情報処理学会論文誌　プログラミング　Vol.6　No.4　50–64 (Dec. 2013)

# 限定継続命令shift/reset付き型主導部分評価器の抽出

廣田 知子[1,a)]　浅井 健一[1,b)]

受付日 2013年4月12日, 採録日 2013年9月2日

**概要**：Danvy により提案された型主導部分評価器（TDPE）は，与えられた項を正規形に評価する関数であり，Filinski によって call-by-name および call-by-value の単純型付き λ 計算における TDPE の正当性が示されている．一方，Tsushima らは，限定継続命令 shift/reset 付きに拡張された call-by-value の λ 計算に対する TDPE を shift/reset を使って実装している．しかし，その背景にある理論については明らかになっておらず，直感的な説明と実装が与えられているのみであった．本論文では，Tsushima らによって提案された直接形式の shift/reset 付き TDPE を継続渡し形式の shift/reset 付き TDPE に変換し，それが Kripke モデルの推論規則に対する completeness の証明と対応することを示す．この Kripke モデルの特徴は，構成要素である二項関係が継続渡し形式で定義される点にある．Curry Howard 同型の観点において，この Kripke モデルに対する completeness の性質と TDPE が対応関係にあるため，completeness の定理を定理証明系 Coq で定式化することにより，TDPE プログラムを抽出することができる．得られたプログラムは，継続渡し形式の shift/reset 付き TDPE と同じ構造をしていることが見て取れる．

**キーワード**：型主導部分評価器，限定継続命令 shift/reset，Kripke モデル，CPS 変換，定理証明系 Coq

# Extraction of Type-directed Partial Evaluator for Delimited Control Operators Shift and Reset

Noriko Hirota[1,a)]　Kenichi Asai[1,b)]

Received: April 12, 2013, Accepted: September 2, 2013

***Abstract:*** Danvy proposed the type-directed partial evaluator (TDPE for short) that reduces an input term to a normal form. The correctness properties of both call-by-name TDPE and call-by-value TDPE for simply-typed lambda calculus have been proved by Filinski. Subsequently, Tsushima et al. showed a TDPE for the call-by-value lambda calculus with delimited control operators, shift and reset, implemented in terms of shift/reset. However, its underlying theory is not given except for intuitive explanation and implementation. The aim of this paper is to transform the above direct-style definition of TDPE to continuation-passing style (CPS) and to show that this CPS TDPE corresponds to the completeness property of a Kripke model. The binary relation of the Kripke model is defined in a CPS manner. The model is then shown to be complete with respect to normal form construction. Since the completeness of the Kripke model exactly corresponds to the TDPE via Curry Howard isomorphism, we can extract TDPE from the proof of completeness. We have formalized the development in the proof assistant Coq. We observe that the obtained TDPE holds the same structure as the CPS TDPE for shift/reset.

***Keywords:*** type-directed partial evaluator, delimited control operator shift and reset, Kripke model, CPS transformation, proof assistant Coq

## 1. はじめに

型主導部分評価器（type-directed partial evaluator，以下 TDPE と略す）は，Danvy [7] により提案された部分評価器であり，term とその term の型が渡されたとき，その

[1] お茶の水女子大学
Ochanomizu University, Bunkyo, Tokyo 112–8610, Japan
[a)] hirota.noriko@is.ocha.ac.jp
[b)] asai@is.ocha.ac.jp

型によって場合分けを行いながら，term を normal form へと評価する関数である．TDPE の利点として，term の中身を見ないがゆえに，term の構造における場合分け等を行わずに部分評価を行うため，高速であることがあげられる．call-by-name および call-by-value の単純型付き $\lambda$ 計算における TDPE においては，その正当性が Filinski [10] によりすでに示されている．Filinski はその論文で，論理関係を用いた Tait 流の証明を行っている．call-by-value だけでなく，任意の monadic effect を持つ体系を対象としており，計算順序を保存するための let-insertion を含めて証明がなされている．しかし，他の体系，特に shift/reset が入った場合にどうなるかを考えようとしても，継続の返す型（アンサタイプ）が変化するような体系は monad で表せないため，そのまま拡張しようとするのは難しい．

Filinski の仕事とは独立に，Tsushima ら [19] は shift/reset を含むプログラムに対する TDPE を shift/reset を使って実装している．しかし，提案されている TDPE については，直感的な説明と実装が示されているのみであった．

一方，Ilik [12], [13], [14] は Kripke モデルを用いた Coquand の研究 [4] をもとに，Kripke モデルの推論規則に対する完全性の証明から TDPE を抽出する方法を示し，定理証明系 Coq で定式化を行っている．Ilik は shift/reset も扱っているが，ここで扱われている限定継続は論理の立場から見たもので，我々が考える通常の限定継続とは異なり，継続の返す型がつねに固定されてしまっているのに加え，reset の持ちうる型が atomic type のみに限定されている．

本論文では，Tsushima らの仕事と Ilik の仕事を結び付け，通常の限定継続を扱える形で Ilik の証明を再構築する．まず，Tsushima らの shift/reset 用の TDPE を継続渡し形式（continuation-passing style，以下 CPS）に変換する．次に，適切な Kripke モデルを構築し，そのモデルに対する completeness の定理を証明する．completeness の定理は，TDPE の中核である reify/reflect の関数と Curry Howard 同型になっており，completeness の証明を抽出することで TDPE が得られる格好となる．証明はすべて Coq で定式化されており，Tsushima らの TDPE に直接，対応する関数が得られている．shift/reset を扱うために CPS 変換を行うのは常套手段だが，TDPE は static/dynamic な term が混ざった式で定義されるため，元の直接形式の TDPE をどのように CPS 変換すればよいのかは自明ではなかった．得られた TDPE はとてもすっきりした形をしており，TDPE と completeness の定理のさらなる理解につながると思われる．これは，証明が複雑になり，TDPE の抽出は手動で行わなくてはならなかった Ilik の証明とは対照的である．

本論文の構成は次のとおり．まず 2 章で限定継続命令 shift/reset の説明を，3 章で Kripke モデルの説明を行う．次に 4 章で call-by-name の TDPE の抽出手法を定式化し，抽出された TDPE の考察を行う．そして 5 章で，4 章の手法を CPS へと拡張することによって call-by-value の TDPE を抽出し，さらにもう一度 CPS 変換することによって call-by-value の shift/reset 付き TDPE を 6 章で抽出する．7 章で関連研究に触れ，8 章で本研究をまとめる．

以降，本論文で定義する call-by-name の shift/reset なし $\lambda$ 計算を $\lambda_{cbn}$，call-by-value の shift/reset なし $\lambda$ 計算を $\lambda_{cbv}$，そして call-by-value の shift/reset 付 $\lambda$ 計算を $\lambda_{cbv}^{S/R}$ と表記する．

## 2. 限定継続命令 shift/reset

shift/reset は Danvy ら [6] によって提案された継続を扱うための命令であり，これらを使ってプログラムの例外処理等を実現することができる．継続はいわば現在の計算が終了した後にする仕事であり，shift は現在の継続を取得し，reset は取得する継続の範囲を限定する命令である．以下に OchaCaml [17] による実行例を示す．ここで使用する関数 `shift(fun k -> M)` は受け取った継続を束縛変数 `k` に渡して `M` を実行する．また，`reset(fun () -> M)` は継続の範囲を `M` に限定する．

```
1 + reset(fun () ->
          4 + shift(fun k -> 3 * (k 2)))
↝ 1 + reset(fun () -> 3 * (4 + 2))
↝ 1 + reset(fun () -> 18)
↝ 1 + 18
↝ 19
```

まず，1 行目の `reset` によって継続が `(4 + [])` に限定され，この継続が `shift` で束縛されている変数 `k` に代入され，2 行目の式となる．そして 2 行目の `reset` 内の式が簡約されて `reset(fun () -> 18)` が得られる．この式はそのまま `18` を返すため，最終的に得られる結果は `19` である．

## 3. Kripke モデルと TDPE 抽出の概要

本論文では，命題論理における Kripke モデルに対する完全性の証明から TDPE を抽出するというアプローチをとっている．ここでは本論文で必要な Kripke モデルを導入する．

**定義 1**　Kripke モデルは以下の 2 つからなる．

- $(K, \leq)$ という可能世界の集合 $K$ 上の前順序 $\leq$（反射律と推移律が成り立つ順序関係）．
- 可能世界 $w$ と命題変数 $X$ の間の forcing と呼ばれる関係 $w \models X$．ここで，この関係は可能世界について monotone であるという条件を満たさなくてはならない．つまり，$w \models X$ かつ $w \leq w'$ なら $w' \models X$ でなくてはならない．

（通常の述語論理に対する Kripke 意味論では，以上に加えて個体変数の動く範囲を規定する集合が存在するが，本論文では命題論理の範囲しか扱わないので，不要である．）

命題変数についての forcing 関係は，「ならば」の意味に対して自然に拡張される．たとえば通常の直接形式の意味を考えるときには以下のように定義される．

$w \models A \to B \Leftrightarrow$ 任意の $w' \geq w$ に対して，$w' \models A$ ならば $w' \models B$.

本論文では，さらに CPS プログラムの意味を反映した定義も使用する．

上記の Kripke のモデルをここでは次のように使う．まず，可能世界 $w$ としては，型付き $\lambda$ 計算における自由変数の型（＝論理式）の列とする（本論文では，以下，この型の列 $w$ を環境と呼ぶことにする）．また，可能世界の間の前順序は環境の拡張と定義する．そのうえで，$w \models X$ は，型変数（＝命題変数）$X$ が $w$ のもとで成り立つかどうかを表すとする．すると，forcing が monotone であることという条件は，単なる weakening となり，本研究で対象とする $\lambda$ 計算においても自明に成り立つ．したがって，これは Kripke モデルとなっている．さらに，関数型についての条件は「現在の環境 $w$ を拡張したどんな環境においても $A$ 型の値を $B$ 型の値に移す」となり，$w$ における $A \to B$ 型の関数閉包の動きを表現するものとなっていることが分かる．

このような Kripke モデルを使って，以下のように TDPE は抽出される．まず，入力の term を Kripke モデルで解釈する．$A$ 型の入力の term を推論規則で $A$ を導けると解釈し，Kripke モデル上の $A$ 型の forcing を Kripke モデル上で $A$ が成り立つと解釈すると，これは Kripke モデルの推論規則に対する soundness になっており，その証明は $A$ 型の term を Kripke モデル上の値に写像するインタプリタとなる．次に，Kripke モデル上の値を term に引き戻す．別々の term でも同じ意味を持つものはたくさんあるので，この引き戻し方はたくさんあるが，その中でも normal form になっているものに引き戻すと元の入力を部分評価した結果を得ることができる．これは Kripke モデルの推論規則に対する completeness になっており，その証明は Kripke モデル上の値を normal form に写像する TDPE となる．本論文では，この先，いろいろな体系に対して soundness と completeness を証明し，そこからインタプリタと TDPE を抽出していく．

