Extracting a Call-by-Name Partial Evaluator from a Proof of Termination

Kenichi Asai
Ochanomizu University
Tokyo, Japan
asai@is.ocha.ac.jp

Abstract

It is well known that the computational content of a termination proof of a calculus is an interpreter that computes the result of an input term. Traditionally, such extraction has been tried for a calculus with deterministic reduction rules, producing the result as a value, i.e., in weak head normal form where no redexes are reduced under lambda. In this paper, we consider non-deterministic reduction rules where any redexes can be reduced, even the ones under lambda, and extract a partial evaluator, rather than an interpreter, that produces the result in normal form. We formalize a call-by-name, simply-typed, lambda calculus in the Agda proof assistant and prove its termination using a logical predicate. We observe that the extracted program can be regarded as an online partial evaluator and present future perspectives about how we can extend the framework to a call-by-value calculus.

CCS Concepts • **Mathematics of computing** → *Lambda calculus*; • **Theory of computation** → *Proof theory*;

Keywords Partial evaluation, termination, logical relation, de Bruijn index, Agda

ACM Reference Format:
Kenichi Asai. 2019. Extracting a Call-by-Name Partial Evaluator from a Proof of Termination. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3294032.3294084

1 Introduction

A term e in a calculus is terminating, if there exists a value v such that e reduces to v and v does not reduce further (i.e., terminates). By the Curry-Howard correspondence, the computational content of this proof is to compute the value v from the input term e. If we can show that all the terms in a calculus are terminating, the computational content of the proof is an interpreter that computes the result value from any input term.

Related work. Biernacka, Danvy, and Støvring [7] prove termination of both call-by-name and call-by-value lambda calculi in a first-order minimal logic and extract interpreters for both calculi from the termination proofs. Biernacka and Biernacki [6] extend this approach to a calculus with an abortive control operator *call/cc* and obtain interpreters written in continuation-passing style (CPS). However, they all handle weak normalization and do not reduce redexes under lambda. Thus, the extracted programs are interpreters, rather than a partial evaluator.

For strong normalization, Berger, Berghofer, Letouzey, and Schwichitenberg [5] extract an interpreter from termination proofs in Minlog, Coq, and Isabelle/HOL. However, the extracted program follows the normalization-by-evaluation approach, which operates by induction on types and is closer to type-directed partial evaluation [8].

This work. In this paper, we aim at obtaining a traditional partial evaluator [10] that operates by induction on terms. We do so by sticking to the original idea of regarding a termination proof as an evaluator: we only change the reduction relation and observe that the proof of termination is a partial evaluator.

We formalize the call-by-name lambda calculus in the Agda proof assistant using de Bruijn index and renaming. We then prove termination of the calculus for both deterministic and non-deterministic reduction. For the former, we obtain an interpreter. For the latter, we obtain a partial evaluator, assuming that the calculus has the Church-Rosser property. The resulting partial evaluator is essentially the standard one that eagerly evaluates all redexes.

Although we obtain a partial evaluator, it is for a call-by-name calculus. Ultimately, we want to obtain a partial evaluator for a call-by-value calculus. However, it turns out that it is not so straightforward. This paper sets the scene and serves as the first step toward obtaining a partial evaluator for a call-by-value calculus.

In the next section, we present the base calculus we work on and show how we formalize it in Agda using renaming

PEPM '19, January 14–15, 2019, Cascais, Portugal
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6226-9/19/01...$15.00
https://doi.org/10.1145/3294032.3294084

$$
\begin{array}{rcll}
\tau & := & \mathsf{int} \mid \tau \to \tau & \text{types} \\[6pt]
x & := & \mathsf{z} \mid \mathsf{s}\, x & \text{variables} \\
v & := & n \mid \lambda.\, e & \text{values} \\
e & := & v \mid x \mid e \,@\, e & \text{terms} \\[6pt]
ne & := & x \mid ne \,@\, nf & \text{neutral terms} \\
nf & := & ne \mid n \mid \lambda.\, nf & \text{normal forms}
\end{array}
$$

Figure 1. Types, variables, terms, and normal forms

and substitution. In Section 3, we review the termination proof found in Pierce's textbook [12] and show that the computational content of the proof is an interpreter. In Section 4, we extend the calculus to include non-deterministic reduction rules and obtain a partial evaluator. In Section 5, we conclude with future perspectives on obtaining a partial evaluator for a call-by-value calculus.

The paper is accompanied by supplementary files that contain the complete formalization and termination proof in Agda, available at `http://pllab.is.ocha.ac.jp/~asai/papers/pepm2019/`.

2 Term Representation

The types and terms we consider in this paper are presented in Figure 1. We employ de Bruijn indices [9] to represent variables, using zero (z) and a successor function (s). A value is either a natural number n (of base type int, to be distinguished from variables) or a lambda abstraction. Since our language is call by name, variables are not values; they can be bound to arbitrary computation including applications. A term is either a value, a variable, or an application.

