Type Directed Partial Evaluation for Level-1 Shift and Reset

Danko Ilik[*]
Laboratory for Complex Systems and Networks
Macedonian Academy of Sciences and Arts
Skopje, Macedonia
`danko.ilik@gmail.com`

We present an implementation in the Coq proof assistant of type directed partial evaluation (TDPE) algorithms for call-by-name and call-by-value versions of shift and reset delimited control operators, and in presence of strong sum types. We prove that the algorithm transforms well-typed programs to ones in normal form. These normal forms can not always be arrived at using the so far known equational theories. The typing system does not allow answer-type modification for function types and allows delimiters to be set on at most one atomic type. The semantic domain for evaluation is expressed in Constructive Type Theory as a dependently typed monadic structure combining Kripke models and continuation passing style translations.

1 Introduction

Type directed partial evaluation (TDPE) is a technique that partially evaluates a program by first compiling it, and pre-computing known ("static") input data on the fly, and then decompiling it to normal form in an efficient process driven by the program's type. It was discovered by Danvy [6] in Programming Languages Theory, although the exact same algorithm had been isolated at about the same time also in the study of typed lambda calculi and in Logic: Berger and Schwichtenberg [3] found it while looking for an efficient procedure for reducing *open* lambda terms and called it Normalization by Evaluation (NBE); Catarina Coquand [5] realized that it is the procedure behind the proof of completeness of minimal intuitionistic logic (without $\bot, \vee$ and $\exists$) with respect to Kripke models.

However, when one moves from simply typed lambda calculus towards richer programming languages, to extend the TDPE method to cope with the new constructs does not appear to be straightforward. Already adding strong sum types seems to require one to implement TDPE using delimited control operators – indeed, this is one of the more important applications of Danvy and Filinski's operators shift and reset [8]. In turn, when considering TDPE for a language extended with the delimited control operators themselves, there has only been preliminary work on the subject, for the call-by-value case, by Tsushima and Asai [18].

In this paper, we consider TDPE for the first level of the shift and reset hierarchy. Using their simpler non-extended CPS semantics, we build a type-theoretic framework that acts as a specification for TDPE algorithms (Section 2). The algorithms themselves, for both call-by-value and call-by-name, are given in Section 3, where we also look at specific examples and compare their partial evaluations to the ones predicted by the known equational theories. In the concluding Section 4, we give further explanation about our implementation and about the related works.

The Coq implementation of the algorithms can be found at the address dankoi.github.com/metamath. Originally, this work was conceived as an alternative normalization proof for the core logical system from [12], a proper constructive extension of intuitionistic logic with delimited control operators.

[*]This work is covered by a Kurt Gödel Research Prize Fellowship 2011

$$\frac{}{\mathsf{hyp} : A,\Gamma \vdash_b A} \qquad \frac{p : \Gamma \vdash_b A}{\mathsf{wkn}(p) : B,\Gamma \vdash_b A}$$

$$\frac{p : \Gamma \vdash_b A}{\mathsf{inl}(p) : \Gamma \vdash_b A \vee B} \qquad \frac{p : \Gamma \vdash_b B}{\mathsf{inr}(p) : \Gamma \vdash_b A \vee B}$$

$$\frac{p : \Gamma \vdash_b A \vee B \qquad q : A,\Gamma \vdash_b C \qquad r : B,\Gamma \vdash_b C}{\mathsf{case}(p,q,r) : \Gamma \vdash_b C}$$

$$\frac{p : A,\Gamma \vdash_b B}{\mathsf{lam}(p) : \Gamma \vdash_b A \to B} \qquad \frac{p : \Gamma \vdash_b A \to B \qquad q : \Gamma \vdash_b A}{\mathsf{app}(p,q) : \Gamma \vdash_b B}$$

$$\frac{p : \Gamma \vdash_1 \bot}{\mathsf{reset}(p) : \Gamma \vdash_b \bot} \qquad \frac{p : A \to \bot,\Gamma \vdash_1 \bot}{\mathsf{shift}(p) : \Gamma \vdash_1 A}$$

Table 1: A typing system for lambda calculus with sum types and shift and reset, where variable binding is handled using deBruijn indices (hyp and $\mathsf{wkn}(\cdot)$)

2 Type-theoretic Model

The programming language that we want to partially evaluate, our *object language*, will be the lambda calculus with function and sum types and the shift/reset delimited control operators, described in Table 1. We do not work with the most general known typing system for shift and reset in which implication is a quaternary connective [7] and we allow a delimiter ($\mathsf{reset}(\cdot)$) to be set only at an atomic type ($\bot$).

For expressing variable binding, we rely on deBruijn indices in the form of hyp and $\mathsf{wkn}(\cdot)$ rules, where hyp can be thought of as zero and $\mathsf{wkn}(\cdot)$ as the successor. Lambda abstraction ($\mathsf{lam}(\cdot)$) and control ($\mathsf{shift}(\cdot)$) are therefore unary.

The turnstile "$\vdash$" is annotated by a Boolean b of value 0 or 1, value 1 meaning that a delimiting $\mathsf{reset}()$ has been previously applied in the lambda term (typing tree derivation). All rules, except for $\mathsf{shift}(\cdot)$ and $\mathsf{reset}(\cdot)$ ignore this annotation b. The rule $\mathsf{reset}()$ sets it to 1, and $\mathsf{shift}()$ can only be used if the annotation has been previously set to 1 i.e. in a delimited sub-term.

The idea behind every TDPE algorithm is the following: we want to transform a program written in the object language to a meta-level "bytecode" version of it, "run" this bytecode (this is called the *evaluation* phase), and then, based on the program's type, recover a program in the object language that is already in normal form (the *reification* phase) and $\beta(\eta)$-equal to the starting one. In other words, one relies on normalization at the meta-level, to produce an object-level normal form. This becomes non-trivial if the meta-level and the object level are not essentially the same, like in our case where the meta-level has no control feature while the object-level does.

The essential choice to make is what to choose for the "semantic" meta-level structure that evaluation will take place in. CPS semantics imposes itself, because it is the orthodox and simplest way to specify shift and reset [9]. The TDPE will thus be of the form of the two-phase transformation,

$$\text{Syntax} \rightsquigarrow ((\text{Value} \to \text{Answer}) \to \text{Answer}) \rightsquigarrow \text{Syntax},$$

where Syntax denotes the type of programs of the object language, and Value and Answer are “values” and “answers” of a “continuation” in the usual terminology [6]. If we also want the transformation to account for *open* terms (which allows to do normalization below a binder) and to guarantee that the input and output programs are actually programs of the same type, we need to enrich the semantic domain (bytecode) by a pre-order, keeping track of a context Γ denoting open variables, and a parameter A corresponding to the type of the transformed program. We obtain the statement

$$\Gamma \vdash A \rightsquigarrow \Gamma \Vdash A \rightsquigarrow \Gamma \vdash^{\mathrm{nf}} A,$$

where $\Gamma \Vdash A$ denotes the semantic domain that is the target of evaluation (Subsection 2.1), and source of reification (Subsection 2.2).