## 4. $\lambda_{cbn}$ における TDPE の抽出

本章では call-by-name の（shift/reset を含まない）$\lambda$ 計算 $\lambda_{cbn}$ において，どのように TDPE を抽出したかについて説明する．まず，Danvy により提案された TDPE の定義 [7] を**図 1** に示す．

型は，型変数 $\mathsf{base}$ か関数型 $t_1 \to t_2$ である．TDPE を行う際には，入力の term を部分評価時に実行してしまう式（$\in \mathsf{Static}$）と実行しない式（$\in \mathsf{Dynamic}$）に分解する．前者は static な式と呼ばれオーバラインを付けて表される．

$$
\begin{array}{rcl}
t \in \mathsf{Type} & := & \mathsf{base} \mid t_1 \to t_2 \\
v \in \mathsf{Static} & := & x \mid \overline{\lambda} x.v \mid v_0 \mathbin{\overline{@}} v_1 \\
e \in \mathsf{Dynamic} & := & x \mid \underline{\lambda} x.e \mid e_0 \mathbin{\underline{@}} e_1 \\
d \in \mathsf{TLT} & := & x \mid \overline{\lambda} x.d \mid \underline{\lambda} x.d \mid d_0 \mathbin{\overline{@}} d_1 \mid d_0 \mathbin{\underline{@}} d_1 \\
 & & \\
\text{reify } (\downarrow) & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{TLT} \\
\downarrow^{\mathsf{base}} v & = & v \\
\downarrow^{A \to B} v & = & \underline{\lambda} x^{\diamond}. \downarrow^{B} (v \mathbin{\overline{@}} \uparrow_{A} x^{\diamond}) \\
 & & \\
\text{reflect } (\uparrow) & : & \mathsf{Type} \to \mathsf{Dynamic} \to \mathsf{TLT} \\
\uparrow_{\mathsf{base}} e & = & e \\
\uparrow_{A \to B} e & = & \overline{\lambda} v. \uparrow_{B} (e \mathbin{\underline{@}} \downarrow^{A} v) \\
 & & \\
\text{residualize} & = & \text{statically-reduce} \circ \text{reify} \\
 & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{Dynamic}
\end{array}
$$

**図 1**　Danvy の call-by-name の TDPE
**Fig. 1**　Danvy's call-by-name TDPE.

後者は dynamic な式と呼ばれアンダラインを付けて表される．また，両者が混ざった式（$\in \mathsf{TLT}$）は 2 レベルの式（two-level term）と呼ばれる（TLT は「Two-Level Term」から名付けられている）．Coq で TDPE を実装するなら，static な式は Coq における関数，関数呼び出しそのものに相当し，dynamic な式はデータ型を使って表現されたプログラムの構文木に相当する．

入力の式に対して TDPE を行うには次のようにする．まず，入力の式は一度，完全に static な式，つまりオーバラインの付いた式に変換される．たとえば $\underline{\lambda} x.((\underline{\lambda} y.y) \mathbin{\underline{@}} x)$ という dynamic な式が入力されたとすると，この式はまず完全に static な式 $\overline{\lambda} x.((\overline{\lambda} y.y) \mathbin{\overline{@}} x)$ に変換される．次に，この static な式が，その型 $\mathsf{base} \to \mathsf{base}$ とともに reify に渡される．reify は，reflect を使いながら入力の式を 2 レベルの式に変換する．最後に，得られた 2 レベルの式の中の static な部分をすべて実行すると TDPE 結果が得られる．具体的には，以下のようになる．

$$
\begin{array}{ll}
 & \downarrow^{\mathsf{base} \to \mathsf{base}} \overline{\lambda} x.((\overline{\lambda} y.y) \mathbin{\overline{@}} x) \\
= & \underline{\lambda} x_1^{\diamond}.(\overline{\lambda} x.((\overline{\lambda} y.y) \mathbin{\overline{@}} x)) \mathbin{\overline{@}} x_1^{\diamond} \\
\rightsquigarrow & \underline{\lambda} x_1^{\diamond}.x_1^{\diamond}
\end{array}
$$

ここで $\diamond$ がついている変数は，他の変数とぶつかることのないように選ばれた fresh な変数である．上で得られた dynamic な式は，入力の式を部分評価した結果になっていることが分かる．入力の式がいったん，完全に static な式に変換された後は，その内部構造に立ち入ることなく TDPE が行われていることに注意しよう．このように構文に関する場合分けを行うことなく部分評価を行えるため，通常の部分評価よりも高速に行うことができる．

### 4.1 $\lambda_{cbn}$ の定義

本研究における Coq での定式化に合わせて，Ilik [14] の定式化を参考に，本節では $\lambda_{cbn}$ を定義する（$\lambda_{cbn}$ の term

が，図 1 における dynamic な式に相当している）．$\lambda_{cbn}$ の型システムは**図 2** のように定義される．この定義で特徴的なのは term の定義であろう．term は構文木で定義されており，図 1 のような一般的に使われる term の syntax に，さらに型情報として環境と型が加えられている．このように定義すると，型の付く term のみしか書けなくなる，つまり，一般の型推論規則の条件を満たしていなければ term として成立しない．ただし，型情報が implicit に推論できる場合（もしくは型情報に関する議論が不必要な場合）には，本論文内においてコロンから右側の式（$: w \vdash_{\mathsf{t}} A$）は省いて記述する（実際，Coq で定式化する際にも，`Set Implicit Arguments.` と宣言することにより，自明な環境と型は省略して記述することができる）．逆に，$w \vdash_{\mathsf{t}} A$ のみが書かれた式は，ある $p$ が存在して，$p : w \vdash_{\mathsf{t}} A$ を根とする構文木（term）が書けることを意味している（次節以降に登場する記号 $\vdash_{\mathsf{nf}}$，$\vdash_{\mathsf{ne}}$ に関しても同様である）．

以下，各構文について説明していく．本論文においては，束縛変数は de Bruijn index で表される．$\mathsf{hyp}$ は一番内側の binder で束縛されている変数を，$\mathsf{wkn}(p)$ は $p$ の中の束縛変数の指す先を 1 つ外側の binder に繰り上げるような命令を意味している（定義を見れば分かるように，$\mathsf{wkn}(p)$ の規則は weakening を表すものとなっている）．たとえば恒等関数 $\underline{\lambda}x.x$ は，$\mathsf{lam}(\mathsf{hyp})$，$\underline{\lambda}x.\underline{\lambda}y.y\,\underline{@}\,x$ は $\mathsf{lam}(\mathsf{lam}(\mathsf{app}(\mathsf{hyp}, \mathsf{wkn}(\mathsf{hyp}))))$ と書ける．この体系では，$\mathsf{wkn}(\_)$ が $\mathsf{hyp}$ あるいは $\mathsf{wkn}(\_)$ 以外の式の外に付くような，たとえば $\mathsf{wkn}(\mathsf{app}(\mathsf{hyp}, \mathsf{hyp}))$ のような式を許している．直感的には，application や $\lambda$ 抽象の外側に $\mathsf{wkn}(\_)$ が付いているような term は，その $\mathsf{wkn}(\_)$ を変数に到達するまで中に潜らせた term と同じ意味である．以下に簡単な例を示す：

- $\mathsf{lam}(\mathsf{lam}(\mathsf{lam}(\mathsf{wkn}(\mathsf{app}(\mathsf{wkn}(\mathsf{hyp}), \mathsf{hyp})))))$
  $= \mathsf{lam}(\mathsf{lam}(\mathsf{lam}(\mathsf{app}(\mathsf{wkn}(\mathsf{wkn}(\mathsf{hyp})), \mathsf{wkn}(\mathsf{hyp})))))$
  $= \underline{\lambda}x.\underline{\lambda}y.\underline{\lambda}z.x\,\underline{@}\,y$
- $\mathsf{lam}(\mathsf{lam}(\mathsf{wkn}(\mathsf{lam}(\mathsf{wkn}(\mathsf{hyp})))))$
  $= \mathsf{lam}(\mathsf{lam}(\mathsf{lam}(\mathsf{wkn}(\mathsf{wkn}(\mathsf{hyp})))))) = \underline{\lambda}x.\underline{\lambda}y.\underline{\lambda}z.x$

ただし，中に潜らせる際，束縛変数の束縛先がくずれないようにする必要がある．

- $\mathsf{lam}(\mathsf{wkn}(\mathsf{lam}(\mathsf{hyp}))) = \mathsf{lam}(\mathsf{lam}(\mathsf{hyp})) = \underline{\lambda}x.\underline{\lambda}y.y$

$$
\begin{array}{rcccccl}
\text{type} & : & \mathsf{typ} & \ni & A & := & \mathsf{base} \mid A_1 \to A_2 \\
\text{environment} & : & \mathsf{world} & \ni & w, \Gamma & := & \text{list of } \mathsf{typ}
\end{array}
$$

$$
\text{term} : \quad \frac{}{\mathsf{hyp} : A, w \vdash_{\mathsf{t}} A} \qquad \frac{p : w \vdash_{\mathsf{t}} A}{\mathsf{wkn}(p) : B, w \vdash_{\mathsf{t}} A}
$$

$$
\frac{p : A, w \vdash_{\mathsf{t}} B}{\mathsf{lam}(p) : w \vdash_{\mathsf{t}} A \to B}
$$

$$
\frac{p : w \vdash_{\mathsf{t}} A \to B \quad q : w \vdash_{\mathsf{t}} A}{\mathsf{app}(p, q) : w \vdash_{\mathsf{t}} B}
$$

**図 2**　$\lambda_{cbn}$ と $\lambda_{cbv}$ の型システム
**Fig. 2**　Type systems of $\lambda_{cbn}$ and $\lambda_{cbv}$.

- $\mathsf{lam}(\mathsf{lam}(\mathsf{lam}(\mathsf{wkn}(\mathsf{app}(\mathsf{lam}(\mathsf{hyp}), \mathsf{wkn}(\mathsf{hyp})))))) = \mathsf{lam}(\mathsf{lam}(\mathsf{lam}(\mathsf{app}(\mathsf{lam}(\mathsf{hyp}), \mathsf{wkn}(\mathsf{wkn}(\mathsf{hyp})))))) = \underline{\lambda}x_1.\underline{\lambda}x_2.\underline{\lambda}x_3.(\underline{\lambda}x_4.x_4)\,\underline{@}\,x_1$

一般の de Bruijn index term への変換関数は簡単に定義することができる．その変換プログラムを付録に記す．

Coq で term を定式化する際は，文献 [14] を参考に，以下のように定義した．これは図 2 の term の定義をそのまま Coq で表現しただけである．

```
Set Implicit Arguments.
Inductive tm : world -> typ -> Set :=
| tm_Hyp : forall w A, tm (A :: w) A
| tm_Wkn : forall w A B, tm w A ->
           tm (B :: w) A
| tm_Lam : forall w A B,
           tm (A :: w) B -> tm w (arrow A B)
| tm_App : forall w A B,
           tm w (arrow A B) ->
           tm w A ->
           tm w B.
```