When we consider a partial evaluator, we will talk about normal forms. A normal form is either a natural number, a lambda abstraction whose body is also in normal form, or a *neutral* term, which is a stuck term whose head position is a variable and whose argument position is a normal form. It is easy to see that the normal form in the standard call-by-name lambda calculus with unrestricted β-reduction is specified by this grammar.

In our Agda formalization, we represent all the terms in a typeful manner, following Altenkirch and Reus [2]. Put differently, we use intrinsically typed representation where typing rules are incorporated into the definition of terms. The typing rules are shown in Figure 2. We represent a type environment Γ as a list of types, using OCaml notation where [] and :: represent an empty type environment and the cons operator, respectively. We will also use the OCaml list notation $[\tau_1; \tau_2; \ldots]$ for concrete examples of type environments.

The typing rules are standard. To obtain intuition on how the typing rules for variables work, we show a judgement for an example term $\lambda x.\, a \,@\, x \,@\, b$ with two free variables a and b. Let Γ be a type environment $[\mathsf{int} \to \mathsf{int} \to \mathsf{int}; \mathsf{int}]$

$$\boxed{\Gamma \vdash_{\mathsf{v}} x : \tau}$$

$$\frac{}{\tau :: \Gamma \vdash_{\mathsf{v}} \mathsf{z} : \tau}\ (\textsc{Zero}) \qquad \frac{\Gamma \vdash_{\mathsf{v}} x : \tau}{\sigma :: \Gamma \vdash_{\mathsf{v}} \mathsf{s}\, x : \tau}\ (\textsc{Suc})$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ (\textsc{TNum}) \qquad \frac{\tau_2 :: \Gamma \vdash e : \tau_1}{\Gamma \vdash \lambda.\, e : \tau_2 \to \tau_1}\ (\textsc{TAbs})$$

$$\frac{\Gamma \vdash_{\mathsf{v}} x : \tau}{\Gamma \vdash x : \tau}\ (\textsc{TVar}) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \,@\, e_2 : \tau_1}\ (\textsc{TApp})$$

$$\boxed{\Gamma \vdash_{\mathsf{ne}} nf : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{v}} x : \tau}{\Gamma \vdash_{\mathsf{ne}} x : \tau}\ (\textsc{NeVar})$$

$$\frac{\Gamma \vdash_{\mathsf{ne}} ne : \tau_2 \to \tau_1 \quad \Gamma \vdash_{\mathsf{nf}} nf : \tau_2}{\Gamma \vdash_{\mathsf{ne}} ne \,@\, nf : \tau_1}\ (\textsc{NeApp})$$

$$\boxed{\Gamma \vdash_{\mathsf{nf}} nf : \tau}$$

$$\frac{}{\Gamma \vdash_{\mathsf{nf}} n : \mathsf{int}}\ (\textsc{NfNum}) \qquad \frac{\tau_2 :: \Gamma \vdash_{\mathsf{nf}} nf : \tau_1}{\Gamma \vdash_{\mathsf{nf}} \lambda.\, nf : \tau_2 \to \tau_1}\ (\textsc{NfAbs})$$

Figure 2. Typing rules

keeping the types of a and b in this order.

$$\frac{\dfrac{\mathsf{int} :: \Gamma \vdash \mathsf{s\,z} \,@\, \mathsf{z} : \mathsf{int} \to \mathsf{int} \quad \mathsf{int} :: \Gamma \vdash \mathsf{s\,(s\,z)} : \mathsf{int}}{\mathsf{int} :: \Gamma \vdash \mathsf{s\,z} \,@\, \mathsf{z} \,@\, \mathsf{s\,(s\,z)} : \mathsf{int}}\ (\textsc{TApp})}{\Gamma \vdash \lambda.\, \mathsf{s\,z} \,@\, \mathsf{z} \,@\, \mathsf{s\,(s\,z)} : \mathsf{int} \to \mathsf{int}}\ (\textsc{TAbs})$$

In the top-right premise, we see that the second free variable b (namely, $\mathsf{s\,(s\,z)}$) has the second type int in Γ as follows (expanding the definition of Γ):

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathsf{int} :: [\,] \vdash_{\mathsf{v}} \mathsf{z} : \mathsf{int}}\ (\textsc{Zero})}{\mathsf{int} \to \mathsf{int} \to \mathsf{int} :: \mathsf{int} :: [\,] \vdash_{\mathsf{v}} \mathsf{s\,z} : \mathsf{int}}\ (\textsc{Suc})}{\mathsf{int} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} :: \mathsf{int} :: [\,] \vdash_{\mathsf{v}} \mathsf{s\,(s\,z)} : \mathsf{int}}\ (\textsc{Suc})}{\mathsf{int} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} :: \mathsf{int} :: [\,] \vdash \mathsf{s\,(s\,z)} : \mathsf{int}}\ (\textsc{TVar})$$

Likewise, from the top-left premise, we see that x has type int and a has type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$, the first type in Γ.