2.1 Evaluating into the Models

We will use a combination of Kripke-CPS models for classical logic (used previously with Lee and Herbelin for proving NBE for the classical sequent calculus $\mathrm{LK}_{\mu\tilde{\mu}}$ [14]) and those for intuitionistic logic (used for proving NBE for intuitionistic natural deduction with $\vee$ and $\exists$ [13]). We give the mathematical definitions, trying to be precise but as informal as possible – the interested reader may find the fully formal version in the Coq implementation – keeping also in mind that, while we do use dependent types, the dependencies are rather weak (Π-types over the small set of formulas, booleans, and the type K).

Definition 2.1. A *Kripke CPS structure* is given by a type K, a relation $\leq : (K \to K \to \text{Type})$ that is a preorder, i.e. both of

$$\begin{aligned} w &\leq w && \text{(reflexivity)} \\ w_1 \leq w_2 \to w_2 \leq w_3 &\to w_1 \leq w_3 && \text{(transitivity)} \end{aligned}$$

hold, and a relation

$$X : K \to \text{Bool} \to \text{Formula} \to \text{Type}$$

with the properties:

$$\begin{aligned} & w_1 \leq w_2 \to Xw_1bA \to Xw_2bA && (\leq\text{-monotonicity}) \\ & b_1 \sqsubseteq b_2 \to Xwb_1A \to Xwb_2A && (\sqsubseteq\text{-monotonicity}) \\ & Xw1\bot \to Xw0\bot && (\text{meta-reset}()). \end{aligned}$$

Bool is the type of booleans with inhabitants 0 (false) and 1 (true), and $\sqsubseteq$ is the order on booleans defined by the relation of less-than-or-equal of their numerical values. “Type” denotes the type universe of the meta-language, while Formula is the type of types of the object language i.e. those built from $\to$, $\vee$, atomic types, and the special fixed type $\bot$ that reset can be set on – we do not make the usual assumption that $\bot$ denotes the empty type, it is simply a notation for a chosen atomic type.

Inhabitants w of type K are called *worlds*, and when we have $XwbA$ we say that the world w is *exploding* for the formula A with annotation b. This terminology (“exploding” or “fallible”) comes from classic use of Kripke models when interpreting absurdity in a constructive way [17]. The relation X, the answer type of the continuations, will later be instantiated with the set of typable terms in normal form, that is, it will be used to pass on the output of the TDPE between different sub-phases of the algorithm in the process of building the final normal form.

Definition 2.2. Given $F : (K \to \text{Bool} \to \text{Formula} \to \text{Type})$, $A : \text{Formula}$, $b : \text{Bool}$, $w : K$, the dependently typed continuations "monad", $w \Vdash_b A$, defined by

$$w \Vdash_0 A := (C : \text{Formula})(w_1 : K)(w \leq w_1 \to \\ (w_2 : K)(w_1 \leq w_2 \to F w_2 0 A \to X w_2 0 C) \to X w_1 0 C)$$

$$w \Vdash_1 A := (w_1 : K)(w \leq w_1 \to \\ (w_2 : K)(w_1 \leq w_2 \to F w_2 1 A \to X w_2 1 \bot) \to X w_1 1 \bot)$$

is called *forcing*. That is, we read $w \Vdash_b A$ as "the world w forces the type A with annotation b".

Remark 2.3. We have put the word "monad" in quotes because we have not sought to prove the usual categorical or the functional programming laws for monads hold. Yet, the fact that we can define the monadic unit, bind, and run, will be quite convenient for structuring the computation/proofs later on.

The following two definitions present two alternatives that can be used to instantiate F from Definition 2.2; when the (non-strong) forcing relation is used in the definitions, it is implicitly instantiated with the strong forcing relation being defined. Note that, type theoretically, (non-strong) forcing and strong forcing need not be defined simultaneously, Definition 2.2 comes first.

Definition 2.4 (Strong forcing, call-by-value variant)**.** The *strong forcing* relation $w \Vdash^s_b A$ is defined by recursion on the type A, by the following clauses:

$$\begin{aligned} w \Vdash^s_b A &:= XwbA \qquad (A - \text{atomic type}) \\ w \Vdash^s_b A \vee B &:= w \Vdash^s_b A + w \Vdash^s_b B \\ w \Vdash^s_b A \to B &:= (w' : K)(w \leq w' \to w' \Vdash^s_b A \to w' \Vdash_b B) \end{aligned}$$

Definition 2.5 (Strong forcing, call-by-name variant)**.** The *strong forcing* relation $w \Vdash^s_b A$ is defined by recursion on the type A, by the following clauses:

$$\begin{aligned} w \Vdash^s_b A &:= XwbA \qquad (A - \text{atomic type}) \\ w \Vdash^s_b A \vee B &:= w \Vdash_b A + w \Vdash_b B \\ w \Vdash^s_b A \to B &:= (w' : K)(w \leq w' \to w' \Vdash_b A \to w' \Vdash_b B) \end{aligned}$$

Although a different strong forcing relation $(\cdot \Vdash^s_\cdot \cdot)$ determines a different forcing relation $(\cdot \Vdash_\cdot \cdot)$, the important properties that hold of the latter are nonetheless the same regardless of which strong forcing was chosen.

Lemma 2.6. *The following properties hold of strong and ordinary forcing:*

$$\begin{aligned} & w \leq w' \to w \Vdash^s_b A \to w' \Vdash^s_b A \\ & w \leq w' \to w \Vdash_b A \to w' \Vdash_b A \\ & b \sqsubseteq b' \to w \Vdash^s_b A \to w \Vdash^s_{b'} A \\ & b \sqsubseteq b' \to w \Vdash_b A \to w \Vdash_{b'} A \\ & w \Vdash_b \bot \to Xwb\bot && (run(\cdot)) \\ & w \Vdash^s_b A \to w \Vdash_b A && (return(\cdot)) \\ & (w' : K)(w \leq w' \to w' \Vdash^s_b A \to w' \Vdash_b B) \\ & \qquad \to w \Vdash_b A \to w \Vdash_b B && (bind(\cdot,\cdot)) \end{aligned}$$

Proof. The proofs of monotonicity of strong forcing with respect to $\leq$ and $\sqsubseteq$ are done by induction on the formula A using monotonicity of X. Monotonicity of (non-strong) forcing requires no induction. The proofs of $\mathrm{run}(\cdot), \mathrm{return}(\cdot)$, and $\mathrm{bind}(\cdot,\cdot)$ follow the structure given on Figure 1. $\square$

We will use the same turnstile symbols to denote forcing and strong forcing of *finite lists* of formulas, Γ, defined by,

$$\begin{aligned}
w \Vdash_b \mathrm{nil} &:= \mathrm{Unit}\\
w \Vdash_b \mathrm{cons}\,(A,\Gamma) &:= w \Vdash_b A \times w \Vdash_b \Gamma\\
w \Vdash_b^{\mathsf{s}} \mathrm{nil} &:= \mathrm{Unit}\\
w \Vdash_b^{\mathsf{s}} \mathrm{cons}\,(A,\Gamma) &:= w \Vdash_b^{\mathsf{s}} A \times w \Vdash_b^{\mathsf{s}} \Gamma,
\end{aligned}$$

where $\times$ is the product type (i.e. logical conjunction, when used as a predicate) and Unit is the singleton type. Naturally, the monotonicity properties from the previous lemma extend to forcing and strong forcing for lists.