この定義を用いて，たとえば恒等関数は，（world $w$ において (base → base) 型を持つとすると），

```
(tm_Lam (tm_Hyp w base))
```

と記述できる（`Set Implicit Arguments` を使用しない場合は，すべての情報を記述しないといけないため，

```
(tm_Lam w base base (tm_Hyp w base))
```

と記述する）．

### 4.2　$\lambda_{cbn}$ の Kripke モデル

$\lambda_{cbn}$ の Kripke モデルを定義するために，まず $\lambda_{cbn}$ の normal form と neutral term の定義を行う．normal form は正規形であり，これ以上簡約できない term となっている．neutral term は変数を正規形で呼び出した形で，変数の値が具体化しない限り，これ以上，実行を進められない term を意味している．定義は**図 3** に載せる．term の場合と同様に，normal form と neutral term も構文木の形で定義される．

$$
\text{normal form} : \quad \frac{p : w \vdash_{\mathsf{ne}} A}{p : w \vdash_{\mathsf{nf}} A} \qquad \frac{p : A, w \vdash_{\mathsf{nf}} B}{\mathsf{lam}(p) : w \vdash_{\mathsf{nf}} A \to B}
$$

$$
\text{neutral term} : \quad \frac{}{\mathsf{hyp} : A, w \vdash_{\mathsf{ne}} A} \qquad \frac{p : w \vdash_{\mathsf{ne}} A}{\mathsf{wkn}(p) : B, w \vdash_{\mathsf{ne}} A}
$$

$$
\frac{p : w \vdash_{\mathsf{ne}} A \to B \quad q : w \vdash_{\mathsf{nf}} A}{\mathsf{app}(p, q) : w \vdash_{\mathsf{ne}} B}
$$

**図 3**　$\lambda_{cbn}$ の normal form と neutral term
**Fig. 3**　normal forms and neutral terms for $\lambda_{cbn}$.

$$w \models \mathsf{base} \iff w \vdash_{\mathsf{ne}} \mathsf{base}$$
$$w \models A \to B \iff \forall w', w \le w' \Rightarrow w' \models A \Rightarrow w' \models B$$
$$w \models_s (A_1, ..., A_n) \iff (w \models A_1) \wedge \cdots \wedge (w \models A_n)$$

図 4　$\lambda_{cbn}$ で使用する論理述語
**Fig. 4**　Logical predicates for $\lambda_{cbn}$.

本論文で使用する Kripke モデルにおける可能世界の集合は，論理式（型）のコンマで区切られた列とし，その間の大小関係は以下に示す列の拡張とする．

(*i*)　$w \le w$
(*ii*)　$w \le w' \Rightarrow w \le (A, w')$

この定義は，前順序に求められる推移律を満たしていることが容易に確認できる．

**補題 1（transitivity, $\le$）**

$$\forall w_1, \forall w_2, \forall w_3, w_1 \le w_2 \Rightarrow w_2 \le w_3 \Rightarrow w_1 \le w_3$$

**図 4** に，本節で使用する $\lambda_{cbn}$ 用の論理述語 $\models$ と $\models_s$ を記載する．前者は単純型付き $\lambda$ 計算の強正規化性を証明する際に用いられる Tait 流の論理述語 [18] を Kripke モデル用に拡張したものであり，$A \to B$ のケースは 3 章で紹介した定義となっている．また，$w \models_s (A_1, ..., A_n)$ は $A_1, \ldots, A_n$ それぞれが論理述語を満たすことを示している．

この論理述語 $w \models A$ は，論理式 $A$ の Kripke モデル上での意味を表している．そこから TDPE の結果を normal form の形で得たいので，$w \models \mathsf{base}$ の意味として（dynamic な）normal form の集合をとればよい．ただし，型が base の場合は関数型にはなりえないので，normal form の中でも neutral term のみを考慮すればよい．よって，$w \models \mathsf{base}$ は $w \vdash_{\mathsf{ne}} \mathsf{base}$（つまり，$w$ のもとで base 型の neutral term の集合）と定義している．

Kripke モデルとなるためには，$w \models \mathsf{base}$（つまり $w \vdash_{\mathsf{ne}} \mathsf{base}$）は monotonicity を満たさなくてはならない．これは，次の補題により確認される．

**補題 2（monotonicity, $\lambda_{cbn}$）**

$$\forall w, \forall w', \forall A, w \le w' \Rightarrow w \vdash_{\mathsf{ne}} A \Rightarrow w' \vdash_{\mathsf{ne}} A$$

この補題は $w \le w'$ の定義に関する帰納法により容易に示すことができる．この補題で $A$ を base とすると，必要な monotonicity を得ることができる．

Coq で定式化する際には，$\le$ は Prop 型ではなく Set 型で定義している．そのため，あとで TDPE を抽出する際，この $\le$ に相当する部分もコードとして現れてくることになる．これは，この補題の結論部（$w' \vdash_{\mathsf{ne}} A$）が（TDPE 結果として normal form，neutral term を得たいので）Set 型であるためである．結論部が Set 型である場合，Coq には「Goal が Set または Type 型である場合は，帰納法を用いる式も Set または Type 型でなければならない」という制約があるため，$\le$ の定義を Set 型にしないと $\le$ に関する帰納法を使えない．$\le$ に関する帰納法を使わずに上の補題を証明できれば，$\le$ を Prop 型と定義でき，抽出されたコードに現れないようにできるが，現在のところ，そのような形での証明はできていない．

### 4.3　$\lambda_{cbn}$ の soundness の証明

前節で定義した Kripke モデルの推論規則に対する soundness の定理は以下のように述べられる．

**定理 1（Soundness, $\lambda_{cbn}$）**

$$\forall A, \forall \Gamma, \forall w, \Gamma \vdash_{\mathsf{t}} A \Rightarrow w \models_s \Gamma \Rightarrow w \models A$$

証明は $\Gamma \vdash_{\mathsf{t}} A$ に関する帰納法を用いて行う．ここで $\Gamma$ は $w$ と同じく論理式（型）の列である．この定理が意味するところは，「$\Gamma$ のもとで $A$ が成り立つなら，Kripke モデル上でも $\Gamma$ のもとで $A$ が成り立つ（言い換えると，$\Gamma$ のもとで $A$ 型になるような term が存在するなら，$\Gamma$ を満たす世界（値環境）のもとで $A$ 型として振る舞う値が存在する）」である．これが soundness と呼ばれるのは，推論規則上で $A$ が成り立つなら Kripke モデル上でも $A$ が成り立つという定理になっているからである．soundness の statement は「$w \vdash_{\mathsf{t}} A \Rightarrow w \models A$」であるべきと思われるかもしれないが，この statement を直接的に示すことは今のところできていない．また，Coquand [4] が定理 1 と同様の性質を Kripke モデルの推論規則に対する soundness と呼んでいるため，本論文もそれにならう形をとっている．

Curry Howard 同型により，この定理を抽出すると，$A$ 型の term を Kripke モデル上の値に写像するような（$\lambda_{cbn}$ に対する）インタプリタが得られる．$\Gamma \vdash_{\mathsf{t}} A$ は $A$ 型の項，$w \models_s \Gamma$ は値環境，$w \models A$ は $A$ 型の値に相当する．実際，soundness の証明を Coq で書き下すと以下のようになる（このプログラムは soundness の証明そのものになっており，この定義を使うと Coq では `exact interp.` で soundness の証明が終了する）．

```
Fixpoint interp (G: world) (t: typ)
                (term: tm G t) {struct term}
: forall w: world, Rs w G -> R w t :=
  match term with
| tm_Hyp _ _ => fun _ env =>
   get_first env
| tm_Wkn _ _ _ t1 => fun _ env =>
   interp t1 (get_rest env)
| tm_Lam _ A _ t1 => fun w1 env =>
   fun (w2: world) (H: w1 <== w2) (v: R w2 A) =>
   interp t1 (Rs_cons A v (wkn_Rs H env))
| tm_App _ _ _ t1 t2 => fun w1 env =>
   interp t1 env w1 (lew_refl w1)
```

```
  (interp t2 env)
end.
```

ここで tm G t は $G \vdash_{\mathsf{t}} t$ を，R w t は $w \models t$ を，Rs w G は $w \models_s G$ を表す．このプログラムは $t$ 型を持つ項 term と環境 env を受け取ると，項に従って場合分けをして，解釈実行するようなインタプリタである．別の言い方をすると，各 term に対して Kripke モデル上の意味を与えている．以下，このプログラムを詳細に見ていく．

term が hyp の場合は，環境の最初の要素を値として返す．get_first は（空でない）環境の先頭を返す関数であり，別途，(適切な命題の証明として) 定義されている．term が $\mathsf{wkn}(t_1)$ の場合は，get_rest を使って環境の先頭の要素を捨てたうえで，再帰する．term が $\mathsf{lam}(t_1)$ の場合は「$A$ 型の引数 $v$ を受け取ったら，$t_1$ を実行する」ような関数を返す．Rs_cons は環境の先頭に値を挿入する関数である．その際，この関数は引数に加えて，呼び出されたときの環境 $w_2$ と，それがこの関数の定義時の環境 $w_1$ からどれくらい離れているかを示す $H$ を受け取ってきている．これは，使用している Kripke モデル（図 4）で関数がそのように定義されているためである．しかし，次の $\mathsf{app}(t_1, t_2)$ の場合を見ると分かるとおり，$w_2$ はいつも $w_1$ と等しく，$H$ はいつも lew_refl w1（これは $w_1 \leq w_1$ の証明を示している）となっている．wkn_Rs は env の中の変数に $H$ に示される差分だけ $\mathsf{wkn}(\_)$ を挿入する関数だが，$H$ がいつも lew_refl w1 なので結局 wkn_Rs H は恒等関数になる．つまり tm_Lam のケースで現れる引数 (H: w1 <== w2) はインタプリタの実行には不要である．Coq で定式化する際に $\leq$ を Prop 型で定義できれば，プログラムを抽出した際に，この引数は登場せずにすむが，現在 $\leq$ の定義を Prop 型にはできていないので，このような形で残っている．最後に，term が $\mathsf{app}(t_1, t_2)$ の場合は $t_1$ を実行し，その結果得られる関数に $t_2$ を実行した結果を渡している（メタ言語である Coq では一般の $\beta$ 簡約が許されているので，これで call-by-name のインタプリタになっている）．

なお，上記のインタプリタプログラムは，最初に Coq で soundness の証明を行い，それを Print コマンドで表示させて，整理することで得たものである．しかし，一度，このようなインタプリタが得られるということを理解しておくと，それに従って証明を楽に進めることができるようになる．

### 4.4 $\lambda_{cbn}$ の completeness の証明

$\lambda_{cbn}$ の completeness の定理は，本研究では以下のように定義する．

**定理 2（Completeness, $\lambda_{cbn}$）**

$\forall A,\quad (i)\quad \forall w, w \models A \Rightarrow w \vdash_{\mathsf{nf}} A$