2.1 Renaming

To define a reduction relation, we need to define substitution. However, it is not quite straightforward to define substitution using de Bruijn indices. Here, following Benton et al. [4], we first define parallel renaming of variables before defining substitution. It gives us a systematic view on renaming and substitution. It is also known that this technique can be used to define generic simulation and fusion lemmas [1].

A renaming ρ is a list of variables, where the i-th element is a variable to be substituted for the i-th variable of the term being renamed. For example, if we have $\rho_0 = [\mathsf{z}; \mathsf{s\,(s\,z)}; \mathsf{s\,(s\,(s\,z))}]$, the following renaming happens:

$$
\begin{array}{rcl}
\mathsf{z} & \to & \mathsf{z} \\
\mathsf{s\,z} & \to & \mathsf{s\,(s\,z)} \\
\mathsf{s\,(s\,z)} & \to & \mathsf{s\,(s\,(s\,z))}
\end{array}
$$

$$\boxed{\Delta \vdash_{\mathsf{r}} \rho : \Gamma}$$

$$\frac{}{\Delta \vdash_{\mathsf{r}} [] : []}\ (\textsc{Rmt}) \qquad \frac{\Delta \vdash_{\mathsf{v}} x : \tau \quad \Delta \vdash_{\mathsf{r}} \rho : \Gamma}{\Delta \vdash_{\mathsf{r}} x :: \rho : \tau :: \Gamma}\ (\textsc{Rcons})$$

Figure 3. Renaming

If we apply this renaming to the body of the previous example term $\mathsf{s\,z} \mathbin{@} \mathsf{z} \mathbin{@} \mathsf{s\,(s\,z)}$, we obtain $\mathsf{s\,(s\,z)} \mathbin{@} \mathsf{z} \mathbin{@} \mathsf{s\,(s\,(s\,z))}$. That is, we pushed away the two (originally free) variables and made room for a new variable $\mathsf{s\,z}$. It is used when we put a term under a new binder. Suppose that we want to put our example term $\lambda x.\, a \mathbin{@} x \mathbin{@} b$ under $\lambda y. \ldots$ to obtain $\lambda y.\, \lambda x.\, a \mathbin{@} x \mathbin{@} b$. The de Bruijn index of x is unchanged, but those of a and b are pushed forward by one to account for the new bound variable y. This is exactly what the above renaming achieves.

An identity renaming ρ_{id} is a list of increasingly ordered de Bruijn indices:

$$\rho_{\mathsf{id}} \quad = \quad [\mathsf{z}; \mathsf{s\,z}; \mathsf{s\,(s\,z)}; \ldots]$$

Renaming is formally defined in Figure 3. A renaming ρ with the judgement $\Delta \vdash_{\mathsf{r}} \rho : \Gamma$ renames a variable in Γ to a variable in Δ. We write $\rho(x)$ to denote the variable that x is renamed to. We define $\mathsf{wkr}(\rho)$ and $\mathsf{liftr}(\rho)$ as follows:[1]

$$\begin{aligned} \mathsf{wkr}([]) &= [] \\ \mathsf{wkr}(x :: \rho) &= \mathsf{s}\,x :: \mathsf{wkr}(\rho) \\ \mathsf{liftr}(\rho) &= \mathsf{z} :: \mathsf{wkr}(\rho) \end{aligned}$$

Intuitively, $\mathsf{wkr}(\rho)$ adds one to (weakens by one) all the variable indices while $\mathsf{liftr}(\rho)$ additionally allocates a new variable at the front. The latter corresponds to applying ρ under a new binder.

We can now define remaining on terms, which is also written as an application of ρ, as follows:

$$\begin{aligned} \rho(n) &= n \\ \rho(\lambda.\, e) &= \lambda.\, (\mathsf{liftr}(\rho))(e) \\ \rho(x) &= \rho(x) \\ \rho(e_1 \mathbin{@} e_2) &= \rho(e_1) \mathbin{@} \rho(e_2) \end{aligned}$$

where by abuse of notation, the left-hand side of the third equation is an application of ρ to a term (which happens to be a variable) while the right-hand side is an application of ρ to a variable index. By inspection, we see that if we have $\Gamma \vdash e : \tau$ and $\Delta \vdash_{\mathsf{r}} \rho : \Gamma$, then we have $\Delta \vdash \rho(e) : \tau$. (See the accompanying Agda code.)