Theorem 2.7 (Evaluation for call-by-name)**.** *If $p : \Gamma \vdash_b A$, then for any w and any b' such that $b \sqsubseteq b'$ we have that from the finite product $w \Vdash_{b'} \Gamma$ we can construct $w \Vdash_{b'} A$.*

Theorem 2.8 (Evaluation for call-by-value)**.** *If $p : \Gamma \vdash_b A$, then for any w and any b' such that $b \sqsubseteq b'$ we have that from the finite product $w \Vdash_{b'}^{\mathsf{s}} \Gamma$ we can construct $w \Vdash_{b'} A$.*

Proof. The proofs of both theorems are done in continuation-passing style, by using induction on the derivation of p. The program skeletons that corresponds to the proofs can be seen on figures 2 and 3, and the full proofs are available in the Coq formalization. $\square$

2.2 Reifying from the Models

While the evaluation theorems from the previous subsection can be used for any concrete structure that implements the Kripke-CPS models axiomatization, in this section we build one such model, $\mathscr{U}$, the *universal model*, from syntactic elements. It gets its name from the fact that if something is forced in $\mathscr{U}$ then it is also forced in any other possible model.

To obtain a finer grained characterization of the TDPE procedure, we will separate the lambda terms into a level of *normal terms* and a level of *neutral terms* using the following inductive definition.

$$\begin{aligned}
(- \vdash_b^{\mathrm{nf}} -) \ni r &::= \mathsf{lam}(r) \mid \mathsf{inl}(r) \mid \mathsf{inr}(r) \mid \mathsf{shift}(r) \mid e\\
(- \vdash_b^{\mathrm{ne}} -) \ni e &::= \mathsf{app}(e,r) \mid \mathsf{case}(e,r_1,r_2) \mid \mathsf{reset}(e) \mid \mathsf{hyp} \mid \mathsf{wkn}(r)
\end{aligned}$$

This definition concerns *typed* lambda terms (i.e. typing tree derivations), although typing information has been suppressed.

The separation into normal versus neutral terms is standard in the NBE literature, but what is new here is that, in order to obtain the Disjunction Property at the end of this section, $\mathsf{reset}(\cdot)$ has to be neutral.

Definition 2.9 (The model $\mathscr{U}$)**.** The universal Kripke-CPS model $\mathscr{U}$ is built when the set of worlds is the set of contexts Γ,

$$K := \mathrm{List}(\mathrm{Formula}),$$

and the predicate X is defined by recursion on the structure of types of the object language,

$$\begin{aligned} X\Gamma bA &:= \Gamma \vdash^{\mathrm{ne}}_b A \qquad (A - \text{atomic type}) \\ X\Gamma b\bot &:= \Gamma \vdash^{\mathrm{ne}}_b \bot \\ X\Gamma b(A \vee B) &:= \Gamma \vdash^{\mathrm{nf}}_b A \vee B \\ X\Gamma b(A \to B) &:= \Gamma \vdash^{\mathrm{nf}}_b A \to B, \end{aligned}$$

as the set of terms in normal or neutral form of the given type.

The pre-order $\leq$ is defined as the prefix relation on lists. It is not hard to see that reflexivity and transitivity of $\leq$ hold, and that $\leq$-monotonicity and $\sqsubseteq$-monotonicity hold by the weakening properties of the typing system (formal lemmas `proof_nf_mon`, `proof_ne_mon`, `proof_nf_mon2`, and `proof_ne_mon2`). The property meta-reset$()$ is provided by the syntactic reset$()$ rule (formal lemma `X_reset`).

We can now prove that for any meta-level evaluation there exists a term in the object language (*reification* part). Due to contravariance of implication (function types), we need a simultaneous map in the other direction (*reflection* part) [1].

Theorem 2.10 (Reification ($\downarrow$) and reflection ($\uparrow$)). *Given A : Formula, Γ : List(Formula) and b : Bool, the following two statements hold:*

$$\begin{aligned} &\Gamma \Vdash_b A \to \Gamma \vdash^{nf}_b A && \text{“reify”} && ({}^{\Gamma}{\downarrow}^A_b\,(\cdot)) \\ &\Gamma \vdash^{ne}_b A \to \Gamma \Vdash_b A && \text{“reflect”} && ({}^{\Gamma}{\uparrow}^A_b\,(\cdot)) \end{aligned}$$

Proof. The two statements are proved simultaneously, by induction on the type A. The program skeleton corresponding to the proof can be seen on figures 4 and 5. The full proof is done in continuation passing style and is available in the Coq formalization. □

Let reflect(Γ, b) denote the fold-left of the list Γ for the reflection function applied to a variable (hyp), using the unit type constructor tt in the base case. For example, for $\Gamma := \mathrm{cons}\,(A, \mathrm{cons}\,(B, \mathrm{cons}\,(C, \mathrm{nil})))$, we have

$$\begin{aligned} \mathrm{reflect}(\Gamma, b) &: \Gamma \Vdash_b A \times \Gamma \Vdash_b B \times \Gamma \Vdash_b C \times \mathrm{Unit} \\ \mathrm{reflect}(\Gamma, b) &= {}^{\Gamma}{\uparrow}^A_b\,(\mathsf{hyp}), {}^{\Gamma}{\uparrow}^B_b\,(\mathsf{hyp}), {}^{\Gamma}{\uparrow}^C_b\,(\mathsf{hyp}), \mathrm{tt} \end{aligned}$$

We can now obtain the main result of the paper by composing the Evaluation theorems with the Reification theorem, all of which have constructive proofs. In other words, we take a term p, apply a meta-CPS translation $[\![\cdot]\!]$ on it, in an initial environment built from the context Γ by the reflect function, and then reconstruct a term in normal form based on the type A using the reification function $(\downarrow \cdot)$.

Corollary 2.11 (TDPE for call-by-name). *Given $p : \Gamma \vdash_b A$, we have that ${}^{\Gamma}{\downarrow}^A_b\,([\![p]\!]_{reflect(\Gamma,b)}) : \Gamma \vdash^{nf}_b A$.*

Corollary 2.12 (TDPE for call-by-value). *Given $p : \mathrm{nil} \vdash_b A$, we have that ${}^{\mathrm{nil}}{\downarrow}^A_b\,([\![p]\!]_{\mathrm{Unit}}) : \mathrm{nil} \vdash^{nf}_b A$.*

Remark 2.13. The difference in formulation between the two corollaries is due to the fact that the Evaluation theorem for call-by-name (Theorem 2.7) uses ordinary forcing for the context Γ, while the corresponding Theorem 2.8 for call-by-value uses strong forcing. TDPE for CBN can therefore be run on open terms directly, while for CBV we have to have a closed term as input, although TDPE for CBV does normalize below lambda abstractions.