$\qquad (ii)\quad \forall w, w \vdash_{\mathsf{ne}} A \Rightarrow w \models A$

この定理の前半が意味するところは，「Kripke モデル上で $A$ を証明できるなら，$A$ 型の正規形が存在する（言い換えると，$A$ 型の値を受け取ったら，その正規形を返す）」であり，これはまさに reify に対応していることが分かる．一方，後半の意味するところは「$A$ 型の neutral term は $A$ 型の値に変換できる」となり，reflect に対応している．後者は，前者を型に関する帰納法で証明する際に必要となる．

この定理は $(i)$ の結論部を $w \vdash_{\mathsf{t}} A$ に置き換えても証明が可能である．論理の意味から考えると，その方が自然であると思われるが，本研究では正規形を導出するプログラム（つまり reify）を得ることを目的としているので，上記のように限定して $w \vdash_{\mathsf{nf}} A$ を結論部としている．また，$(ii)$ の前件部が $w \vdash_{\mathsf{ne}} A$ となっているのは，論理述語 $w \models \mathsf{base}$ の定義が $w \vdash_{\mathsf{ne}} \mathsf{base}$ となっているため，neutral term よりも制約の緩い項を前件部にて許してしまうと $A = \mathsf{base}$ のケースの証明がうまくいかなくなってしまうためである．

completeness の定理の証明の概略は以下のようになる．論理式の proof term（図 1 中の TDPE プログラム）を，その式の後ろに大括弧で括って [...] 示すこととする．

Proof.　型に関する帰納法．

base の場合：

$(i)$ (reify，$\downarrow^{\mathsf{base}} v$ のケース)

$w \models \mathsf{base}\ [v]$ は定義から $w \vdash_{\mathsf{ne}} \mathsf{base}$ となり，neutral term は normal form の 1 つなので $w \vdash_{\mathsf{nf}} \mathsf{base}\ [v]$ である．

$(ii)$ (reflect，$\uparrow_{\mathsf{base}} e$ のケース)

$w \models \mathsf{base}\ [e]$ の定義は $w \vdash_{\mathsf{ne}} \mathsf{base}\ [e]$ なので自明．

$A \to B$ の場合：

$(i)$ (reify，$\downarrow^{A \to B} e$ のケース)

仮定 $w \models A \to B\ [v]$ は定義により $\forall w', w \leq w' \Rightarrow w' \models A \Rightarrow w' \models B$ と等価である．ここで $w'$ として $w$ を拡張した $(A, w)$ を考える．hyp の規則により $(A, w) \vdash_{\mathsf{ne}} A\ [x^\diamond]$ が成り立つので，$A$ に関する帰納法の仮定 $(ii)$ を使うと $(A, w) \models A\ [\uparrow_A x^\diamond]$ を得る．これに対して，仮定を使うと $(A, w) \models B\ [v@ \uparrow_A x^\diamond]$ が得られるので，$B$ についての帰納法の仮定 $(i)$ を使うと $(A, w) \vdash_{\mathsf{nf}} B\ [\downarrow^B v@(\uparrow_A x^\diamond)]$ となる．よって，$\mathsf{lam}(\_)$ の規則を使うと，結論の $w \vdash_{\mathsf{nf}} A \to B\ [\lambda x^\diamond. \downarrow^B (v@ \uparrow_A x^\diamond)]$ を得る．

$(ii)$ (reflect，$\downarrow^{A \to B} e$ のケース)

$w \vdash_{\mathsf{ne}} A \to B\ [e]$ を仮定して，$w \models A \to B$ を示す．示したい結論部は定義により $\forall w', w \leq w' \Rightarrow w' \models A \Rightarrow w' \models B$ である．そこでさらに $w \leq w'$ と $w' \models A\ [v]$ を仮定して $w' \models B$ を示す．今，$w' \models A\ [v]$ に対して $A$ に関する帰納法の仮定 $(i)$ を使うと $w' \vdash_{\mathsf{nf}} A\ [\downarrow^A v]$ を得る．一方，$w \leq w'$ なので，$w \vdash_{\mathsf{ne}} A \to B\ [e]$ に適切に $\mathsf{wkn}(\_)$ の規則を使うと $w' \vdash_{\mathsf{ne}} A \to B\ [e]$ を得る．この 2 つに対して $\mathsf{app}(\_, \_)$ の規則を適用すると $w' \vdash_{\mathsf{ne}} B\ [e @ \downarrow^A v]$ を得る．最後に $B$ に関する帰納法の仮定 $(ii)$ を使うと，欲しかった $w' \models B\ [\downarrow^B (e @ \downarrow^A v)]$ が得られる．　□

上の証明は，そのまま TDPE の定義にそっていること

が見て取れる．実際，completeness の証明は TDPE の定義と 1 対 1 の関係にあり，Coq の証明もそれにそって行った．その結果，得られた証明は**図 5** のようになる．

`nf w t`，`ne w t` はそれぞれ $w \vdash_{\mathsf{nf}} t$，$w \vdash_{\mathsf{ne}} t$ を表す．また，`nf_ne` は neutral term を normal form に埋め込む関数，それ以外の `nf_`，`ne_` で始まる関数は，それぞれ normal form，neutral term の構成子である．

このプログラムを見ると，TDPE の定義と 1 対 1 に対応しているばかりでなく，他とぶつからない fresh な変数をとってくる部分についても正しく扱われていることが分かる．fresh な変数が導入されるのは reify の $A \to B$ 型のところであり，`nf_Lam` の中で `ne_Hyp` が使われている．今のところこの変数は `nf_Lam` の直下にあるので `ne_Hyp` のままであるが，実際にはその変数は `reflect` に渡され，その結果は `v` にも渡される．その中でさらに別の `nf_Lam` が使われるかもしれない．そこで，この変数を `reflect` に渡すときには（implicit に）現在の環境を渡すとともに，`v` を実行する際には呼び出し時の環境 `A :: w` と環境が定義時からどのくらいずれているかを示す証明 `lew_cons A (lew_refl w)` を一緒に渡している（`lew_cons A l` は `l` が $w \le w'$ の証明だったとき，$w \le (A, w')$ の証明を示す）．この証明は `reflect` の $A \to B$ 型のケースで使われる．受け取った neutral term `e` をコードに残す際には `wkn_ne` を使って適切に `ne_Wkn` を挿入している．`wkn_ne` は補題 2 に対応する関数で，受け取った証明に従って，必要な数（= `e` の指し示す `nf_Lam` までの間に出てくる別の `nf_Lam` の数）だけ `ne_Wkn` を挿入する関数である．

ここで得られたプログラムからさらに OCaml のコードを抽出することも可能である．しかし，TDPE は OCaml の型システムでは（特殊なエンコーディングをしない限り [20]）直接は表現できないので，`Obj.magic` が現れる格好になる．

```
Fixpoint reify (t: typ) : forall w, R w t -> nf w t :=
  match t return (forall w, R w t -> nf w t) with
| base => fun _ v => nf_ne v
| arrow A B => fun w v =>
   nf_Lam (reify B (A :: w)
                (v (A :: w) (lew_cons A (lew_refl w))
                   (reflect (ne_Hyp w A))))
 end
with reflect (t: typ) : forall w, ne w t -> R w t :=
  match t return (forall w, ne w t -> R w t) with
| base => fun _ e => e
| arrow A B => fun w e =>
   fun w1 (H: w <== w1) (v: R w1 A) =>
       reflect (ne_App (wkn_ne H e)
                       (reify A w1 v))
 end.
```

**図 5**　$\lambda_{cbn}$ の抽出された TDPE (reify/reflect)
**Fig. 5**　The extracted TDPE (reify/reflect) for $\lambda_{cbn}$.

### 4.5　$\lambda_{cbn}$ 用の reify への入力について

soundness と completeness の定理を証明できると，そこから抽出されたプログラムを使って以下のように TDPE の結果を得ることができる．まず，入力のプログラムは `interp` を使って対応する Kripke モデル上の意味へと変換される．これが，入力が static な式に変換されたことに対応する．次に，その結果を `reify` に渡すと，TDPE 結果である normal form が得られる．このように `reify` への入力は Kripke モデル上の意味となっている．

## 5.　$\lambda_{cbv}$ における TDPE の抽出

前章を受けて，本章では call-by-value の（shift/reset を含まない）$\lambda$ 計算 $\lambda_{cbv}$ における TDPE の抽出を説明する．TDPE の抽出手法は，前章のそれをきれいな形で拡張したものになっている．まずは先行研究でなされている call-by-value における TDPE の定義を示す．Tsushima らが文献 [19] で記している call-by-value の $\lambda$ 計算における TDPE の定義は，let-insertion を用いて行われている．その let-insertion に shift/reset を用いた定義が**図 6** である．static な shift を $\overline{S}$，static な reset を $\overline{\langle \_ \rangle}$ で表している．

reflect の $A \to B$ のケースは，call-by-name では $\overline{\lambda} v.\uparrow_B (e \underline{@} \downarrow^A v)$ と定義されているが，call-by-value では dynamic な $(e \underline{@} \downarrow^A v)$ を let-insertion を使って出力している．$\overline{S}k$ で，その時点での継続を切り取り，fresh な変数 $g^\diamond$ を使って $(e \underline{@} \downarrow^A v)$ を let 文に残し，以降の計算には $g^\diamond$ が渡されている．この let 文が TDPE の実行により出力されるのは，継続が static な reset で区切られているところ，すなわち reify の $A \to B$ のケースで dynamic な束縛が作られる直下である．つまり，$\underline{\lambda} x^\diamond.\underline{\mathsf{let}}\ g^\diamond = \ldots\ \underline{\mathsf{in}}\ \ldots$ といった形で出力される．また，Danvy [8] の定義では $k \,\overline{@}\, \uparrow_B g^\diamond$ を（static な）reset で括っているが，この式を reset で括っても括らなくても，式の意味は変わらない．shift でとってきた継続 $k$ には reset が付いているため，引数 $v$ が value ならば $k \,\overline{@}\, v$ と $\overline{\langle k \,\overline{@}\, v \rangle}$ は等価になるからである（実際に

$$
\begin{array}{rcl}
t \in \mathsf{Type} & := & \mathsf{base} \mid t_1 \to t_2 \\
v \in \mathsf{Static} & := & x \mid \overline{\lambda} x.v \mid v_0 \,\overline{@}\, v_1 \mid \overline{S} x.v \mid \overline{\langle v \rangle} \\
e \in \mathsf{Dynamic} & := & x \mid \underline{\lambda} x.e \mid e_0 \,\underline{@}\, e_1 \mid \underline{\mathsf{let}}\ x = e_1\ \underline{\mathsf{in}}\ e_2 \\
d \in \mathsf{TLT} & := & x \mid \overline{\lambda} x.d \mid \underline{\lambda} x.d \mid d_0 \,\overline{@}\, d_1 \mid d_0 \,\underline{@}\, d_1 \\
 & \mid & \overline{S} x.d \mid \overline{\langle d \rangle} \mid \underline{\mathsf{let}}\ x = d_1\ \underline{\mathsf{in}}\ d_2 \\
 & & \\
\text{reify } (\downarrow) & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{TLT} \\
\downarrow^{\mathsf{base}} v & = & v \\
\downarrow^{A \to B} v & = & \underline{\lambda} x^\diamond.\overline{\langle} \downarrow^B (v \,\overline{@}\, \uparrow_A x^\diamond) \overline{\rangle} \\
 & & \\
\text{reflect } (\uparrow) & : & \mathsf{Type} \to \mathsf{Dynamic} \to \mathsf{TLT} \\
\uparrow_{\mathsf{base}} e & = & e \\
\uparrow_{A \to B} e & = & \overline{\lambda} v.\overline{S} k.\underline{\mathsf{let}}\ g^\diamond = e \,\underline{@}\, \downarrow^A v\ \underline{\mathsf{in}}\ k \,\overline{@}\, \uparrow_B g^\diamond
\end{array}
$$