2.2 Substitution

We are now ready to define substitution. Whereas a renaming ρ is a list of variables, a substitution σ is a list of terms: the i-th element of the list is a term to be substituted for the i-th variable. (If our calculus was call by value, we would represent σ as a list of values.) For example, if we have $\sigma_0 = [\lambda.\, \lambda.\, \mathsf{s\,z}; 3]$, the following substitution happens:

$$\begin{aligned} \mathsf{z} &\rightarrow \lambda.\, \lambda.\, \mathsf{s\,z} \\ \mathsf{s\,z} &\rightarrow 3 \end{aligned}$$

An identity substitution σ_{id} is a list of increasingly ordered variables:

$$\sigma_{\mathsf{id}} \quad = \quad [\mathsf{z}; \mathsf{s\,z}; \mathsf{s\,(s\,z)}; \ldots]$$

Although this definition appears to be identical to ρ_{id}, they are different: ρ_{id} is a list of variables whereas σ_{id} is a list of terms (which happen to be variables).

Substitution is formally defined in Figure 4. Similarly to renaming, we define $\mathsf{wks}(\sigma)$ and $\mathsf{lifts}(\sigma)$ as follows:

$$\begin{aligned} \mathsf{wks}([]) &= [] \\ \mathsf{wks}(e :: \sigma) &= (\mathsf{liftr}(\rho_{\mathsf{id}}))(e) :: \mathsf{wks}(\sigma) \\ \mathsf{lifts}(\rho) &= \mathsf{z} :: \mathsf{wks}(\rho) \end{aligned}$$

In the definition of $\mathsf{wks}(e :: \sigma)$, $\mathsf{liftr}(\rho_{\mathsf{id}})$ adds one to all the free variables in its argument e. Observe how renaming is used here to define $\mathsf{wks}(e :: \sigma)$.

Substitution on terms is defined as follows:

$$\begin{aligned} \sigma(n) &= n \\ \sigma(\lambda.\, e) &= \lambda.\, (\mathsf{lifts}(\sigma))(e) \\ \sigma(x) &= \sigma(x) \\ \sigma(e_1 \mathbin{@} e_2) &= \sigma(e_1) \mathbin{@} \sigma(e_2) \end{aligned}$$

For example, if we apply σ_0 defined above to our example term $\lambda.\, \mathsf{s\,z} \mathbin{@} \mathsf{z} \mathbin{@} \mathsf{s\,(s\,z)}$, we obtain $\lambda.\, (\lambda.\, \lambda.\, \mathsf{s\,z}) \mathbin{@} \mathsf{z} \mathbin{@} 3$ as shown by the following derivation. Define

$$\begin{aligned} \sigma_0' &= \mathsf{lifts}(\sigma_0) \\ &= \mathsf{z} :: \mathsf{wks}(\sigma_0) \\ &= \mathsf{z} :: (\lambda.\, \lambda.\, \mathsf{s\,z}) :: 3 :: [] \end{aligned}$$

We then have:

$$\begin{aligned} \sigma_0(\lambda.\, \mathsf{s\,z} \mathbin{@} \mathsf{z} \mathbin{@} \mathsf{s\,(s\,z)}) &= \lambda.\, \sigma_0'(\mathsf{s\,z} \mathbin{@} \mathsf{z} \mathbin{@} \mathsf{s\,(s\,z)}) \\ &= \lambda.\, (\lambda.\, \lambda.\, \mathsf{s\,z}) \mathbin{@} \mathsf{z} \mathbin{@} 3 \end{aligned}$$

$$\boxed{\Delta \vdash_{\mathsf{s}} \sigma : \Gamma}$$

$$\frac{}{\Delta \vdash_{\mathsf{s}} [] : []}\ (\textsc{Smt}) \qquad \frac{\Delta \vdash e : \tau \quad \Delta \vdash_{\mathsf{s}} \sigma : \Gamma}{\Delta \vdash_{\mathsf{s}} e :: \sigma : \tau :: \Gamma}\ (\textsc{Scons})$$

Figure 4. Substitution

[1] We follow the formalization given by Abel found at: `https://github.com/andreasabel/strong-normalization/tree/master/agda`. (Accessed on October 7, 2018.)

$$\frac{}{(\lambda.\, e_1)\ @\ e_2 \rightsquigarrow (e_2 :: \sigma_{\mathrm{id}})(e_1)}\ (\beta)$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ @\ e_2 \rightsquigarrow e_1'\ @\ e_2}\ (\textsc{EApp}_1)$$

Figure 5. Call-by-name reduction relation

2.3 Reduction Relation

We can now define the call-by-name reduction relation. See Figure 5. In the (β) rule, $e_2 :: \sigma_{\mathrm{id}}$ is the following substitution:

$$\begin{array}{rcl} \mathsf{z} & \rightarrow & e_2 \\ \mathsf{s\,z} & \rightarrow & \mathsf{z} \\ \mathsf{s\,(s\,z)} & \rightarrow & \mathsf{s\,z} \\ \mathsf{s\,(s\,(s\,z))} & \rightarrow & \mathsf{s\,(s\,z)} \\ & \vdots & \end{array}$$

That is, it replaces the variable zero with e_2 and decreases all other variables by one, thus correctly realizing β-reduction. Since the calculus is call by name, e_2 is substituted without being evaluated.