[1] Note that, while reflection and evaluation (theorems 2.7 and 2.8), have the same typing, the first just does eta-expansions by recursion on the object-language type, while the latter is more informative being defined by recursion on the object-language *term*.

$$\begin{aligned}
\mathrm{return}(\cdot) &: - \Vdash^{\mathrm{s}}_{b} A \to - \Vdash_{b} A \\
\mathrm{return}(\alpha) &:= \kappa \mapsto \kappa \cdot \alpha \\
\\
\mathrm{bind}(\cdot,\cdot) &: (- \Vdash^{\mathrm{s}}_{b} A \to - \Vdash_{b} B) \to - \Vdash_{b} A \to - \Vdash_{b} B \\
\mathrm{bind}(\phi,\alpha) &:= \kappa \mapsto \alpha \cdot (\alpha' \mapsto \phi \cdot \alpha' \cdot \kappa) \\
\\
\mathrm{run}(\cdot) &: - \Vdash_{b} \bot \to - \Vdash^{\mathrm{s}}_{b} \bot \\
\mathrm{run}(\alpha) &:= \alpha \cdot (\chi \mapsto \chi)
\end{aligned}$$

Figure 1: Monadic glue functions

The following property shows that the calculus from Table 1 can be considered a constructive logical system, despite the fact that it contains control operators which are usually connected with classical logic. (Classical logic does not have this property)

Proposition 2.14 (Disjunction Property)**.** *If* $p : \mathrm{nil} \vdash_0 A \vee B$ *then from* p *one can get* p' *such that either* $p' : \mathrm{nil} \vdash_0 A$ *or* $p' : \mathrm{nil} \vdash_0 B$*.*

Proof. We can use TDPE to transform p to a term in normal form $r : \mathrm{nil} \vdash^{\mathrm{nf}}_0 A \vee B$. Now, from the syntax of normal and neutral forms, one can see that the only possibilities for r are that it is either a $\mathsf{inl}(r')$ or a $\mathsf{inr}(r')$ – r can not be any of the neutral forms because it does not have a free variable (the context is nil) – and r cannot be a $\mathsf{shift}(\cdot)$ because of the annotation 0 on the turnstile. □

3 Algorithm

In this section we show the algorithmic core of the TDPE procedure. While the exact program in a dependently typed language can be seen with all its gory details in the Coq formalization, our intention here is to give a human readable account of the procedure that we extracted by hand from the Coq formalization. This extraction consists in deleting the dependently typed information which is mostly connected to handling worlds (members of the preorder K) and the associated monotonicity proofs.

We will use two levels of lambda calculus: on one level we will have the "dynamic" lambda terms from Table 1, and on the other "static" level we will use ordinary mathematical function notation: "$\mapsto$" for abstraction, "$\cdot$" for application, ι_1 for injection-left, ι_2 for injection-right, and the usual big-open-curly-bracket for definition by cases. Small Greek letters $\alpha, \beta, \gamma, \phi, \kappa$ are used for static variables; there are no explicit dynamic variables since we use deBruijn indices. The equality symbol ":=" denotes definitional equality.

The monadic glue functions are defined on Figure 1. Parameters corresponding to dependent types for world-handling have been left out (worlds are marked with bars "$-$").

The evaluation algorithms corresponding to theorems 2.8 and 2.7 are given on figures 2 and 3.

The reification algorithms are defined by mutual recursion with reflection algorithms on figures 4 and 5. For facilitating comparison, the places where call-by-value and call-by-name versions differ are marked with boxes.

$$
\begin{aligned}
[\![p : \Gamma \vdash_b A]\!]_{w \Vdash_b \Gamma} &: w \Vdash_b A \\
[\![\mathsf{hyp}]\!]_\rho &:= \mathrm{fst}(\rho) \\
[\![\mathsf{wkn}(p)]\!]_\rho &:= [\![p]\!]_{\mathrm{snd}(\rho)} \\
[\![\mathsf{lam}(p)]\!]_\rho &:= \mathrm{return}(\alpha \mapsto [\![p]\!]_{\alpha,\rho}) \\
[\![\mathsf{app}(p,q)]\!]_\rho &:= \mathrm{bind}(\phi \mapsto \phi \cdot [\![q]\!]_\rho, [\![p]\!]_\rho) \\
[\![\mathsf{inl}(p)]\!]_\rho &:= \mathrm{return}(\iota_1 [\![p]\!]_\rho) \\
[\![\mathsf{inr}(p)]\!]_\rho &:= \mathrm{return}(\iota_2 [\![p]\!]_\rho) \\
[\![\mathsf{case}(p,q,r)]\!]_\rho &:= \mathrm{bind}(\gamma \mapsto \begin{cases} [\![q]\!]_{\alpha,\rho} & \text{, if } \gamma = \iota_1 \alpha \\ [\![r]\!]_{\beta,\rho} & \text{, if } \gamma = \iota_2 \beta \end{cases}, [\![p]\!]_\rho) \\
[\![\mathsf{shift}(p)]\!]_\rho &:= \kappa \mapsto \mathrm{run}([\![p]\!]_{\mathrm{return}(\alpha \mapsto \mathrm{return}(\alpha \cdot \kappa)),\rho}) \\
[\![\mathsf{reset}(p)]\!]^{b=1}_\rho &:= \mathrm{return}(\mathrm{run}([\![p]\!]_\rho)) \\
[\![\mathsf{reset}(p)]\!]^{b=0}_\rho &:= \mathrm{return}(\text{meta-reset}(\mathrm{run}([\![p]\!]_\rho)))
\end{aligned}
$$

Figure 2: Evaluation for call-by-name

$$
\begin{aligned}
[\![p : \Gamma \vdash_b A]\!]_{w \Vdash^s_b \Gamma} &: w \Vdash_b A \\
[\![\mathsf{hyp}]\!]_\rho &:= \mathrm{return}(\mathrm{fst}(\rho)) \\
[\![\mathsf{wkn}(p)]\!]_\rho &:= [\![p]\!]_{\mathrm{snd}(\rho)} \\
[\![\mathsf{lam}(p)]\!]_\rho &:= \mathrm{return}(\alpha \mapsto [\![p]\!]_{\alpha,\rho}) \\
[\![\mathsf{app}(p,q)]\!]_\rho &:= \mathrm{bind}(\phi \mapsto \mathrm{bind}(\phi, [\![q]\!]_\rho), [\![p]\!]_\rho) \\
[\![\mathsf{inl}(p)]\!]_\rho &:= \mathrm{bind}(\alpha \mapsto \mathrm{return}(\iota_1 \alpha), [\![p]\!]_\rho) \\
[\![\mathsf{inr}(p)]\!]_\rho &:= \mathrm{bind}(\alpha \mapsto \mathrm{return}(\iota_2 \alpha), [\![p]\!]_\rho) \\
[\![\mathsf{case}(p,q,r)]\!]_\rho &:= \mathrm{bind}(\gamma \mapsto \begin{cases} [\![q]\!]_{\alpha,\rho} & \text{, if } \gamma = \iota_1 \alpha \\ [\![r]\!]_{\beta,\rho} & \text{, if } \gamma = \iota_2 \beta \end{cases}, [\![p]\!]_\rho) \\
[\![\mathsf{shift}(p)]\!]_\rho &:= \kappa \mapsto \mathrm{run}([\![p]\!]_{\alpha \mapsto \mathrm{return}(\kappa \cdot \alpha),\rho}) \\
[\![\mathsf{reset}(p)]\!]^{b=1}_\rho &:= \mathrm{return}(\mathrm{run}([\![p]\!]_\rho)) \\
[\![\mathsf{reset}(p)]\!]^{b=0}_\rho &:= \mathrm{return}(\text{meta-reset}(\mathrm{run}([\![p]\!]_\rho)))
\end{aligned}
$$