**図 6**　call-by-value の TDPE
**Fig. 6**　The call-by-value TDPE.

$$
\begin{array}{lcl}
\text{reify } (\downarrow) & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{TLT} \\
\downarrow^{\mathsf{base}} v & = & v \\
\downarrow^{A \to B} v & = & \underline{\lambda} x^{\diamond}.v\, \overline{@}\, (\uparrow_A x^{\diamond})\, \overline{@}\, (\overline{\lambda} v'.\downarrow^B v')
\end{array}
$$

$$
\begin{array}{lcl}
\text{reflect } (\uparrow) & : & \mathsf{Type} \to \mathsf{Dynamic} \to \mathsf{TLT} \\
\uparrow_{\mathsf{base}} e & = & e \\
\uparrow_{A \to B} e & = & \overline{\lambda} v.\overline{\lambda} k.\underline{\mathrm{let}}\ g^{\diamond} = e\, \underline{@}\, \downarrow^A v\ \underline{\mathrm{in}}\ k\, \overline{@}\, \uparrow_B g^{\diamond}
\end{array}
$$

図 7　call-by-value の CPS TDPE
**Fig. 7**　The call-by-value CPS TDPE.

Kameyama らの諸公理 [15] を使って証明することが可能）．

この TDPE の定義は，static な shift/reset が使われている．Coq は shift/reset を提供していないため，図 6 の TDPE 定義を直接 Coq で定式化することはできない．また，static な shift/reset が入ってくると，Kripke モデルの構築が複雑になり扱いにくくなると予想される．そこで本研究では，（TDPE 中の）static な箇所のみを CPS 変換することにより，その shift/reset を削除する（TDPE には static な式と dynamic な式が入り混じっているため，どの場所を CPS 変換すればよいのかが明らかではなかったが，Ilik の先行研究 [14] を詳細に検討した結果，static な部分のみを CPS 変換すればよいことが分かった）．図 6 の static な term を CPS 変換したものが，**図 7** で定義した関数である．この定義は，先の call-by-value の TDPE のうち，reify に渡される引数 $v$ と reflect が返す static な値が継続を受け取るように変換されている．reify に渡される引数 $v$ は CPS になったので，それを使う際には引数 $(\uparrow_A x^{\diamond})$ に加えて，継続が渡されている．また，reflect が返す値は，$\overline{\lambda} k$ のように継続を受け取る格好になっている．このように CPS 変換を施すと，shift/reset を使って行っていた let-insertion を shift/reset を使わずに行うことができる．

### 5.1　$\lambda_{cbv}$ の定義と Kripke モデル

$\lambda_{cbv}$ の term の定義は $\lambda_{cbn}$ と同じである（図 2）．$\lambda_{cbv}$ で使用する Kripke モデルは，$\lambda_{cbn}$ で定義したモデルで使用した neutral term に新たに let 文を加え，さらに論理述語の $A \to B$ のケースにおける定義を CPS の形に拡張して定義する．

$\lambda_{cbv}$ における normal form, neutral term の定義に関しては**図 8** に記載する．ここで定義されている $\mathsf{let}(\mathsf{app}(q, q'), p)$ は，$\mathsf{app}(q, q')$ を（$p$ の中の）0 番の変数に束縛して $p$ を実行するような let 式である．この式は $\mathsf{app}(\mathsf{lam}(p), \mathsf{app}(q, q'))$ と同じことであるが，見やすさのために導入した．この式は call-by-name ではさらに $\beta$ 簡約ができる．しかし call-by-value では $\mathsf{lam}(p)$ に渡される項が（これ以上簡約できないような）application だと簡約ができない．図 8 の let 式の定義を見れば分かるとおり $q$ は neutral term

$$
\begin{array}{ll}
\text{normal form} & : \\
& \dfrac{p : w \vdash_{\mathsf{ne}} A}{p : w \vdash_{\mathsf{nf}} A} \qquad \dfrac{p : A, w \vdash_{\mathsf{nf}} B}{\mathsf{lam}(p) : w \vdash_{\mathsf{nf}} A \to B} \\
\text{neutral term} & : \\
& \dfrac{}{\mathsf{hyp} : A, w \vdash_{\mathsf{ne}} A} \qquad \dfrac{p : w \vdash_{\mathsf{ne}} A}{\mathsf{wkn}(p) : B, w \vdash_{\mathsf{ne}} A} \\
& \dfrac{q : w \vdash_{\mathsf{ne}} C \to A \quad q' : w \vdash_{\mathsf{nf}} C \quad p : A, w \vdash_{\mathsf{nf}} B}{\mathsf{let}(\mathsf{app}(q, q'), p) : w \vdash_{\mathsf{ne}} B}
\end{array}
$$

図 8　$\lambda_{cbv}$ の normal form と neutral term
**Fig. 8**　normal forms and neutral terms for $\lambda_{cbv}$.

$$
\begin{array}{lcl}
w \models \mathsf{base} & \iff & w \vdash_{\mathsf{ne}} \mathsf{base} \\
w \models A \to B & \iff & \forall w_1, w \le w_1 \Rightarrow w_1 \models A \Rightarrow \\
& & \forall T, (\forall w_2, w_1 \le w_2 \Rightarrow \\
& & \quad w_2 \models B \Rightarrow w_2 \vdash_{\mathsf{nf}} T) \Rightarrow \\
& & \quad w_1 \vdash_{\mathsf{nf}} T
\end{array}
$$

$$
w \models_s (A_1, ..., A_n) \iff (w \models A_1) \wedge \cdots \wedge (w \models A_n)
$$

図 9　$\lambda_{cbv}$ で使用する論理述語
**Fig. 9**　Logical predicates for $\lambda_{cbv}$.

であるので，$\mathsf{app}(q, q')$ はこれ以上簡約できず，ゆえに $\mathsf{app}(\mathsf{lam}(p), \mathsf{app}(q, q'))$ はこれ以上簡約できない項となっている．そのため，neutral term として定義する必要がある．この式は，これ以上実行を進められない項 $\mathsf{app}(q, q')$ を let 文に残していることに相当する．neutral term において $\mathsf{app}(\_, \_)$ が単独では現れないことに注意されたい．$\mathsf{app}(\_, \_)$ は必ず let 文の中にのみ現れる．

このように拡張された normal form と neutral term を用いて定義された論理述語が**図 9** である．この定義が $\lambda_{cbn}$ の場合と異なるのは，$w \models \mathsf{base}$ のケースの $w \vdash_{\mathsf{ne}} \mathsf{base}$ の定義が変更されていること，そして $w \models A \to B$ のケースの定義が CPS になっていることの 2 点である．後者について，$\lambda_{cbn}$ では $w \models A \Rightarrow w \models B$ となっていた式が，$\lambda_{cbv}$ では $w \models B$ を受け取る継続を使うようになっている．この定義は少し複雑に見えるが，$w \vdash_{\mathsf{ne}} T$ の形をした部分を継続が返す型 $T$ と思い，さらに $w$ に関する部分を無視すると $A \Rightarrow \forall T.(B \Rightarrow T) \Rightarrow T$ という形をしており，確かに CPS になっていることが分かる．

$\lambda_{cbv}$ のときと同様，論理述語が Kripke モデルの forcing の関係を満たすためには，base のケースにおいて monotone でなくてはならない．$\lambda_{cbn}$ のときと同様，$\vdash_{\mathsf{ne}}$ について以下の性質が成立する．

**補題 3**（**monotonicity,** $\lambda_{cbv}$）

$$
\forall w, \forall w', \forall A, w \le w' \Rightarrow w \vdash_{\mathsf{ne}} A \Rightarrow w' \vdash_{\mathsf{ne}} A
$$

この性質は，（前章のときと同様に），$w \le w'$ の定義に関する帰納法を用いて容易に示せ，この補題の $A$ を base にすると，$\models$ における monotonicity を得ることができ

る．以上のセッティングにより，$\lambda_{cbn}$ のときと同様に，Kripke モデルを定義することができた．この諸定義を用いて，soundness と completeness の証明を行う．

### 5.2 $\lambda_{cbv}$ の soundness の証明

$\lambda_{cbv}$ での soundness 定理は以下のように，$\lambda_{cbn}$ の場合よりも複雑な定義となる．

**定理 3**（**Soundness,** $\lambda_{cbv}$）

$$\forall A, \forall \Gamma, \forall w, \Gamma \vdash_{\mathsf{t}} A \Rightarrow w \models_s \Gamma \Rightarrow$$
$$\forall T, (\forall w', w \leq w' \Rightarrow w' \models A \Rightarrow w' \vdash_{\mathsf{nf}} T) \Rightarrow w \vdash_{\mathsf{nf}} T$$

定理 3 の定義が意味するところは基本的に定理 1 と同じであるが，結論部「$\Gamma$ を満たす値環境の中で $A$ 型として振る舞う値が存在する」の「$A$ 型として振る舞う値」というのは，ここでは CPS になっているので，「$\Gamma$ を満たす環境の中で，『$A$ 型の値を受け取ったら $T$ 型の値になるような継続』を受け取ったら，$T$ 型の値を返す」という意味になる．

この定理の証明は定理 1 と同じく term に関する帰納法となる．証明は省略するが，証明の結果，得られるプログラムは CPS で書かれたインタプリタになる．前節の Kripke モデルにおいては，関数は $A$ 型のものを受け取ったら $B$ 型のものを返す直接形式のものとして扱われていたので，直接形式のインタプリタが得られていた．一方，本節の Kripke モデルにおいては，関数は継続を受け取る CPS の形のものとなっている．そのため，term の意味も CPS となっているのである．

### 5.3 $\lambda_{cbv}$ の completeness の証明

$\lambda_{cbv}$ における completeness 定理の定義は（$\vdash_{\mathsf{nf}}$，$\vdash_{\mathsf{ne}}$，$\models$ の定義が異なっていることを除けば）$\lambda_{cbn}$ での定義とまったく同じである．

