive tm (var: typ -> Type) : typ -> Type :=
| tm_Var : forall A, var A -> tm var A
| tm_Lam : forall A B, (var A -> tm var B) ->
                       tm var (arrow A B)
| tm_App : forall A B, tm var (arrow A B) ->
                       tm var A -> tm var B.
```

Unlike the standard HOAS, the type of variables is parameterized as a function `var` over a type. By not instantiating `var` to `tm var` itself (with the same `var`), the negative occurrence of `tm` in its definition is avoided, while keeping most of the benefits of HOAS. The term is then defined by closing over `var`:

```
Definition TM A := forall var, tm var A.
```

We will use metavariables `t` and `T` for `tm` and `TM`, respectively.

When constructing a term of type `tm`, we omit writing `var`, `A`, and `B` by declaring they are implicit arguments:

```
Arguments tm_Var [var A] _.
Arguments tm_Lam [var A B] _.
Arguments tm_App [var A B] _ _.
```

Using this representation, for example, an identity function and the S combinator are represented as follows:

```
Example TM_id: TM (arrow base base) := fun var =>
  tm_Lam (fun x => tm_Var x).

Example TM_s:
  TM (arrow (arrow base (arrow base base))
            (arrow (arrow base base)
                   (arrow base base))) := fun var =>
  tm_Lam (fun x => tm_Lam (fun y => tm_Lam (fun z =>
    tm_App (tm_App (tm_Var x) (tm_Var z))
           (tm_App (tm_Var y) (tm_Var z))))).
```

The output of TDPE is in the $\beta\eta$-long normal form. We define the normal form and a neutral term using PHOAS as follows:

```
Inductive nf (var: typ -> Type) : typ -> Type :=
| nf_Lam : forall A B, (var A -> nf var B) ->
                       nf var (arrow A B)
| nf_ne  : ne var base -> nf var base
with ne (var: typ -> Type) : typ -> Type :=
| ne_Var : forall A, var A -> ne var A
| ne_App : forall A B, ne var (arrow A B) ->
                       nf var A -> ne var B.

Arguments nf_Lam [var A B] _.
Arguments nf_ne [var] _.
Arguments ne_Var [var A] _.
Arguments ne_App [var A B] _ _.
```

A normal form is either a neutral term of `base` type or a lambda abstraction whose body is in normal form. A neutral term is either a variable or a variable applied to normal forms. A neutral term represents an irreducible term unless the head variable is instantiated to an abstraction. Since only the neutral term of `base` type is allowed as a normal form, the above definition produces η-long normal forms. We will use metavariables `f` and `e` for `nf` and `ne`, respectively.

We can also define `NF` and `NE` by closing over `var`, although we will not use them in this paper.

```
Definition NF A := forall var, nf var A.
Definition NE A := forall var, ne var A.
```

A term of `nf` and `ne` can be injected to `tm` by the following two functions:

```
Fixpoint tm_of_nf {var A} (f: nf var A) : tm var A :=
  match f with
| nf_Lam _ _ f' => tm_Lam (fun x => tm_of_nf (f' x))
| nf_ne e' => tm_of_ne e'
  end
with tm_of_ne {var A} (e: ne var A) : tm var A :=
  match e with
| ne_Var _ x => tm_Var x
| ne_App _ _ e' f' => tm_App (tm_of_ne e')
                             (tm_of_nf f')
  end.
```

3. Soundness

The semantics of a type `A` is defined by the following predicate `V` (for values) parameterized by the semantics `b` of the `base` type:

```
Fixpoint V b (A: typ) : Type := match A with
| base => b
| arrow A B => V b A -> V b B
  end.
```

We interpret an arrow type from `A` to `B` as a Coq function from values of type `A` to values of type `B`.

The soundness of typing derivation with respect to the semantics is characterized by the following soundness theorem:

```
Theorem soundness: forall b {A}, tm (V b) A -> V b A.
```

It states that for any interpretation `b` for the `base` type, if `A` is derivable using typing rules, `A` is true in the semantics. It can be proved by simple induction on the typing derivation `tm (V b) A`.

The soundness theorem defines the semantics of a term `t` of type `A`. In fact, the proof term of the soundness theorem is the standard direct-style interpreter:

```
Fixpoint soundness b {A} (t: tm (V b) A) : V b A :=
  match t with
| tm_Var _ x => x
```

```
| tm_Lam _ _ t1 => fun x => soundness b (t1 x)
| tm_App _ _ t1 t2 => (soundness b t1)
                       (soundness b t2)
  end.
```

Given a typing derivation `t` for a type `A`, `soundness` computes the compiled value of `t` represented as a Coq value.

The above `soundness` defines a call-by-name interpreter, because the metalanguage Coq allows full β reduction and thus can be regarded as call by name. The call-by-name nature of the interpreter is traced back to the definition of `V`, where the denotation of `arrow` type is defined as the Coq function type. It suggests that if we want to formalize a TDPE for a call-by-value language (in our future work), we need to define `V` in the continuation-passing style to enforce call-by-value semantics in the call-by-name metalanguage.

Finally, the interpreter for a term `T` of type `TM A` can be defined as follows:

```
Definition soundness2 b {A} (T: TM A) :=
  soundness b (T (V b)).
```

4. Completeness

The core of TDPE is characterized by a function `reify` that maps a semantic (compiled) value back to a normal form. It is formalized mutually recursively with a function `reflect`:

```
Theorem reify: forall var A,
  V (ne var base) A -> nf var A
with  reflect: forall var A,
  ne var A -> V (ne var base) A.
```

The first part of this theorem is the completeness theorem of a typing derivation with respect to the semantics: for any value of type `V (ne var base) A`, it constructs a typing derivation of `A` (using only the normal form construction). We can prove this theorem by induction on `A`. Alternatively, we can define the proof terms for `reify` and `reflect` directly:

```
Fixpoint reify (var: typ -> Type) (A: typ) :=
  match A return V (ne var base) A -> nf var A with
| base => fun v => nf_ne v
| arrow A B => fun v =>
    nf_Lam (fun x =>
            reify var B
                  (v (reflect var A (ne_Var x))))
  end
with reflect (var: typ -> Type) (A: typ) :=
  match A return ne var A -> V (ne var base) A with
| base => fun e => e
| arrow A B => fun e v =>
    reflect var B (ne_App e (reify var A v))
  end.
```

This definition exactly corresponds to Danvy's original definition [6] of `reify` ($\downarrow$) and `reflect` ($\uparrow$):

$$
\begin{array}{rcl}
\downarrow^{\mathsf{base}} v & = & v \\
\downarrow^{A \to B} v & = & \underline{\lambda} x^{\diamond}.\, \downarrow^{B} (v\, \overline{@}\, \uparrow_{A} x^{\diamond}) \\
 & & \text{where } x^{\diamond} \text{ is a fresh variable} \\
 & & \\
\uparrow_{\mathsf{base}} e & = & e \\
\uparrow_{A \to B} e & = & \overline{\lambda} v.\, \uparrow_{B} (e\, \underline{@}\, \downarrow^{A} v)
\end{array}
$$

Here, overlined constructs represent static terms (Coq functions) and underlined constructs represent datatypes (`nf_Lam`, `ne_App`, etc.). Danvy's definition requires that the name $x^{\diamond}$ is chosen fresh. This name generation is handled in our Coq formalization simply using PHOAS: `nf_Lam` is passed a Coq function with a bound variable `x`, which is guaranteed to be fresh by the metalanguage Coq.

Intuitively, `reify` transforms a semantic value of type `A` into its syntactic normal form of type `A`. At first sight, it appears to be impossible to do such a thing when `A` is a function type. We know it has the form `nf_Lam` f for some f, but there appears to be no way to find the shape of f. The parametricity comes to the rescue here. If a value v has type $\alpha \to A$ for an uninterpreted type variable α (in our case, `base`) and arbitrary A, we know that v does not inspect its argument. Thus, it is safe to apply v to a syntactic variable $x^{\diamond}$ to see what its body f looks like. Furthermore, since v will evaluate its body to a normal form when applied, we obtain the body f of v in normal form. This is why $\downarrow$ (`reify`) applies v to (a reflect of) $x^{\diamond}$. The above story does not go well, if the argument of v is a function. For that case, we need to properly lift the argument $x^{\diamond}$ to a higher-order value using $\uparrow$ (`reflect`), since it might be applied in the body f.

Using `soundness2` and `reify`, a term `T` of type `A` is normalized to its normal form by first passing `A` and `T` to `soundness2` (with `b` being `(ne var base)` for an arbitrary `var`) to obtain the meaning of `T`, and then by passing the result to `reify` together with `A` and `var` to obtain its normal form.

In the definition (and the theorem), the choice of `(ne var base)` as the base case `b` of `V` is crucial. Since `reify` regards an input of type `base` uninterpreted as a normal form (a neutral term to be more specific), `reify` does not work for an arbitrary `b` but only for a normal form. At reification time, the value of type `base` is interpreted as a neutral term. Filinski referred to it as the residualizing interpretation [7]. In contrast, `soundness` works for any `b`, which Filinski referred to as the evaluating interpretation.

As an example of how to use TDPE, consider the following term representing $\lambda x.\, (\lambda y.\, y)\, x$ of type `TM (arrow base base)`:

```
Example TM_id': TM (arrow base base) := fun var =>
  tm_Lam (fun x =>
          tm_App (tm_Lam (fun y => tm_Var y))
                 (tm_Var x)).
```

To obtain its normal form, we compute as follows for an arbitrary `var`:

```
Eval compute in
  (reify var (arrow base base)
             (soundness2 (ne var base) TM_id')).
```

```
= nf_Lam (fun x : var base => nf_ne (ne_Var x))
: nf var (arrow base base)
```

We observe that reduction under lambda is performed and a normal form $\lambda x.\, x$ is obtained. Since we use typeful representation, we can omit the type in `reify` and `b` in `soundness2` and have them inferred by Coq:

```
Eval compute in (reify var _ (soundness2 _ TM_id')).
```

```
= nf_Lam (fun x : var base => nf_ne (ne_Var x))
: nf var (arrow base base)
```

5. Correctness

Although the theorem in the previous section shows that `reify` in TDPE corresponds to the completeness of typing derivation of a term (a normal form, in particular) with respect to the semantics, it does not directly prove the correctness property of TDPE.[1] In this

[1] It would be interesting if we could use the parametricity argument to prove the correctness of TDPE directly from the completeness result, but this is beyond the scope of this paper.

section, we establish the correctness property of TDPE (i.e., the semantics is preserved by TDPE) using a logical relation argument.

Define the following two relations:

```
Definition interp_nf b {A} (f: nf (V b) A)
                           (v: V b A) :=
  soundness b (tm_of_nf f) = v.

Definition interp_ne b {A} (e: ne (V b) A)
                           (v: V b A) :=
  soundness b (tm_of_ne e) = v.
```

The relation `interp_nf b f v` (`interp_ne b e v`) represents that a normal form `f` (a neutral term `e`, respectively) of type `A` is evaluated to a value `v`.

The goal of this section is to show the following theorem:

```
Theorem correctness: forall b A (T: TM A),
  interp_nf b (reify (V b) A
                     (soundness2 (ne (V b) base) T))
              (soundness2 b T).
```

For any well-typed term `T` of type `A`, the result of TDPE (obtained by evaluating `T` and reifying the result) behaves the same as evaluating the original term `T`. Notice that two occurrences of `soundness2` in the theorem are applied to different interpretation for the `base` type. The first one uses `(ne (V b) base)`, because the result is passed to `reify` (which requires that `base` be interpreted as neutral terms). In other words, `T` has to be evaluated under residualizing interpretation here.

To prove this correctness theorem, we use the standard logical relation argument. The logical relation `R` relates a reification-time value `e` to a runtime value `v` and is defined as follows:

```
Fixpoint R b (A: typ) :=
  match A
  return V (ne (V b) base) A -> V b A -> Type with
| base => fun e v => interp_ne b e v
| arrow A B => fun e v => forall e' v',
    R b A e' v' -> R b B (e e') (v v')
  end.
```

It is defined by induction on the type `A`. For the `base` type, the input `e` is related to `v` if `e` evaluates to `v`. Remember that when `A` is `base`, `e` is a neutral term that is uninterpreted at reification time. For a higher type, `e` and `v` are related if their application to related arguments are also related.

The logical relation `R b A e v` can be regarded as an extension of `V b' A` where `b'` is `(ne (V b) base)`. If we take a *slice* of `R` related to `e`, i.e., if we ignore `v` and `interp_ne` and regard `R b A e v` as `e: V b' A`, the above definition of `R` coincides with the definition of `V`. The logical relation `R` extends `V` with the information that `e` evaluates to `v`.

The correctness theorem we want to establish is divided into two lemmas. The first one shows the relationship between the completeness theorem and `R`.

```
Lemma reify_R: forall b A (v: V (ne (V b) base) A)
                          (f: V b A),
  R b A v f -> interp_nf b (reify (V b) A v) f
with reflect_R: forall b A (e: ne (V b) A)
                           (v: V b A),
  interp_ne b e v -> R b A (reflect (V b) A e) v.
```

This lemma is an extension of the completeness theorem and includes information on the interpretation of terms. For example, `reify_R` states that `v` and the reification of `v` are evaluated to the same value `f`. Similarly for `reflect_R`. As such, the proof of this lemma goes much the same way as the completeness theorem (by induction on type `A`). One complication is that we need to keep track of how term constructions and `interp_nf`, `interp_ne` interact with each other. We used the following helper propositions, which are all proved easily.

```
Proposition interp_nf_ne: forall b
        (e: ne (V b) base) v,
  interp_ne b e v -> interp_nf b (nf_ne e) v.

Proposition interp_nf_Lam: forall b A B
        (f: V b A -> nf (V b) B)
        (v: V b (arrow A B)),
  (forall v', interp_nf b (f v') (v v')) ->
  interp_nf b (nf_Lam f) v.

Proposition interp_ne_Var: forall b A (v: V b A),
  interp_ne b (ne_Var v) v.

Proposition interp_ne_App: forall b A B
        (e: ne (V b) (arrow A B)) e' f f',
  interp_ne b e e' ->
  interp_nf b f f' ->
  interp_ne b (ne_App e f) (e' f').
```

These propositions enable us to regard terms in PHOAS much the same way as the first-order representation most of the time. The second one, `interp_nf_Lam`, characterizes how PHOAS representation of abstractions interacts with `interp_nf`.

We note here that the proof of `interp_nf_Lam` requires the `extensionality` tactic, which states that two functions are equal if they are always equal when applied to the same arguments. The proposition `interp_nf_Lam` is in turn used in the proof of the lemma `reify_R` for the function case: when `v` evaluates to `f` (i.e., `R b A v f`), the reification of `v` does not exactly evaluate to `f` itself, but evaluates to a function that, when applied to an argument, behaves the same as `f`. Thus, we need the `extensionality` tactic to prove the correctness property of TDPE, although it was not required for the soundness and completeness theorems.

The second lemma that we need for the proof of correctness of TDPE is the main lemma of logical relations.