Figure 3: Evaluation for call-by-value

$$
\begin{array}{rll}
{}^{\Gamma}{\downarrow}^{A}_{b}(\cdot) & : \Gamma \Vdash_b A \to \Gamma \vdash^{\mathrm{nf}}_b A & \\
{}^{\Gamma}{\downarrow}^{\bot}_{b}(\alpha) & := \mathrm{run}(\alpha) & \\
{}^{\Gamma}{\downarrow}^{A_0}_{0}(\alpha) & := \mathrm{run}(\alpha) & \text{for atomic } A_0 \neq \bot \\
{}^{\Gamma}{\downarrow}^{A_0}_{1}(\alpha) & := \mathsf{shift}(\alpha \cdot (\chi \mapsto \mathsf{app}(\mathsf{hyp}, \chi))) & \text{for atomic } A_0 \neq \bot \\
{}^{\Gamma}{\downarrow}^{A\to B}_{b}(\alpha) & := \mathsf{lam}({}^{\Gamma}{\downarrow}^{B}_{b}(\kappa \mapsto {}^{A,\Gamma}{\uparrow}^{A}_{b}(\mathsf{hyp}) \cdot (\alpha' \mapsto \alpha \cdot (\phi \mapsto \phi \cdot (\mathrm{return}(\alpha')) \cdot \kappa)))) & \\
{}^{\Gamma}{\downarrow}^{A\vee B}_{0}(\alpha) & := \alpha \cdot \left(\gamma \mapsto \begin{cases} \mathsf{inl}({}^{\Gamma}{\downarrow}^{A}_{0}(\beta)) & \text{, if } \gamma = \iota_1 \beta \\ \mathsf{inr}({}^{\Gamma}{\downarrow}^{B}_{0}(\beta)) & \text{, if } \gamma = \iota_2 \beta \end{cases} \right) & \\
{}^{\Gamma}{\downarrow}^{A\vee B}_{1}(\alpha) & := \mathsf{shift}\left(\alpha \cdot \left(\gamma \mapsto \begin{cases} \mathsf{app}(\mathsf{hyp}, \mathsf{inl}({}^{\Gamma}{\downarrow}^{A}_{1}(\beta))) & \text{, if } \gamma = \iota_1 \beta \\ \mathsf{app}(\mathsf{hyp}, \mathsf{inr}({}^{\Gamma}{\downarrow}^{B}_{1}(\beta))) & \text{, if } \gamma = \iota_2 \beta \end{cases} \right)\right) & \\
 & & \\
{}^{\Gamma}{\uparrow}^{A}_{b}(\cdot) & : \Gamma \vdash^{\mathrm{ne}}_b A \to \Gamma \Vdash_b A & \\
{}^{\Gamma}{\uparrow}^{A_0}_{b}(e) & := \mathrm{return}(e) & \text{for atomic } A_0 \\
{}^{\Gamma}{\uparrow}^{A\to B}_{b}(e) & := \mathrm{return}(\alpha \mapsto {}^{\Gamma}{\uparrow}^{B}_{b}(\mathsf{app}(e, {}^{\Gamma}{\downarrow}^{A}_{b}(\alpha)))) & \\
{}^{\Gamma}{\uparrow}^{A\vee B}_{b}(e) & := \kappa \mapsto \mathsf{case}(e, {}^{A,\Gamma}{\uparrow}^{A}_{b}(\mathsf{hyp}) \cdot (\alpha \mapsto \kappa \cdot \iota_1 \mathrm{return}(\alpha)), {}^{B,\Gamma}{\uparrow}^{B}_{b}(\mathsf{hyp}) \cdot (\alpha \mapsto \kappa \cdot \iota_2 \mathrm{return}(\alpha))) &
\end{array}
$$

Figure 4: Reification and reflection for call-by-name

$$
\begin{array}{rll}
{}^{\Gamma}{\downarrow}^{A}_{b}(\cdot) & : \Gamma \Vdash_b A \to \Gamma \vdash^{\mathrm{nf}}_b A & \\
{}^{\Gamma}{\downarrow}^{\bot}_{b}(\alpha) & := \mathrm{run}(\alpha) & \\
{}^{\Gamma}{\downarrow}^{A_0}_{0}(\alpha) & := \mathrm{run}(\alpha) & \text{for atomic } A_0 \neq \bot \\
{}^{\Gamma}{\downarrow}^{A_0}_{1}(\alpha) & := \mathsf{shift}(\alpha \cdot (\chi \mapsto \mathsf{app}(\mathsf{hyp}, \chi))) & \text{for atomic } A_0 \neq \bot \\
{}^{\Gamma}{\downarrow}^{A\to B}_{b}(\alpha) & := \mathsf{lam}({}^{\Gamma}{\downarrow}^{B}_{b}(\kappa \mapsto {}^{A,\Gamma}{\uparrow}^{A}_{b}(\mathsf{hyp}) \cdot (\alpha' \mapsto \alpha \cdot (\phi \mapsto \phi \cdot (\boxed{\alpha'}) \cdot \kappa)))) & \\
{}^{\Gamma}{\downarrow}^{A\vee B}_{0}(\alpha) & := \alpha \cdot \left(\gamma \mapsto \begin{cases} \mathsf{inl}({}^{\Gamma}{\downarrow}^{A}_{0}(\boxed{\mathrm{return}(\beta)})) & \text{, if } \gamma = \iota_1 \beta \\ \mathsf{inr}({}^{\Gamma}{\downarrow}^{B}_{0}(\boxed{\mathrm{return}(\beta)})) & \text{, if } \gamma = \iota_2 \beta \end{cases} \right) & \\
{}^{\Gamma}{\downarrow}^{A\vee B}_{1}(\alpha) & := \mathsf{shift}\left(\alpha \cdot \left(\gamma \mapsto \begin{cases} \mathsf{app}(\mathsf{hyp}, \mathsf{inl}({}^{\Gamma}{\downarrow}^{A}_{1}(\boxed{\mathrm{return}(\beta)}))) & \text{, if } \gamma = \iota_1 \beta \\ \mathsf{app}(\mathsf{hyp}, \mathsf{inr}({}^{\Gamma}{\downarrow}^{B}_{1}(\boxed{\mathrm{return}(\beta)}))) & \text{, if } \gamma = \iota_2 \beta \end{cases} \right)\right) & \\
 & & \\
{}^{\Gamma}{\uparrow}^{A}_{b}(\cdot) & : \Gamma \vdash^{\mathrm{ne}}_b A \to \Gamma \Vdash_b A & \\
{}^{\Gamma}{\uparrow}^{A_0}_{b}(e) & := \mathrm{return}(e) & \text{for atomic } A_0 \\
{}^{\Gamma}{\uparrow}^{A\to B}_{b}(e) & := \mathrm{return}(\alpha \mapsto {}^{\Gamma}{\uparrow}^{B}_{b}(\mathsf{app}(e, {}^{\Gamma}{\downarrow}^{A}_{b}(\boxed{\mathrm{return}(\alpha)})))) & \\
{}^{\Gamma}{\uparrow}^{A\vee B}_{b}(e) & := \kappa \mapsto \mathsf{case}(e, {}^{A,\Gamma}{\uparrow}^{A}_{b}(\mathsf{hyp}) \cdot (\alpha \mapsto \kappa \cdot \iota_1 \boxed{\alpha}), {}^{B,\Gamma}{\uparrow}^{B}_{b}(\mathsf{hyp}) \cdot (\alpha \mapsto \kappa \cdot \iota_2 \boxed{\alpha})) &
\end{array}
$$