**定理 4**（**Completeness,** $\lambda_{cbv}$）

$$\forall A,\quad (i)\quad \forall w, w \models A \;\Rightarrow\; w \vdash_{\mathsf{nf}} A$$
$$(ii)\quad \forall w, w \vdash_{\mathsf{ne}} A \;\Rightarrow\; w \models A$$

定理 4 の証明は，図 7 の CPS TDPE のプログラムと 1 対 1 に対応した形で行うことができる．この証明から得られた（Coq 上の）プログラムが**図 10** である．定義に登場する `ne_Let q q' p` は $\mathsf{let}(\mathsf{app}(q, q'), p)$ を意味している．図 10 と図 6 を比較すれば，両者が同一の関数を表現しているであろうことが見て取れる．

completeness の定理は，$\lambda_{cbn}$ のケースに合わせるために上のような形になっているが，$\lambda_{cbv}$ のケースに限れば $(ii)$ の前提の $w \vdash_{\mathsf{ne}} A$ の部分は任意の neutral term ではなく変数に対応する式（つまり hyp やそれに $\mathsf{wkn}(\_)$ がかかったもの）のみとしても証明できる．これは，$\lambda_{cbv}$ では let-insertion によりすべての部分式に名前が付くため，変数を扱えさえすれば十分だからである．

```
Fixpoint reify (t: typ)
         : (forall w, R w t -> nf w t) :=
  match t return (forall w, R w t -> nf w t) with
| base => fun _ v => nf_ne v
| arrow A B => fun w v =>
   nf_Lam
    (v (A :: w) B (lew_cons A (lew_refl w))
       (reflect (ne_Hyp w A))
       (fun w2 _ v' => reify B w2 v'))
end
with reflect (t: typ)
     : (forall w, ne w t -> R w t) :=
  match t return (forall w, ne w t -> R w t) with
| base => fun _ e => e
| arrow A B => fun w e =>
   fun w1 _ (H: w <== w1) (v: R w1 A) k =>
    nf_ne
     (ne_Let
      (wkn_ne H e) (reify A w1 v)
      (k (B :: w1) (lew_cons B (lew_refl w1))
         (reflect (ne_Hyp w1 B))))
 end.
```

図 10　$\lambda_{cbv}$ の抽出された TDPE (reify/reflect)
**Fig. 10**　The extracted TDPE (reify/reflect) for $\lambda_{cbv}$.

### 5.4 $\lambda_{cbv}$ 用の reify への入力について

以上で $\lambda_{cbv}$ 用の TDPE が得られたが，この TDPE が受け取る入力について，1 つ面白いことが起きている．$\lambda_{cbn}$ 用の TDPE では，完全に dynamic な入力を一度，完全に static な式に変換してから `reify` をかけていた．その際，`reify` への入力はもとの入力と同じ直接形式であった．一方，$\lambda_{cbv}$ 用の TDPE では，Kripke モデルの意味表現を CPS で表現しているため，`reify` への入力も CPS でなくてはならなくなっている．たとえば $\underline{\lambda}x.((\underline{\lambda}y.y)\,\underline{@}\,x) : \mathsf{base} \to \mathsf{base}$ を TDPE にかけた結果が欲しければ，$\lambda_{cbn}$ 用の TDPE を使うときには $\overline{\lambda}x.((\overline{\lambda}y.y)\,\overline{@}\,x)$ を `reify` に渡せばよかったが，$\lambda_{cbv}$ 用の TDPE を使うときには $\overline{\lambda}x.((\overline{\lambda}y.y)\,\overline{@}\,x)$ を CPS 変換したものを `reify` に渡す必要があるのである．

これは，一見，目指していた TDPE とは異なるものが得られたように感じるかもしれないが，そうではない．TDPE を使うときには，まず入力プログラムをコンパイルし内部表現に変換する．そして，その内部表現を使って TDPE の結果を得る．$\lambda_{cbn}$ 用の TDPE では内部表現として直接形式を使っていたが，$\lambda_{cbv}$ 用の TDPE では内部表現として CPS 形式を使っているのである．実際，$\lambda_{cbv}$ 用の soundness の定理は，CPS インタプリタになっていた．したがって，どちらのケースでも入力プログラムは soundness の示すインタプリタ上で内部表現に変換された後，それが `reify` に渡されて結果が得られるという形になっている．

$$
\begin{array}{lcl}
t \in \mathsf{Type} & := & \mathsf{base} \mid t_1/t_3 \to t_2/t_4 \mid t_1 \to_p t_2 \\
v \in \mathsf{Static} & := & x \mid \overline{\lambda} x.v \mid v_0 \,\overline{@}\, v_1 \mid \overline{S} x.v \mid \overline{\langle v \rangle} \mid \overline{S_2} x.v \mid \overline{\langle_2 x \rangle_2} \\
e \in \mathsf{Dynamic} & := & x \mid \underline{\lambda} x.e \mid e_0 \,\underline{@}\, e_1 \mid \underline{S} x.e \mid \underline{\langle e \rangle} \mid \underline{\mathrm{let}}\ x = e_1\ \underline{\mathrm{in}}\ e_2 \\
d \in \mathsf{TLT} & := & x \mid \overline{\lambda} x.d \mid \underline{\lambda} x.d \mid d_0 \,\overline{@}\, d_1 \mid d_0 \,\underline{@}\, d_1 \mid \overline{S} x.d \mid \overline{\langle d \rangle} \mid \underline{S} x.d \mid \underline{\langle d \rangle} \mid \overline{S_2} x.d \mid \overline{\langle_2 d \rangle_2} \mid \underline{\mathrm{let}}\ x = d_1\ \underline{\mathrm{in}}\ d_2
\end{array}
$$

$$
\begin{array}{lcl}
\text{reify } (\downarrow) & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{TLT} \\
\downarrow^{\mathsf{base}} v & := & v \\
\downarrow^{A/a \to B/b} v & := & \underline{\lambda} x_1^\diamond.\underline{S} k_1^\diamond.\, \overline{\langle_2} \downarrow^b \overline{\langle \mathrm{let}\ v_1 = (v \,\overline{@}\, \uparrow_A x_1^\diamond) \overline{\mathrm{in}}\ (\overline{S_2} k_2.\underline{\mathrm{let}}\ g^\diamond = (k_1^\diamond \,\underline{@}\, \downarrow^B v_1)\ \underline{\mathrm{in}}\ k_2 \,\overline{@}\, \uparrow_a g^\diamond) \rangle} \overline{\rangle_2} \\
\downarrow^{A \to_p B} v & := & \underline{\lambda} x_1^\diamond.\, \overline{\langle_2} \downarrow^B (v \,\overline{@}\, \uparrow_A x_1^\diamond) \overline{\rangle_2}
\end{array}
$$

$$
\begin{array}{lcl}
\text{reflect } (\uparrow) & : & \mathsf{Type} \to \mathsf{Dynamic} \to \mathsf{TLT} \\
\uparrow_{\mathsf{base}} e & := & e \\
\uparrow_{A/a \to B/b} e & := & \overline{\lambda} v_1.\overline{S} k_1.\overline{S_2} k_2.\underline{\mathrm{let}}\ g^\diamond = \underline{\langle \mathrm{let}\ x_1^\diamond = (e \,\underline{@}\, \downarrow^A v_1)\ \underline{\mathrm{in}}\ \overline{\langle_2} \downarrow^a k_1 \,\overline{@}\, \uparrow_B x_1^\diamond \overline{\rangle_2} \rangle}\ \underline{\mathrm{in}}\ k_2 \,\overline{@}\, \uparrow_b g^\diamond \\
\uparrow_{A \to_p B} e & := & \overline{\lambda} v_1.\overline{S_2} k_2.\underline{\mathrm{let}}\ g^\diamond = (e \,\underline{@}\, \downarrow^A v_1)\ \underline{\mathrm{in}}\ k_2 \,\overline{@}\, (\uparrow_B g^\diamond)
\end{array}
$$

**図 11**　shift/reset 付き call-by-value の TDPE
**Fig. 11**　The call-by-value TDPE with shift/reset.

$$
\begin{array}{lcl}
\text{reify } (\downarrow) & : & \mathsf{Type} \to \mathsf{Static} \to \mathsf{TLT} \\
\downarrow^{\mathsf{base}} v & := & v \\
\downarrow^{A/a \to B/b} v & := & \underline{\lambda} x_1^\diamond.\underline{S} k_1^\diamond.v \,\overline{@}\, (\uparrow_A x_1^\diamond) \,\overline{@}\, \overline{\lambda} v_1.\overline{\lambda} k_2.(\underline{\mathrm{let}}\ g^\diamond = (k_1^\diamond \,\underline{@}\, \downarrow^B v_1)\ \underline{\mathrm{in}}\ (k_2 \,\overline{@}\, \uparrow_a g^\diamond)) \,\overline{@}\, \overline{\lambda} v_2.\downarrow^b v_2 \\
\downarrow^{A \to_p B} v & := & \underline{\lambda} x_1^\diamond.v \,\overline{@}\, (\uparrow_A x_1^\diamond) \,\overline{@}\, (\overline{\lambda} v_1.\overline{\lambda} k_2.k_2 \,\overline{@}\, v_1) \,\overline{@}\, (\overline{\lambda} v_2.\downarrow^B v_2)
\end{array}
$$

$$
\begin{array}{lcl}
\text{reflect } (\uparrow) & : & \mathsf{Type} \to \mathsf{Dynamic} \to \mathsf{TLT} \\
\uparrow_{\mathsf{base}} e & := & e \\
\uparrow_{A/a \to B/b} e & := & \overline{\lambda} v_1.\overline{\lambda} k_1.\overline{\lambda} k_2.\underline{\mathrm{let}}\ g^\diamond = \underline{\langle (\underline{\mathrm{let}}\ x_1^\diamond = (e \,\underline{@}\, \downarrow^A v_1)\ \underline{\mathrm{in}}\ (k_1 \,\overline{@}\, (\uparrow_B x_1^\diamond) \,\overline{@}\, (\overline{\lambda} v_2.\downarrow^a v_2))) \rangle}\ \underline{\mathrm{in}}\ (k_2 \,\overline{@}\, \uparrow_b g^\diamond) \\
\uparrow_{A \to_p B} e & := & \overline{\lambda} v_1.\overline{\lambda} k_1.\overline{\lambda} k_2.\underline{\mathrm{let}}\ g^\diamond = (e \,\underline{@}\, \downarrow^A v_1)\ \underline{\mathrm{in}}\ k_1 \,\overline{@}\, (\uparrow_B g^\diamond) \,\overline{@}\, (\overline{\lambda} v_2.k_2 \,\overline{@}\, v_2)
\end{array}
$$