The reflexive, transitive closure of the relation $\cdot \rightsquigarrow \cdot$ is written as $\cdot \rightsquigarrow^* \cdot$.

Since the reduction is deterministic, we can show that the reduct of a term is uniquely determined.

Proposition 2.1. *If we have both $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$, then we have $e_1 = e_2$.*

3 Termination Proof in the Textbook

In this section, we review the termination proof shown by Pierce [12], slightly adjusted to the call-by-name setting. We first define what it means for a term to be terminating.

Definition 3.1. A term e is *terminating* if there exists a value v such that $e \rightsquigarrow^* v$.

To prove termination, one first defines a logical predicate by induction on the structure of types.

Definition 3.2. A logical predicate R on a closed term e of type τ (i.e., we assume $[] \vdash e : \tau$) is defined by induction on τ as follows:

1. $R_{\mathrm{int}}(e)$ iff e is terminating.
2. $R_{\tau_2 \to \tau_1}(e)$ iff e is terminating and for all e_2 of type τ_2 that satisfies $R_{\tau_2}(e_2)$, we have $R_{\tau_1}(e\ @\ e_2)$.

An interesting point of the definition of R is that it not only states that e is terminating, but also tells us that the application of e to a (terminating) argument is terminating. The latter makes the predicate strong enough to capture the property of the static reduction of a term e.

We immediately see by a straightforward case analysis that e is terminating if it satisfies R.

Proposition 3.3. *If $R_\tau(e)$, then e is terminating.*

The next step is to show that the predicate R is preserved by both directions of reduction.

Proposition 3.4. *Suppose $[] \vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2$.*

1. *If $R_\tau(e_1)$, then $R_\tau(e_2)$.*
2. *If $R_\tau(e_2)$, then $R_\tau(e_1)$.*

We omit the simple proof and refer the reader to the textbook [12]. However, we show a seemingly innocent proposition that is needed to prove the above proposition, as it plays an important role in the next section.

Proposition 3.5. *Suppose $[] \vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2$.*

1. *If e_1 is terminating, then e_2 is terminating.*
2. *If e_2 is terminating, then e_1 is terminating.*

The second part is easy to prove: if e_2 is terminating with a value v_2, e_1 is surely terminating because $e_1 \rightsquigarrow e_2 \rightsquigarrow^* v_2$. The first part is more subtle. Since e_1 is terminating, we have $e_1 \rightsquigarrow^* v_1$ for some v_1. However, even if we have $e_1 \rightsquigarrow e_2$, we do not know if e_2 goes back to some term within the reduction sequence of $e_1 \rightsquigarrow^* v_1$. This is where we use uniqueness of reduction (Proposition 2.1): as we have both $e_1 \rightsquigarrow e_2$ and $e_1 \rightsquigarrow^* v_1$, the second term in the reduction sequence of $e_1 \rightsquigarrow^* v_1$ must be e_2. Thus, we have $e_2 \rightsquigarrow^* v_1$, finishing the proof.

We are now ready to prove the main lemma of the logical predicate.

Lemma 3.6. *Assume $\Gamma \vdash e : \tau$ and $[] \vdash \sigma : \Gamma$. If all the terms in σ satisfy the predicate R with their respective types, then we have $R_\tau(\sigma(e))$.*

The first premise says that e is a well-typed term having free variables listed in Γ. The second premise says σ is a list of closed terms that covers all the free variables of e. Thus, after applying substitution, $\sigma(e)$ is a closed term. As a whole, the main lemma says if e is well typed, its closed instances satisfy the logical predicate R.

The proof term $\mathsf{eval}(e, \sigma)$ of the lemma essentially becomes as follows:

$$\begin{array}{rcl} \mathsf{eval}(n, \sigma) & = & n \\ \mathsf{eval}(\lambda.\, e, \sigma) & = & (\lambda.\, (\mathsf{lifts}(\sigma))(e), \mathsf{evlam}(e, \sigma)) \\ \mathsf{eval}(x, \sigma) & = & \mathsf{lookup}(x, \sigma) \\ \mathsf{eval}(e_1\ @\ e_2, \sigma) & = & \mathsf{let}\ (\_, r_1) = \mathsf{eval}(e_1, \sigma)\ \mathsf{in} \\ & & \mathsf{let}\ r_2 = \mathsf{eval}(e_2, \sigma)\ \mathsf{in} \\ & & r_1(\sigma(e_2), r_2) \\ \mathsf{evlam}(e, \sigma)(e_2, r_2) & = & \mathsf{eval}(e, e_2 :: \sigma) \\ \mathsf{lookup}(\mathsf{z}, e :: \sigma) & = & e \\ \mathsf{lookup}(\mathsf{s}\, x, e :: \sigma) & = & \mathsf{lookup}(x, \sigma) \end{array}$$