Figure 5: Reification and reflection for call-by-value

3.1 Known Equational Theories

Before considering computational tests, we recall the available equational theories for shift and reset.

The equational theory for call-by-value shift and reset, for the full hierarchy, has been proven sound and complete with respect to the extended CPS translation [8] by Kameyama [15]. Considering the first level of the hierarchy which is of interest here, the equations are expressed using the classes of *values* (V) and *pure evaluation contexts* (F),

$$V ::= x \mid \lambda x.p \qquad\qquad F ::= [] \mid F\,p \mid VF,$$

as follows:

$$
\begin{aligned}
(\lambda x.p)V &= p\{V/x\} && && (1)\\
\lambda x.Vx &= V && \text{when } x \notin \mathrm{FV}(V) && (2)\\
(\lambda x.F[x])p &= F[p] && \text{when } x \notin \mathrm{FV}(F) && (3)\\
\langle V\rangle &= V && && (4)\\
\langle(\lambda x.p)\langle q\rangle\rangle &= (\lambda x.\langle p\rangle)\langle q\rangle && && (5)\\
\mathscr{S}k.\langle p\rangle &= \mathscr{S}k.p && && (6)\\
\mathscr{S}k.k\langle p\rangle &= \langle p\rangle && \text{when } k \notin \mathrm{FV}(p) && (7)\\
\mathscr{S}k.kp &= p && \text{when } k \notin \mathrm{FV}(p) && (8)\\
\langle F[\mathscr{S}k.p]\rangle &= \langle p\{(\lambda x.\langle F[x]\rangle)\ /\ k\}\rangle && \text{when } x \notin \mathrm{FV}(F)\cup\{k\} && (9)
\end{aligned}
$$

The equational theory for call-by-name shift and reset, for the first level of the hierarchy, has been studied by Kameyama and Tanaka [16]. For the purpose of proving soundness and completeness with respect to Biernacka and Biernacki's [4] call-by-name CPS semantics for shift and reset, Kameyama and Tanaka distinguish between two kinds of term applications, the usual one, and the one to continuation variables ($\hookleftarrow$); and two kinds of substitutions, for normal variables ($\{\cdot/x\}$), and for continuation variables ($\{k \Rightarrow \cdot\}$). The classes of values and pure evaluation contexts are restrictions of the call-by-name ones, given by:[2]

$$U ::= \lambda x.p \qquad\qquad E ::= [] \mid E\,p$$

The equational theory is as follows,

$$
\begin{aligned}
(\lambda x.p)q &= p\{q/x\} && && (10)\\
\langle U\rangle &= U && && (11)\\
k' \hookleftarrow E[\mathscr{S}k.p] &= \langle p\{k \Rightarrow (k' \hookleftarrow E)\}\rangle && && (12)\\
\mathscr{S}k.\langle p\rangle &= \mathscr{S}k.p && && (13)\\
\mathscr{S}k.k \hookleftarrow p &= p && \text{when } k \notin \mathrm{FV}(p) && (14)\\
\langle E[\mathscr{S}k.p]\rangle &= \langle p\{k \Rightarrow E\}\rangle, && && (15)
\end{aligned}
$$

where the substitution $q\{k \Rightarrow \cdot\}$ is defined by recursive descent on the term q and affects only the subterms of the form $k \hookleftarrow p$ by:

$$
\begin{aligned}
(k' \hookleftarrow p)\{k \Rightarrow E\} &= \langle E[p\{k \Rightarrow E\}]\rangle && \text{when } k' = k\\
(k' \hookleftarrow p)\{k \Rightarrow E\} &= k' \hookleftarrow (p\{k \Rightarrow E\}) && \text{when } k' \neq k
\end{aligned}
$$

[2]Kameyama and Tanaka also consider constants c among the call-by-name values, however no variables are allowed. Since we do not have constants in our minimal object-language of study, we did not include them as an option of U.

We have used conventional syntax, writing $\mathsf{reset}(p)$ as $\langle p\rangle$, and $\mathsf{shift}(p)$ as $\mathscr{S}k.p$, and will continue to do so in the next subsection.

3.2 Example Runs of the Algorithm

Let us now consider some test-runs of our TDPE procedure. Each example consists of an input term, marked with a number to refer to, and two outputs: using TDPE for call-by-value (CBV) and for call-by-name (CBN).

We begin with simple examples where the continuation variable of shift is not used (exceptions effect).

$$\lambda x.\langle(\lambda y.y)(\mathscr{S}k.x)\rangle \tag{16}$$
$$\lambda x.\langle x\rangle \tag{CBV}$$
$$\lambda x.\langle\langle x\rangle\rangle \tag{CBN}$$

$$\lambda x.\langle\langle\langle(\lambda y.y)(\mathscr{S}k.x)\rangle\rangle\rangle \tag{17}$$
$$\lambda x.\langle x\rangle \tag{CBV}$$
$$\lambda x.\langle\langle x\rangle\rangle \tag{CBN}$$

The CBN normal forms are not perfect as the resets are systematically duplicated at top level. This duplication is not related to the number of reset as input, as can be seen from Example (17), but to a "bug" in the Coq formalization. Namely, the lemma `Kont_sforces_mon2'`, proving monotonicity of non-strong forcing with respect to the Boolean order, uses a reset in the proof, and, since this lemma is not used in the CBV case, the problem does not appear there.[3]

The CBV equational theory can derive the TDPE output for examples (16) and (17). The CBN equational theory derives $\lambda x.\langle x\rangle$ but not $\lambda x.\langle\langle x\rangle\rangle$. However, our TDPE for CBN identifies the two, because it also normalizes $\lambda x.\langle x\rangle$ to $\lambda x.\langle\langle x\rangle\rangle$.