**図 12**　shift/reset 付き call-by-value の 2CPS TDPE
**Fig. 12**　The call-by-value 2CPS TDPE with shift/reset.

この得られる結果は，どちらを使っても同一，つまりどちらも normal form の定義で示される直接形式で得られる．

## 6. $\lambda_{cbv}^{S/R}$ における TDPE の抽出

本章では $\lambda_{cbv}^{S/R}$ における TDPE の抽出について説明を行う．基本的には前章と同じアプローチで，さらにもう一度 CPS 変換を行う．まずは，Tsushima らの call-by-value の shift/reset 付き $\lambda$ 計算における TDPE の定義を**図 11** に示す．前章までとは違い，shift/reset が入力のプログラムに入ってくると，関数の型は $t_1/t_3 \to t_2/t_4$ という形になる [5]．これは $t_1$ 型の式を受け取ると $t_2$ 型の値を返し，その実行にともない継続の返す型 $t_3$ を $t_4$ に変化させるような関数の型である．$t_1/t_3 \to t_2/t_4$ は CPS 変換すると $t_1 \to (t_2 \to t_3) \to t_4$ という型に対応する．一般に shift/reset を使ったプログラムでは，通常の CPS のプログラムとは違って $t_3$ と $t_4$ は同じ型になるとは限らない．以後，継続の返す型（継続の実行により得られる式が持つ型，言い換えると実行中の式を取り囲むコンテキストの型のことで，上の例でいうところの $t_3$ と $t_4$）のことをアンサタイプと呼ぶ．図 11 では，$t_1/t_3 \to t_2/t_4$ という型以外に，従来の関数の型に対応する $t_1 \to_p t_2$ という型も定義している．これは，$\forall r.t_1/r \to t_2/r$ という型のことで，継続を操作せずアンサタイプを変化させないような関数の型を示す．そのような型を持つ term のことを pure であるという．

ユーザからの入力プログラムに shift/reset が入ってくると，その挙動をサポートするのと let-insertion をするのとの両方で shift/reset を使う必要が出てくる．しかし，同じ shift/reset を使うと両者が干渉して正しい部分評価結果を得ることができなくなる．Tsushima らはこれに対応するために，前者に shift/reset を使い，後者は状態を使った let-insertion を行っている．しかし，状態を使った let-insertion は，状態に対する副作用を使うため定式化するのは簡単ではない．そこで，ここでは CPS 階層 [6] の 2 段目の $\mathrm{shift}_2/\mathrm{reset}_2$ を使うことで対応する．$\mathrm{shift}_2/\mathrm{reset}_2$ は，shift/reset を使ったプログラムもひっくるめて捕捉することができ，ユーザの入力中の shift/reset の動きに干渉することなく let-insertion を行うことができるようになる．図 11 では，（static な）$\mathrm{shift}_2/\mathrm{reset}_2$ を $\overline{S_2}$，$\overline{\langle_2 \rangle_2}$ と表記している．図 11 ではさらに static な let 文 $\overline{\mathrm{let}}\ x = d_1 \overline{\mathrm{in}}\ d_2$ も使っているが，これは $(\overline{\lambda} x.d_2) \,\overline{@}\, d_1$ と同じことである．この TDPE に shift/reset の入った式を入力して実行すると，部分評価して消えなかった shift/reset は dynamic な term に含まれたまま出力される．

しかし，shift/reset と同様，$\mathrm{shift}_2/\mathrm{reset}_2$ も Coq では使用することができない．そこで，この定義を 2 回，CPS 変

$$\begin{array}{rrcl} \text{type} : & \mathsf{typ} \ni A & := & \mathsf{base} \mid A_1/A_3 \to A_2/A_4 \mid A_1 \to_p A_2 \\ \text{environment} : & \mathsf{world} \ni w, \Gamma & := & \text{list of } \mathsf{typ} \end{array}$$

$$\text{term} : \quad \frac{}{\mathsf{hyp} : A, w; r \vdash_{\mathsf{t}} A; r} \quad \frac{v : w; a \vdash_{\mathsf{t}} A; b}{\mathsf{wkn}(v) : B, w; a \vdash_{\mathsf{t}} A; b} \quad \frac{p : A, w; a \vdash_{\mathsf{t}} B; b}{\mathsf{lam}(p) : w; r \vdash_{\mathsf{t}} A/a \to B/b; r}$$

$$\frac{p : w; r \vdash_{\mathsf{t}} c/a \to A/b; d \quad q : w; b \vdash_{\mathsf{t}} c; r}{\mathsf{app}(p, q) : w; a \vdash_{\mathsf{t}} A; d} \quad \frac{p : A \to_p a, w; c \vdash_{\mathsf{t}} c; b}{\mathsf{shift}(p) : w; a \vdash_{\mathsf{t}} A; b} \quad \frac{p : w; a \vdash_{\mathsf{t}} a; A}{\mathsf{reset}(p) : w; r \vdash_{\mathsf{t}} A; r}$$

図 13　$\lambda_{cbv}^{S/R}$ の型システム
**Fig. 13**　Type system of $\lambda_{cbv}^{S/R}$.

$$\text{(pure) normal form} : \quad \frac{p : w \vdash_{\mathsf{ne}} A}{p : w \vdash_{\mathsf{nf}} A} \quad \frac{p : B \to_p a, A, w \vdash_{\mathsf{nf}} b}{\mathsf{lam}(\mathsf{shift}(p)) : w \vdash_{\mathsf{nf}} A/a \to B/b} \quad \frac{p : A, w \vdash_{\mathsf{nf}} B}{\mathsf{lam}(p) : w \vdash_{\mathsf{nf}} A \to_p B}$$

$$\frac{}{\mathsf{hyp} : A, w \vdash_{\mathsf{ne}} A} \quad \frac{p : w \vdash_{\mathsf{ne}} A}{\mathsf{wkn}(p) : B, w \vdash_{\mathsf{ne}} A}$$

$$\text{pure neutral term} : \quad \frac{q : w \vdash_{\mathsf{ne}} c' \to_p c \quad q' : w \vdash_{\mathsf{nf}} c' \quad p : c, w \vdash_{\mathsf{nf}} A}{\mathsf{let}(\mathsf{app}(q, q'), p) : w \vdash_{\mathsf{ne}} A} \quad \frac{p : w; a' \vdash_{\mathsf{nex}} a'; c \quad q : c, w \vdash_{\mathsf{nf}} A}{\mathsf{let}(\mathsf{reset}(p), q) : w \vdash_{\mathsf{ne}} A}$$

$$\text{impure neutral term} : \quad \frac{p : w \vdash_{\mathsf{ne}} A}{p : w; a \vdash_{\mathsf{nex}} A; a} \quad \frac{q : w; b' \vdash_{\mathsf{nex}} c'/a \to c/b'; r \quad q' : w \vdash_{\mathsf{nf}} c' \quad p : c, w \vdash_{\mathsf{nf}} A}{\mathsf{let}(\mathsf{app}(q, q'), p) : w; a \vdash_{\mathsf{nex}} A; r}$$

図 14　$\lambda_{cbv}^{S/R}$ の normal form と neutral term
**Fig. 14**　normal forms and neutral terms for $\lambda_{cbv}^{S/R}$.

換することでこれらを除去する．最初の CPS 変換では，shift/reset が除去され，$\text{shift}_2/\text{reset}_2$ は shift/reset で実現できるようになる．さらに，もう一度 CPS 変換をすると，その shift/reset も除去される．そのようにして 2 回 CPS 変換をかけて，static な shift/reset をはずしたのが**図 12** の定義である．$k_1$ が継続，$k_2$ がメタ継続を表す変数となっている．このように CPS 変換を 2 回施して，2 階層の継続が出てくる形式を 2CPS のプログラムと呼ぶ．この定義を参考にして定式化を行っていく．$\lambda_{cbv}^{S/R}$ のシステムや証明の定式化すべてが $\lambda_{cbn}$，$\lambda_{cbv}$ のそれとまったく同じ流れになっており，$\lambda_{cbv}$（さらに遡れば $\lambda_{cbn}$）の定式化の拡張となっている．

### 6.1　$\boldsymbol{\lambda_{cbv}^{S/R}}$ の定義

まず $\lambda_{cbv}^{S/R}$ を**図 13** で定義する．これは Danvy らによる shift/reset に対する型システム [5] を多相に拡張した Asai らの型システム [2] に基づいており，$\lambda_{cbn}$ や $\lambda_{cbv}$ と比べて term（構文木）の根が $p : w \vdash_{\mathsf{t}} A$ という三つ組から $p : w; a \vdash_{\mathsf{t}} A; b$ という五つ組に拡張されている．この根から構成される term は環境 $w$ において $A$ 型を持っており，この term を実行するとアンサタイプが $a$ から $b$ へと変化する．また，図 11，図 12 と同様，多相でかつ pure である関数型 $A_1 \to_p A_2$ を導入し，shift の束縛変数の型（継続の型）が $\_ \to_p \_$ 型となるように定義している．この多相の関数型を導入した理由は，後述する completeness の証明において使用する必要が生じたためである．

### 6.2　$\boldsymbol{\lambda_{cbv}^{S/R}}$ の Kripke モデル

$\lambda_{cbv}^{S/R}$ で使用する Kripke モデルを定める．$\lambda_{cbv}^{S/R}$ で定義する Kripke モデルは，$\lambda_{cbv}$ でのそれを拡張した形となっている．まず，normal form と neutral term を**図 14** のように定義する．$\lambda_{cbv}^{S/R}$ では，neutral term が pure である場合（$\vdash_{\mathsf{ne}}$）と pure でない場合（$\vdash_{\mathsf{nex}}$）の 2 種類に分かれる（TDPE の実行結果において impure な normal form は登場しないため，normal form は pure の場合のみあれば十分である）．$\mathsf{lam}(\mathsf{shift}(\_))$ は $\lambda$ の直下に shift が入っているような項を示しており，$A/a \to B/b$ 型の関数の部分評価結果がこのような形になる．pure でない関数の部分評価結果に現れる $\lambda$ の下には必ず shift が現れるというのは，shift/reset の通常の部分評価 [1] においても同様である．一方，$\mathsf{lam}(\_)$ は pure な関数を TDPE した結果として現れる．

これら normal form，neutral term を用いて定義された $\lambda_{cbv}^{S/R}$ における論理述語を**図 15** に示す．これも定義が 2CPS になっている点以外はこれまでと同様である．型 $A/a \to B/b$ は，一度 CPS 変換すると $A \to (B \to a) \to b$ となるが，これをさらにもう一度（アンサタイプが任意のもとで）CPS 変換すると $A \to (B \to \forall T.(a \to T) \to T) \to \forall T'.(b \to T') \to T'$ となる．図 15 の論理述語は，この型と同じ構造になってい

る．特に，$(B \to \forall T.(a \to T) \to T)$ に対応する部分が継続，$a \to T$ と $b \to T'$ に対応する部分がメタ継続である．