Given a term e and a substitution σ (serving as a runtime environment), $\mathsf{eval}(e, \sigma)$ proves that $\sigma(e)$ satisfies R by providing the value $\sigma(e)$ evaluates to, together with a static function to perform evaluation in the case e has a function type.

$$\frac{e_2 \rightsquigarrow e_2'}{e_1 @ e_2 \rightsquigarrow e_1 @ e_2'} \ (\text{EApp}_2) \qquad \frac{e \rightsquigarrow e'}{\lambda.\, e \rightsquigarrow \lambda.\, e'} \ (\text{EFun})$$

Figure 6. Additional reduction relation to permit arbitrary β-reduction

Let us closely look at the proof term. If e is a natural number n, n itself is the value it evaluates to. If e is an abstraction $\lambda.\, e$, it returns a *symbolic value* [13] that consists of a dynamic value representing the result of evaluation and a static function that performs further computation. The former is the evidence that $\lambda.\, e$ terminates, while the latter is the proof that $\lambda.\, e$ is terminating if applied to a terminating argument. The returned static function, $\mathsf{evlam}(e, \sigma)$, receives an argument e_2 and a proof r_2 that e_2 satisfies R, and proves that the body e of the abstraction satisfies R in an extended environment where the variable zero is replaced with the argument. (To show that the right-hand $\mathsf{eval}(e, e_2 :: \sigma)$ of $\mathsf{evlam}(e, \sigma)(e_2, r_2)$ is actually what we want, we need to use Proposition 3.4. We elide this part, because it is purely for the proof and does not produce any computational content. See the accompanying Agda code for details.)

If e is a variable x, it looks up the environment σ to extract the corresponding term. Since all terms in σ satisfy R, it gives the required proof for this case. Finally, if e is an application $e_1 @ e_2$, we recurse on e_1 to retrieve a static function r_1 and passes the (unevaluated) argument $\sigma(e_2)$ to prove that $e_1 @ e_2$ satisfies R.

It is a bit peculiar here that we need to recurse on e_2, too, to obtain the evidence that e_2 satisfies R. In call-by-name languages, we should not evaluate the argument e_2. The recursive call is justified, because it is done solely for the purpose of the proof. If we divide the proof term into two, one for computing the result and one for proving that R is satisfied, the recursive call on e_2 appears only in the latter.

As a whole, we successfully extracted a call-by-name interpreter from a termination proof.

4 Termination Proof of Strong Reduction

The program we extracted in the previous section is an interpreter. It produces only weak-head normal forms and does not reduce redexes under lambda. In this section, we turn our attention to extracting a partial evaluator rather than an interpreter. The basic idea is to replace the reduction relation: rather than using a deterministic relation, we use a relation that permits β-reduction at any place in a term, even under lambda. By reducing all redexes, we would obtain a normal form (rather than a weak head normal form), and the extracted program of its termination proof would be a partial evaluator.

Figure 6 shows the reduction relation we add to the original reduction relation in Figure 5. We allow β-reduction at argument position (regardless of the function position is value or not) and under lambda. Because of the addition, the reduction becomes non-deterministic: if both the function position and the argument position of an application are β-redexes, we can use both (EApp$_1$) and (EApp$_2$). Thus, uniqueness of reduction (Proposition 2.1) no longer holds.

The addition to the reduction relation also affects the definition of termination. Since we want to reduce all β-redexes, we say a term e is terminating only when it reduces to a normal form, rather than a value.

Definition 4.1. A term e is *terminating* if there exists a normal form nf such that $e \rightsquigarrow^* nf$.

Reducing a term under lambda means we have to handle open terms, i.e., reduction in the presence of free variables. In the previous section, the relation R is defined for closed terms only. In particular, the definition of R for a function type considers only when a closed term is applied to a closed argument. Here, we have to extend it to cope with open terms.

Definition 4.2. A logical predicate R on an open term e of type τ under Γ (i.e., we assume $\Gamma \vdash e : \tau$) is defined by induction on τ as follows:

1. $R^{\Gamma}_{\mathsf{int}}(e)$ iff e is terminating.
2. $R^{\Gamma}_{\tau_2 \to \tau_1}(e)$ iff e is terminating and for all e_2 of type τ_2 under Δ (i.e., $\Delta \vdash e_2 : \tau_2$) that satisfies $R^{\Delta}_{\tau_2}(e_2)$, and for all σ that renames variables from Γ to Δ (i.e, $\Delta \vdash_r \sigma : \Gamma$), we have $R^{\Delta}_{\tau_1}(\sigma(e) @ e_2)$.