The next example does not use a control operator, but has a delimiter.

$$\lambda xy.\langle\langle xy\rangle\rangle \tag{18}$$
$$\lambda xy.\langle xy\rangle \tag{CBV}$$
$$\lambda xy.\langle\langle x\langle y\rangle\rangle\rangle \tag{CBN}$$

The CBV and CBN equational theories do not transform Example (18) further, because the subterm xy is not a value. TDPE for CBV removes one delimiter, as if xy were a value, and TDPE for CBN delimits the inside variable y, as if it were taking into account that variables are not values according to the CBN equational theory.

Let us consider an example that uses the continuation inside a shift.

$$\lambda xy.\langle x(\mathscr{S}k.k(ky))\rangle \tag{19}$$
$$\lambda xy.\langle x(xy)\rangle \tag{CBV}$$
$$\lambda xy.\langle\langle x\langle y\rangle\rangle\rangle \tag{CBN}$$

[3]The solution might be to make the non-strong forcing monad monotone also for the $\sqsubseteq$ relation and not only for $\leq$ on worlds, and is the subject of future work.

Starting from Example (19), the CBV equational theory can obtain the term $\lambda xy.\langle(\lambda a.\langle xa\rangle)\big((\lambda a.\langle xa\rangle)y\big)\rangle$, and then also the term $\lambda xy.\langle(\lambda a.\langle xa\rangle)\langle xy\rangle\rangle$, however, no further rewriting is possible using that theory, because neither is $\langle x[]\rangle$ a pure evaluation context nor is $\langle xy\rangle$ a value. As for the CBN equational theory, it can not rewrite the starting term (19), because there are nested applications to the continuation variable $k \hookleftarrow (k \hookleftarrow y)$ and, unlike in the CBV case, $x[]$ is not a pure evaluation context in CBN. Note that: 1) there is no x missing in the output of CBN TDPE; 2) the term $\lambda xy.\langle(\lambda a.\langle xa\rangle)\langle xy\rangle\rangle$ obtained by CBV equations is normalized by the CBV TDPE procedure to the same thing as term (19) (see `nbe_tests.v` from the implementation), meaning that TDPE knows how to further reduce those "blocked" terms.

The following example is very similar to the previous one, hence we will not comment on it much, but its purpose is to show that CBN TDPE can duplicate the variable x if needed.

$$\lambda xy.\langle x\langle x(\mathscr{S}k.k(ky))\rangle\rangle \tag{20}$$
$$\lambda xy.\langle x(x(xy))\rangle \tag{CBV}$$
$$\lambda xy.\langle\langle x\langle x\langle y\rangle\rangle\rangle\rangle \tag{CBN}$$

The next example contains an evaluation context and a form of shift that can be used with the equational theory for CBN.

$$\lambda xy.\langle(\mathscr{S}k.ky)x\rangle \tag{21}$$
$$\lambda xy.\langle yx\rangle \tag{CBV}$$
$$\lambda xy.\langle\langle y\langle x\rangle\rangle\rangle \tag{CBN}$$

The CBV equational theory rewrites (21) to $\lambda xy.\langle\langle yx\rangle\rangle$, and the CBN one rewrites (21) to $\lambda xy.\langle yx\rangle$. If we apply TDPE on these results of rewriting, we get the same output as the TDPE for (21).

In the following example, two shifts interact inside the same reset.

$$\lambda xyz.\langle\big(\mathscr{S}k.y(kz)\big)\big(\mathscr{S}k'.z(k'x)\big)\rangle \tag{22}$$
$$\lambda xyz.\langle y(z(zx))\rangle \tag{CBV}$$
$$\lambda xyz.\langle\langle y\langle z\langle z\langle x\rangle\rangle\rangle\rangle\rangle \tag{CBN}$$

The CBV equational theory can transform (22) to $\lambda xyz.\langle y\langle z\langle zx\rangle\rangle\rangle$ which can in turn be transformed by CBV TDPE to the same result as for (22). The CBN equational theory can rewrite the left occurrence of shift and obtain $\lambda xyz.\langle y\langle z\big(\mathscr{S}k'.z(k'x)\big)\rangle\rangle$, but no further rewriting is possible because $z[]$ is not a pure evaluation context in CBN; nevertheless, $\lambda xyz.\langle y\langle z\big(\mathscr{S}k'.z(k'x)\big)\rangle\rangle$ can further be transformed by CBN TDPE to the same output as for (22).

We consider an example that involves sum types, but briefly, since we do not have a ready made equational theory to compare the output to.

$$\lambda xy.\langle\text{case } x \text{ of } (\lambda z.\mathscr{S}k.kz \parallel \lambda z.z)\rangle \tag{23}$$
$$\lambda xy.\text{case } x \text{ of } (\lambda z.\langle z\rangle \parallel \lambda z.\langle z\rangle) \tag{CBV}$$
$$\lambda xy.\text{case } x \text{ of } (\lambda z.\langle\langle\langle z\rangle\rangle\rangle \parallel \lambda z.\langle\langle\langle z\rangle\rangle\rangle) \tag{CBN}$$

We see that not only the reset is pushed from the front of the case-expression into its branches.

4 Discussion and Related Work

The CPS translation that we use for CBV TDPE is exactly[4] the standard (non-extended) CBV CPS translation of Danvy and Filinski [8], known also as 1-CPS. Terms in 1-CPS arising from shift are not evaluation-order independent when executed in regular functional programming languages, and that is why, to fix the semantics of shift and reset regardless of the target language of CPS, an additional CBV CPS translation of the CPS result is usually performed, and this composition of two CPS translations is known as the extended CPS, or 2-CPS. It is with respect to this 2-CPS that Kameyama [15] proved the equational theory for CBV to be sound and complete.

The CPS translation used for CBN TDPE is *not* the available 1-CPS of Biernacka and Biernacki [4]. The difference is in the shift rule (see Figure 2), as Biernacka and Biernacki's $[\![\mathsf{shift}(p)]\!]_\rho := \kappa \mapsto \mathrm{run}([\![p]\!]_{\kappa,\rho})$ would not type check in our type theoretic model. The standard 2-CPS translation, that Kameyama and Tanaka [16] proved their equational theory for CBN sound and complete for, is obtained by performing a *CBV* translation of the 1-CPS CBN translation.

We profit from our implementation language having strong reduction[5] in that we do not have to apply two passes of CPS. That is, 1-CPS is sufficient because our evaluation (CPS translation) of a term at the meta-level is a typed and closed term which reduces to the same normal form regardless of the reduction strategy. The typed CPS-s used by Kameyama and Tanaka need recursive types, while we do not. On the other hand, we do not know if it possible at all to account for constants defined by general recursion in our model [6]. The question, therefore, of whether our TDPE could be useful in practice is open. We certainly find it useful when "practice" concerns lambda calculi for Logic and proof assistants.