```
Lemma main: forall b A (T: TM A),
  R b A (soundness2 (ne (V b) base) T)
        (soundness2 b T).
```

It states the reification-time value and the standard value of the same well-typed term `T` of type `A` are related. If this lemma is proved, the correctness theorem can be simply proved by applying the two lemmas:

```
Proof. intros. apply reify_R. apply main. Qed.
```

In words, the `main` lemma establishes its conclusion for any well-typed term `T`, to which we can apply `reify_R` directly to obtain the required correctness.

However, the `main` lemma appears to be unprovable in Coq. We want to prove it by induction on the structure of `T`. However, `T` is represented in PHOAS and is not inductive. To apply induction, we have to unfold the definition of `T`. By unfolding `soundness2` in the `main` lemma, we obtain:

```
R b A (soundness (ne (V b) base)
                 (T (V (ne (V b) base))))
      (soundness b (T (V b)))
```

Here, `(T (V (ne (V b) base)))` and `(T (V b))` have type `tm (V (ne (V b) base))` and `tm (V b)`, respectively, and are both inductive. They are exactly the same term except for the interpretation of `base` type. To factor out this property, we can define the following modified main lemma:

```
Lemma main': forall b A t1 t2,
  related_term b t1 t2 ->
  R b A (soundness (ne (V b) base) t1)
        (soundness b t2).
```

where `related_term` is defined as follows:

```
Inductive related_term b : forall {A},
    tm (V (ne (V b) base)) A -> tm (V b) A -> Type :=
| related_Var : forall A
    (v: V (ne (V b) base) A) (v': V b A),
    R b A v v' ->
    related_term b (tm_Var v) (tm_Var v')
| related_Lam : forall A B
    (t: V (ne (V b) base) A ->
        tm (V (ne (V b) base)) B)
    (t': V b A -> tm (V b) B),
    (forall v v', R b A v v' ->
                  related_term b (t v) (t' v')) ->
    related_term b (tm_Lam t) (tm_Lam t')
| related_App : forall A B
    (t1: tm (V (ne (V b) base)) (arrow A B))
    (t1': tm (V b) (arrow A B)) t2 t2',
    related_term b t1 t1' ->
    related_term b t2 t2' ->
    related_term b (tm_App t1 t2) (tm_App t1' t2').
```

The predicate `related_term b t1 t2` means that the two terms, `t1` and `t2`, have the same structure and their subterms have suitable relationship. The modified `main'` lemma can be proved by induction on the proof of `related_term b t1 t2`.

It is then straightforward to prove `main` using `main'`, if we can prove the following property. That is, the same term `T` instantiated with different interpretation for `base` are related terms:

Property . `related_term b (T (V (ne (V b) base))) (T (V b))`

This property appears to hold. The predicate `related_term b t1 t2` simply says that the two terms `t1` and `t2` have the same structure and possess a suitable property. However, we were unable to prove it in Coq because of the higher-order nature of the representation of terms: we cannot induct on `T` because it is a function. Instead, we establish it by a manual proof in the next section. Chlipala [2] assumes a similar axiom for proving the correctness of CPS transformation using PHOAS.

6. Manual proof of the property

In this section, we prove the necessary property manually. We first prove a general theorem that holds for open terms and derive the property as its corollary.

Suppose that a well-typed term `T` contains free variables $x_1, \ldots, x_n$ of type $A_1, \ldots, A_n$, i.e., `T` contains a subterm `tm_Var` x_i where x_i is a free variable in the metalanguage Coq. We write `T`$[t_i/x_i]$ for substituting a term t_i for x_i. We then prove the following theorem:

Theorem . If `R b` A_i t_i t_i' holds for all i, we have

```
related_term b (T[t_i/x_i] (ne (V b) base))
               (T[t_i'/x_i] (V b)).
```

Proof. By induction on the structure of `T`. We utilize the fact that the metalanguage Coq avoids name conflicts automatically.

(`T` **is** `fun var => tm_Var` x_i) The goal is:

```
related_term b
  ((fun var => tm_Var t_i) (ne (V b) base))
  ((fun var => tm_Var t_i') (V b))
```

which simplifies to:

```
related_term b (tm_Var t_i) (tm_Var t_i').
```

From the definition of `related_term`, we need to show `R b` A_i t_i t_i', which holds by assumption.

(`T` **is** `fun var => tm_Lam (fun x => t)`) The goal is:

```
related_term b
  ((fun var =>
    tm_Lam (fun x => t[t_i/x_i])) (ne (V b) base))
  ((fun var =>
    tm_Lam (fun x => t[t_i'/x_i])) (V b))
```

which can be rewritten to:

```
related_term b
  (tm_Lam (fun x =>
           ((fun var => t[t_i/x_i]) (ne (V b) base))))
  (tm_Lam (fun x =>
           ((fun var => t[t_i'/x_i]) (V b)))).
```

From the definition of `related_term`, we need to show for any t and t' of type A such that `R b` A t t',

```
related_term b
  ((fun x =>
    ((fun var => t[t_i/x_i]) (ne (V b) base))) t)
  ((fun x =>
    ((fun var => t[t_i'/x_i]) (V b))) t')
```

which reduces to:

```
related_term b
  ((fun var => t[t_i/x_i, t/x]) (ne (V b) base))
  ((fun var => t[t_i'/x_i, t'/x]) (V b))
```

which holds from the induction hypothesis.

(`T` **is** `fun var => tm_App t1 t2`) The goal is:

```
related_term b
  ((fun var =>
    tm_App t1[t_i/x_i] t2[t_i/x_i]) (ne (V b) base))
  ((fun var =>
    tm_App t2[t_i'/x_i] t2[t_i'/x_i]) (V b))
```

which can be rewritten to:

```
related_term b
  (tm_App ((fun var => t1[t_i/x_i]) (ne (V b) base))
          ((fun var => t2[t_i/x_i]) (ne (V b) base)))
  (tm_App ((fun var => t1[t_i'/x_i]) (V b))
          ((fun var => t2[t_i'/x_i]) (V b))).
```

From the definition of `related_term`, we need to show:

```
related_term b
  ((fun var => t1[t_i/x_i]) (ne (V b) base))
  ((fun var => t1[t_i'/x_i]) (V b))
```

and

```
related_term b
  ((fun var => t2[t_i/x_i]) (ne (V b) base))
  ((fun var => t2[t_i'/x_i]) (V b))
```

both of which hold from the induction hypotheses.

□

By instantiating the above theorem to the case where T is closed (i.e., $n = 0$), we obtain the following corollary that we wanted to prove.

Corollary . For any closed term T, we have
`related_term b (T (V (ne (V b) base))) (T (V b)).`

Thus, we can assume the following axiom, which is then used to finish the proof of the `main` lemma.