This definition is different from the one in the previous section in two ways. First, the definition of termination is changed. The new definition returns a normal form as a proof term rather than a value. Secondly, in the definition of R for a function type, the argument e_2 does not have to be typed under the original Γ, but under any Δ which extends Γ (with a renaming σ). In other words, R is now defined in Kripke style. This flexibility is required when we evaluate under lambda.

The rest of the story goes similarly to the previous section, but with extra care on non-deterministic reduction and renaming of variables. As in the previous section, R implies termination.

Proposition 4.3. *If $R^{\Gamma}_{\tau}(e)$, then e is terminating.*

To show that reduction preserves R, we need to show that reduction preserves termination. However, due to the non-deterministic reduction, the proof is not straightforward. In particular, we have to assume Church-Rosser property. Let $\cdot \sim \cdot$ be a reflective, symmetric, transitive closure of $\cdot \rightsquigarrow \cdot$.

Postulate 4.4 (Church-Rosser)**.** *If $e_1 \sim e_2$, there exists e such that $e_1 \rightsquigarrow^* e$ and $e_2 \rightsquigarrow^* e$.*

It is possible to prove this theorem, as we all know it holds for the call-by-name lambda calculus [3]. We did not prove it, however, since it is already known to hold, is tedious, and does not contribute to the computational content of the proof. We thus accept it as is and go forward.

Given the Church-Rosser property, we can prove that equality preserves termination.

Proposition 4.5. *Suppose $e_1 \sim e_2$. If e_1 is terminating, then e_2 is terminating.*

Since e_1 is terminating, we have $e_1 \rightsquigarrow^* \mathit{nf}_1$ for some normal form nf_1. From $e_1 \sim e_2$ and $e_1 \rightsquigarrow^* \mathit{nf}_1$, we have $e_2 \sim \mathit{nf}_1$. From the Church-Rosser property, we have a term e such that $e_2 \rightsquigarrow^* e$ and $\mathit{nf}_1 \rightsquigarrow^* e$. Since nf_1 is a normal form, we must have $\mathit{nf}_1 = e$. Thus, we have $e_2 \rightsquigarrow^* \mathit{nf}_1$ as required.

From this proposition, we immediately obtain:

Proposition 4.6. *Suppose $\Gamma \vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2$.*

1. *If e_1 is terminating, then e_2 is terminating.*
2. *If e_2 is terminating, then e_1 is terminating.*

and the preservation of R by reduction:

Proposition 4.7. *Suppose $\Gamma \vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2$.*

1. *If $R^{\Gamma}_{\tau}(e_1)$, then $R^{\Gamma}_{\tau}(e_2)$.*
2. *If $R^{\Gamma}_{\tau}(e_2)$, then $R^{\Gamma}_{\tau}(e_1)$.*

We can now prove the main lemma of the logical predicate.

Lemma 4.8. *Assume $\Gamma \vdash e : \tau$ and $\Delta \vdash \sigma : \Gamma$. If all the terms in σ satisfy the predicate R under Δ with their respective types, then we have $R^{\Delta}_{\tau}(\sigma(e))$.*

Like Lemma 3.6, the first premise says that e is a well-typed term having free variables listed in Γ. Unlike Lemma 3.6, the second premise says σ does *not* close all the free variables of e; the free variables in Δ remain after substitution. The conclusion says that the resulting term $\sigma(e)$ satisfies the logical predicate R under Δ.

The proof term $\mathsf{eval}(e, \sigma)$ of the lemma essentially becomes as follows:

$$\begin{array}{rcl}
\mathsf{eval}(n, \sigma) & = & n \\
\mathsf{eval}(\lambda.\, e, \sigma) & = & \mathsf{let}\ \mathit{nf} = \mathsf{eval}(e, \mathsf{lifts}(\sigma))\ \mathsf{in} \\
 & & (\lambda.\, \mathit{nf}, \mathsf{evlam}(e, \sigma)) \\
\mathsf{eval}(x, \sigma) & = & \mathsf{lookup}(x, \sigma) \\
\mathsf{eval}(e_1 \mathbin{@} e_2, \sigma) & = & \mathsf{let}\ (\_, r_1) = \mathsf{eval}(e_1, \sigma)\ \mathsf{in} \\
 & & \mathsf{let}\ r_2 = \mathsf{eval}(e_2, \sigma)\ \mathsf{in} \\
 & & r_1(\rho_{\mathsf{id}}, \sigma(e_2), r_2) \\
\mathsf{evlam}(e, \sigma)(\rho, e_2, r_2) & = & \mathsf{eval}(e, e_2 :: \mathsf{map}\ \rho\ \sigma)
\end{array}$$

The changes from the previous section are three-fold. First, in the case for abstraction $\lambda.\, e$, we evaluate the body e under the lifted environment $\mathsf{lifts}(\sigma)$ to obtain the normal form of $\lambda.\, e$. In other words, we partially evaluate e with the variable zero unknown to obtain its normal form. Secondly, the static part of a function, $\mathsf{evlam}(e, \sigma)$, receives an additional renaming ρ to be mapped over σ to account for any environment extensions. Finally, in the case for application, an identity renaming ρ_{id} is passed accordingly to indicate that the static function r_1 should be evaluated under the current environment.