We saw in Subsection 3.2 that some terms that cannot be further rewritten by the equational theories, can be further normalized by the TDPE. On the other hand, the equational theories have been proven to be sound and complete with respect to the CPS translation, that is, an equation holds between two terms if and only if the two terms have β-η-equal CPS translations. That means that the extra "rewriting" done by the TDPE somehow extends the equality of CPS translations – indeed, the outputs of TDPE for the examples of Subsection 3.2 do *not* have CPS translations β-η-equal with the ones of the inputs. Nevertheless, at least for all the examples that we have tested, the TDPE identifies the original terms with the "intermediary" results arrived to by the equational theories.

As for the typing system we use, we note that it is Filinski's system [10], which is sufficient for representing monadic effects by delimited control. A difference with that system is that there is an annotation b on the turnstile ($\vdash_b$) whose purpose is to not allow shifts appearing outside the delimited. This could have also been guaranteed by an external syntactic criteria on whole terms, but the calculus is easier to model if all information is already present in the typing system. The more general typing system for shift and reset, with answer type modification, can type check more programs, but with the price of a function being able to modify its own answer type that we are not ready to pay. In particular, the modified meening of implication would not immediately correspond to something well known on the side of Logic.

[4]There are additional typing annotations concerning worlds attached to the continuations at the type theoretic level. Another subtle point is that, when evaluating reset (figures 2 and 3), the type theoretic model predicates when to insert a syntactic reset between the return and the run. One may insert a reset in the other cases as well, if one wants to obtain normal forms with more resets, but we prefer to not do it since it is not mandated by the model.

[5]Constructive type theory is strongly normalizing and there is a simple and efficient implementation of a virtual machine for strong reduction [11].

[6]We have however, in separate work, extended the model to higher type primitive recursion (Godel's System T) plus shift and reset on numeric types.

Our work was developed independently of the results of the previous work on TDPE for CBV shift and reset of Tsushima and Asai [18], which seems to derive from their preceding works on traditional offline and online partial evaluation for shift and reset [1, 2]. The difference between theirs and our results seems to be that: 1) they treat a more general typing system (function types with answer type modification); 2) we aim to produce normal forms that eliminate as many shifts and resets as possible: for example, during reification for function types, Tsushima and Asai's TDPE constructs a $\mathsf{shift}(\cdot)$ immediately after the first $\mathsf{lam}(\cdot)$, whereas we postpone the construction of $\mathsf{shift}(\cdot)$ to some cases of reification that will subsequently called.

Acknowledgements

I thank the anonymous referees for pointing out problematic parts that led to improvement of the paper.

References

[1] Kenichi Asai (2002): *Online partial evaluation for shift and reset*. In Peter Thiemann, editor: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02), Portland, Oregon, USA, January 14-15, 2002*, ACM, pp. 19–30, doi:10.1145/503032.503034.

[2] Kenichi Asai (2004): *Offline partial evaluation for shift and reset*. In Nevin Heintze & Peter Sestoft, editors: *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulatiog, 2004, Verona, Italy, August 24-25, 2004*, ACM, pp. 3–14, doi:10.1145/1014007.1014009.

[3] Ulrich Berger & Helmut Schwichtenberg (1991): *An Inverse of the Evaluation Functional for Typed lambda-calculus*. In: *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science, 15-18 July, 1991, Amsterdam, The Netherlands*, IEEE Computer Society, pp. 203–211, doi:10.1109/LICS.1991.151645.

[4] Malgorzata Biernacka & Dariusz Biernacki (2009): *Context-based proofs of termination for typed delimited-control operators*. In: *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, ACM, New York, NY, USA, pp. 289–300, doi:10.1145/1599410.1599446.

[5] Catarina Coquand (1993): *From Semantics to Rules: A Machine Assisted Analysis*. In: *CSL '93*, *Lecture Notes in Computer Science* 832, Springer, pp. 91–105, doi:10.1007/BFb0049326.

[6] Olivier Danvy (1999): *Type-Directed Partial Evaluation*. In John Hatcliff, Torben Mogensen & Peter Thiemann, editors: *Partial Evaluation*, *Lecture Notes in Computer Science* 1706, Springer Berlin / Heidelberg, pp. 367–411, doi:10.1007/3-540-47018-2_16.

[7] Olivier Danvy & Andrzej Filinski (1989): *A Functional Abstraction of Typed Contexts*. Technical Report, Computer Science Department, University of Copenhagen. DIKU Rapport 89/12.

[8] Olivier Danvy & Andrzej Filinski (1990): *Abstracting Control*. In: *LISP and Functional Programming*, pp. 151–160, doi:10.1145/91556.91622.

[9] Olivier Danvy & Andrzej Filinski (1992): *Representing Control: A Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2(4), pp. 361–391, doi:10.1017/S0960129500001535.

[10] Andrzej Filinski (1996): *Controlling Effects*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Technical Report CMU-CS-96-119 (144pp.).

[11] Benjamin Grégoire & Xavier Leroy (2002): *A compiled implementation of strong reduction*. *SIGPLAN Not.* 37(9), pp. 235–246, doi:10.1145/583852.581501.

[12] Danko Ilik (2012): *Delimited control operators prove Double-negation Shift*. *Annals of Pure and Applied Logic* 163(11), pp. 1549 – 1559, doi:10.1016/j.apal.2011.12.008.

[13] Danko Ilik (2013): *Continuation-passing style models complete for intuitionistic logic*. *Annals of Pure and Applied Logic* 164(6), pp. 651 – 663, doi:`10.1016/j.apal.2012.05.003`.

[14] Danko Ilik, Gyesik Lee & Hugo Herbelin (2010): *Kripke models for classical logic*. *Annals of Pure and Applied Logic* 161(11), pp. 1367 – 1378, doi:`10.1016/j.apal.2010.04.007`. Special Issue: Classical Logic and Computation (2008).

[15] Yukiyoshi Kameyama (2007): *Axioms for control operators in the CPS hierarchy*. *Higher Order Symbol. Comput.* 20(4), pp. 339–369, doi:`10.1007/s10990-007-9009-x`.

[16] Yukiyoshi Kameyama & Asami Tanaka (2010): *Equational axiomatization of call-by-name delimited control*. In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, ACM, New York, NY, USA, pp. 77–86, doi:`10.1145/1836089.1836100`.

[17] A. S. Troelstra & D. van Dalen (1988): *Constructivism in mathematics. Vol. I*. *Studies in Logic and the Foundations of Mathematics* 121, North-Holland Publishing Co., Amsterdam, doi:`10.1016/S0049-237X(09)70523-3`. An introduction.

[18] Kanae Tsushima & Kenichi Asai (2009): *Towards Type-Directed Partial Evaluation for Shift and Reset*. In: *Informal proceedings of the 2009 Workshop on Normalization by Evaluation*, Los Angeles, California, pp. 57–64.