```
Axiom T_related: forall b A (T: TM A),
  related_term b (T (V (ne (V b) base))) (T (V b)).
```

7. Related Work

The proof shown in this paper closely corresponds to the proof given by Filinski [7]. The values of types `(V (ne (V b) base) A)` and `(V b A)` correspond to Filinski's residualizing and evaluating interpretation, respectively. Since Filinski handled name generation explicitly, he introduced Kripke semantics to keep track of generated names. In our proof, name generation is all handled automatically by the metalanguage Coq, allowing us to stay in the standard semantics. Filinski also handled static divergence. He allowed infinite loop for the parts executed at TDPE time. This feature is not supported in our formalization, because all the definable terms in our language are simply typed.

Coquand [4] formalized a proof of soundness and completeness in the proof editor ALF and observed that the completeness corresponds exactly to TDPE. Subsequently, Ilik [10–12] formalized TDPE for call by name, for call by value, and for call by value with delimited control operators. Both Coquand and Ilik used Kripke semantics to keep track of name generation. We followed the same strategy for the first half of our formalization, but without using Kripke semantics. Thanks to the simple formalization using PHOAS, the extracted proof is simple and corresponds directly to Danvy's original TDPE. This is in contrast to the previous work where the proof term was cluttered with bookkeeping operations (such as name generation) and was often hard to decipher.

PHOAS was used to show correctness of program transformation in Coq [2, 3]. There, Chlipala observed that some property (like our property on `related_term`) appears to be impossible to prove in Coq and resorted to proving it outside of Coq. We followed this approach to prove the property on `related_term`.

8. Conclusion

As an attempt toward establishing the correctness of TDPE for shift and reset, we have formalized Filinski's proof of correctness of call-by-name TDPE in Coq. The use of PHOAS leads to a sufficiently simple proof that can be a basis for showing the correctness property of a more sophisticated TDPE.

The correctness of call-by-name TDPE itself is already established by Filinski long time ago. In this sense, this paper does not add any new facts about TDPE. Still, we believe our Coq formalization is worthwhile because it serves as a good and rigorous explanation of Filinski's work which is not completely easy to follow; it can be a basis for extension because of the simplicity of the proof script; and it is a non-trivial case study of PHOAS.

In the future, we plan to extend this proof to handle call by value as well as delimited control operators, and ultimately to prove the correctness property of our TDPE for shift and reset [15]. One milestone for this goal is to transform the whole development into a continuation-passing style (CPS). Filinski's work [8] will be a guide again. The initial development shows that it is not straightforward to define CPS values by induction on the structure of types, because the answer type is not a subterm of a type in question. For example, the CPS transform of a type `A` is `arrow (arrow A B) B` for any answer type `B`, but `B` is not a part of `A`. We are currently investigating the possibility of fixing the answer type to a particular pre-determined type.

References

[1] Balat, V., R. D. Cosmo, and M. Fiore "Extensional Normalization and Type-Directed Partial Evaluation for Typed Lambda Calculus with Sums," *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pp. 64–76 (January 2004).

[2] Chlipala, A. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics," *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pp. 143–156 (September 2008).

[3] Chlipala, A. *Certified Programming with Dependent Types*, available from `http://adam.chlipala.net/cpdt/`.

[4] Coquand, C. "A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions" *Higher-Order and Symbolic Computation*, Volume 15, Issue 1, pp. 57-90 (March 2002).

[5] Danvy, O., A. Filinski "Abstracting Control," *ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[6] Danvy, O. "Type-Directed Partial Evaluation," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257 (January 1996).

[7] Filinski, A. "A Semantic Account of Type-Directed Partial Evaluation," In G. Nadathur, editor, *Principles and Practice of Declarative Programming (LNCS 1702)*, pp. 378–395 (September 1999).

[8] Filinski, A. "Normalization by Evaluation for the Computational Lambda-Calculus," In S. Abramsky, editor, *Typed Lambda Calculi and Applications (LNCS 2044)*, pp. 151–165 (May 2001).

[9] Fiore, M. "Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus," *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pp. 26–37 (October 2002).

[10] Ilik, D. *Constructive Completeness Proofs and Delimited Control*, Ph.D. thesis, Ecole Polytechnique X (October 2010).

[11] Ilik, D. "Continuation-passing style models complete for intuitionistic logic", *Annals of Pure and Applied Logic, Special issue: Classical logic and computation 2010.*

[12] Ilik, D. "A formalized type-directed partial evaluator for shift and reset", Control Operators and their Semantics, available from `http://arxiv.org/abs/1210.2094`, 18 pages, (April 2013).

[13] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[14] Lindley, S. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*, Ph.D. thesis, University of Edinburgh (2005).

[15] Tsushima, K., and K. Asai "Towards Type-Directed Partial Evaluation for Shift and Reset," *Proceedings of the 2009 Workshop on Normalization by Evaluation*, pp. 57–64 (August 2009).

[16] Washburn, G., and S. Weirich "Boxes Go Bananas: Encoding Higher-order Abstract Syntax with Parametric Polymorphism," *Journal of Functional Programming*, Vol. 18, No. 1, pp. 87–140, Cambridge University Press (January 2008).