It may first appear that since we pass only ρ_{id} to r_1, it would not be necessary to pass a renaming at all. This is not the case, since behind the scene, eval receives an additional proof stating that each element of σ satisfies R. When we evaluate a body e of an abstraction, as in $\mathsf{eval}(e, \mathsf{lifts}(\sigma))$, we have to construct a proof that each element of $\mathsf{lifts}(\sigma)$ satisfies R. It eventually requires us to prove that arbitrary weakening (renaming) preserves R (as well as termination, equality, and reduction).

The obtained partial evaluator exhibits features that are present in the traditional online partial evaluator and the one that is not. As in the traditional online partial evaluator, it constructs both the specialized dynamic function for residualization and a static function for further specialization. To construct the former, we eagerly evaluate the body under an environment where the current variable is unknown (which we could postpone until the abstraction is actually residualized for efficiency reason).

On the other hand, to specialize an application, we recurse over the function part and simply apply the result to an argument. This is in contrast to the traditional partial evaluator, where we dispatch according to whether the function part is statically known or not. If it is known at partial evaluation time, we proceed, while if it is not, we residualize. In the extracted program above, such a case dispatch does not exist. This is because our definition of the logical predicate R is quite expressive: it constructs a neutral term when the function part is a variable (dynamic). In other words, the case analysis on whether the function part is statically known is built into the definition of R. It would be interesting to investigate if we could define a weaker version of R that does not handle application of a dynamic variable and thus we need to dispatch explicitly in the application case of eval.

5 Conclusion and Perspectives

In this paper, we have shown that we can extract a standard online partial evaluator from a termination proof of a call-by-name calculus with a non-deterministic reduction relation.

Although we could obtain a partial evaluator for the call-by-name lambda calculus, our goal is to obtain a partial evaluator for the call-by-value lambda calculus. A preliminary investigation shows that scaling the textbook proof to deterministic call-by-value case is straightforward (but with some twists to make sure that the argument to be substituted is always a value). For the non-deterministic case, however, it is non-trivial. We ultimately want to extract a CPS-based partial evaluator [11] that performs the so-called let insertion. To support let insertion, however, it turns out that we need to incorporate a CPS transformation or a monadic transformation into account. We would then need a reduction relation that can simulate reduction in CPS [14]. As future work, we intend to pursue this direction.

Acknowledgments

I would like to thank Youyou Cong and anonymous reviewers for comments and discussions. This work was partly supported by JSPS KAKENHI under Grant No. JP18H03218.

References

[1] Allais, G., J. Chapman, C. McBride, and J. McKinna "Type-and-scope safe programs and their proofs," *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP'17)*, pp. 195–207 (January 2017).

[2] Altenkirch, T., and B. Reus "Monadic Presentations of Lambda Terms Using Generalized Inductive Types," *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL'99)*, pp. 453–468 (September 1999).

[3] Barendregt, H. P. *The Lambda Calculus: its Syntax and Semantics*, North-Holland (1984).

[4] Benton, N., C-K. Hur, A. J. Kennedy, and C. McBride "Strongly Typed Term Representations in Coq," *Journal of Automated Reasoning*, Vol. 49, No. 2, pp. 141-159, Springer (August 2012).

[5] Berger, U., S. Berghofer, P. Letouzey, and H. Schwichitenberg "Program extraction from normalization proofs," *Studia Logica 82*, pp. 25–49 (2006).

[6] Biernacka, M., and D. Biernacki "A Context-based Approach to Proving Termination of Evaluation," *Electronic Notes in Theoretical Computer Science*, Vol. 249, pp. 169–192 (August 2009).

[7] Biernacka, M., O. Danvy, and K. Støvring "Program Extraction From Proofs of Weak Head Normalization," *Electronic Notes in Theoretical Computer Science*, Vol. 155, pp. 169–189 (May 2006).

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

[9] de Bruijn, N. G. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," *Indagationes Mathematicae*, Vol. 34, pp. 381–392 (1972).

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

[11] Lawall, J. L., and O. Danvy "Continuation-Based Partial Evaluation," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).

[12] Pierce, B. C. *Types and Programming Languages*, Cambridge: MIT Press (2002).

[13] Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.

[14] Sabry, A., and M. Felleisen "Reasoning about Programs in Continuation-Passing Style," *Lisp and Symbolic Computation*, Vol. 6, Nos. 3/4, pp. 289–360, Kluwer Academic Publishers (1993).