この論理述語の base のケースの定義は $w \vdash_{\mathsf{ne}} \mathsf{base}$ であるため，この論理述語が Kripke モデルの forcing の関係を満たすには，やはり $w \vdash_{\mathsf{ne}} \mathsf{base}$ が monotone でなくてはならない．$\lambda_{cbn}$，$\lambda_{cbv}$ の場合と同様，$w \vdash_{\mathsf{ne}} A$ に関しては以下の性質が成立する．

**補題 4（monotonicity, $\lambda_{cbv}^{S/R}$）**

$$\forall w, \forall w', \forall A, w \le w' \Rightarrow w \vdash_{\mathsf{ne}} A \Rightarrow w' \vdash_{\mathsf{ne}} A$$

この補題の $A$ を base に置き換えれば，$w \vdash_{\mathsf{ne}} \mathsf{base}$ における monotonicity となる．ゆえに図 15 で定義した $\models$ は Kripke モデル上の forcing の関係となっている．

### 6.3 $\lambda_{cbv}^{S/R}$ の soundness の証明

Soundness の定義は以下のように，$\lambda_{cbv}$ での soundness 定理をもう一度 CPS 変換した形となっている．

**定理 5（Soundness, $\lambda_{cbv}^{S/R}$）**

$$\begin{aligned}
&\forall A, \forall a, \forall b, \forall \Gamma, \Gamma; a \vdash_{\mathsf{t}} A; b \Rightarrow \forall w,\ w \models_s \Gamma \Rightarrow \\
&(\forall w_1, w \le w_1 \Rightarrow w_1 \models A \Rightarrow \\
&\ \forall T_1, (\forall w_2, w_1 \le w_2 \Rightarrow w_2 \models a \Rightarrow \\
&\qquad w_2 \vdash_{\mathsf{nf}} T_1) \Rightarrow \\
&\qquad w_1 \vdash_{\mathsf{nf}} T_1) \Rightarrow \\
&\forall T, (\forall w_1, w \le w_1 \Rightarrow w_1 \models b \Rightarrow w_1 \vdash_{\mathsf{nf}} T) \Rightarrow \\
&\quad w \vdash_{\mathsf{nf}} T
\end{aligned}$$

この定理も，これまでの soundness の定理と同じように項に関する帰納法で証明できる．その結果，得られるプログラムは 2CPS で書かれたインタプリタになる．これは，本章の Kripke モデルにおける $A/a \to B/b$ 型の term の意味が 2CPS で与えられているためである．

### 6.4 $\lambda_{cbv}^{S/R}$ の completeness の証明

次に，TDPE を抽出するための completeness の証明を行う．定理の定義自体は，本質的に $\lambda_{cbn}$，$\lambda_{cbv}$ の定義と同じである．

**定理 6（Completeness, $\lambda_{cbv}^{S/R}$）**

$$\begin{aligned}
&\forall A, \quad (i) \quad \forall w, w \models A \Rightarrow w \vdash_{\mathsf{nf}} A \\
&\qquad (ii) \quad \forall w, w \vdash_{\mathsf{ne}} A \Rightarrow w \models A
\end{aligned}$$

定理 6 の証明は，本章のはじめに図 12 で示した 2CPS の TDPE の定義にそって行うことができる．証明から得られた（Coq 上の）プログラムを**図 16** に示す．`nf_LamShift` は（pure な）normal form の定義における $\mathsf{lam}(\mathsf{shift}(\_))$ を，`ne_LetReset` は pure な neutral term である $\mathsf{let}(\mathsf{reset}(\_), \_)$ を表す．また，`ne_LetApp` は pure な neutral term である $\mathsf{let}(\mathsf{app}(\_, \_), \_)$ を，`nex_LetApp` は impure な neutral term である $\mathsf{let}(\mathsf{app}(\_, \_), \_)$ をそれぞれ表している．`arrowP`, `arrow` はそれぞれ $(\_ \to_p \_)$，$(\_/\_ \to \_/\_)$ を表す．図 16 の関数定義においても，図 12 の CPS の shift/reset 付き TDPE と等しいであろうことが見て取れる．

$$\begin{aligned}
w \models \mathsf{base} &\iff w \vdash_{\mathsf{ne}} \mathsf{base} \\
w \models A/a \to B/b &\iff \forall w_1, w \le w_1 \Rightarrow w_1 \models A \Rightarrow \\
&\quad (\forall w_2, w_1 \le w_2 \Rightarrow w_2 \models B \Rightarrow \\
&\quad\ \forall T, (\forall w_3, w_2 \le w_3 \Rightarrow w_3 \models a \Rightarrow \\
&\qquad w_3 \vdash_{\mathsf{nf}} T) \Rightarrow \\
&\qquad w_2 \vdash_{\mathsf{nf}} T) \Rightarrow \\
&\quad \forall T', (\forall w_2, w_1 \le w_2 \Rightarrow w_2 \models b \Rightarrow \\
&\qquad w_2 \vdash_{\mathsf{nf}} T') \Rightarrow \\
&\qquad w_1 \vdash_{\mathsf{nf}} T' \\
w \models A \to_p B &\iff \forall w_1, w \le w_1 \Rightarrow w_1 \models A \Rightarrow \\
&\quad (\forall w_2, w_1 \le w_2 \Rightarrow w_2 \models B \Rightarrow \\
&\quad\ \forall T, (\forall w_3, w_2 \le w_3 \Rightarrow w_3 \models B \Rightarrow \\
&\qquad w_3 \vdash_{\mathsf{nf}} T) \Rightarrow \\
&\qquad w_2 \vdash_{\mathsf{nf}} T) \Rightarrow \\
&\quad \forall T', (\forall w_2, w_1 \le w_2 \Rightarrow w_2 \models B \Rightarrow \\
&\qquad w_2 \vdash_{\mathsf{nf}} T') \Rightarrow \\
&\qquad w_1 \vdash_{\mathsf{nf}} T' \\
w \models_s (A_1, ..., A_n) &\iff (w \models A_1) \wedge \cdots \wedge (w \models A_n)
\end{aligned}$$

**図 15**　$\lambda_{cbv}^{S/R}$ で使用する論理述語
**Fig. 15**　Logical predicates for $\lambda_{cbv}^{S/R}$.

なお，多相性を用いなければ証明できなかった箇所の proof term は，`reify` の $A_1/A_3 \to A_2/A_4$ のケースにおける式（`(wkn_ne H (ne_Hyp (A1 :: w) (arrowP A2 A3)))`）である．この式は図 11，図 12 の同ケースにおける let 式の中に登場する $k_1^\circ$ に対応している．この $k_1^\circ$ のアンサタイプが多相でなくてはならないのは，$k_1^\circ$ が初めて現れる $\underline{S}k_1^\circ$ の部分では，$k_1^\circ$ がどのようなコンテキストで使われるか不明であるためである．実際，$k_1^\circ$ が使われているのは図 11 の $\overline{\langle_2 ... \rangle_2}$ のコンテキストの中だが，そこでさらに $\downarrow^b$ が呼ばれており，$b$ が関数型だとその中でさらに別のコンテキストが作られている．

### 6.5 $\lambda_{cbv}^{S/R}$ 用の reify への入力について

$\lambda_{cbv}$ のときと同様，$\lambda_{cbv}^{S/R}$ 用の reify への入力は $\lambda_{cbv}^{S/R}$ 用の（soundness の証明によって与えられる）意味にそったものでなくてはならない．具体的には，2CPS の格好の static な term となる．したがって，shift/reset を含んだ入力プログラムを TDPE にかけたかったら，一度，soundness の証明に対応する 2CPS インタプリタに通し，2CPS における内部表現を得てから，それを reify に渡す格好になる．reify に渡す内部表現は 2CPS になるが，最終的に得られる結果は shift/reset を含んだ直接形式であり，結果が 2CPS になってしまうことはない．

## 7. 関連研究

Filinski は TDPE の正当性を示すために，論理関係を

```
Fixpoint reify (A: typ) : (forall w, R w A -> nf w A) :=
  match A return (forall w, R w A -> nf w A) with
| base => fun _ v => nf_ne v
| arrow A1 A2 A3 A4 => fun w v =>
   nf_LamShift (v (arrowP A2 A3 :: A1 :: w) A4
                  (lew_trans (lew_cons (arrowP A2 A3) (lew_cons A1 (lew_refl w)))
                             (lew_refl (arrowP A2 A3 :: A1 :: w)))
                  (reflect (ne_Wkn (arrowP A2 A3) (ne_Hyp w A1)))
                  (fun w2 _ (H: arrowP A2 A3 :: A1 :: w <== w2) v1 k2 =>
                   nf_ne (ne_LetApp (wkn_ne H (ne_Hyp (A1 :: w) (arrowP A2 A3)))
                                    (reify A2 w2 v1)
                                    (k2 (A3 :: w2) (lew_cons A3 (lew_refl w2))
                                        (reflect (ne_Hyp w2 A3)))))
                  (fun w2 _ (v2: R w2 A4) => reify A4 w2 v2))
| arrowP A1 A2 => fun w v =>
   nf_Lam (v (A1 :: w) A2 (lew_cons A1 (lew_refl w))
             (reflect (ne_Hyp w A1))
             (fun w2 _ _ v1 k2 => k2 w2 (lew_refl w2) v1)
             (fun w2 _ (v2 : R w2 A2) => reify A2 w2 v2))
  end
with reflect (A: typ) : forall w, ne w A -> R w A :=
  match A return (forall w, ne w A -> R w A) with
| base => fun _ e => e
| arrow A1 A2 A3 A4 => fun w e =>
   fun w1 _ (H: w <== w1) v1 k1 k2 =>
   nf_ne (ne_LetReset (nex_LetApp (nex_ne (wkn_ne H e) A4) (reify A1 w1 v1)
                                  (k1 (A2 :: w1) A3 (lew_cons A2 (lew_refl w1))
                                      (reflect (ne_Hyp w1 A2))
                                      (fun w3 _ (X2: R w3 A3) => reify A3 w3 X2)))
                      (k2 (A4 :: w1) (lew_cons A4 (lew_refl w1))
                          (reflect (ne_Hyp w1 A4))))
| arrowP A1 A2 => fun w e =>
   fun w1 T1 (H: w <== w1) v1 k1 k2 =>
   nf_ne (ne_LetApp (wkn_ne H e) (reify A1 w1 v1)
                    (k1 (A2 :: w1) T1 (lew_cons A2 (lew_refl w1))
                        (reflect (ne_Hyp w1 A2))
                        (fun w3 H1 v2 =>
                         k2 w3 (lew_trans (lew_cons A2 (lew_refl w1)) H1) v2)))
  end.
```

**図 16**　$\lambda_{cbv}^{S/R}$ の抽出された TDPE (reify/reflect)
**Fig. 16**　The extracted TDPE (reify/reflect) for $\lambda_{cbv}^{S/R}$.

使った証明を行っている [9]．また，それを拡張して，任意の monadic effect を持つ体系における TDPE の正当性も示している [10]．アンサタイプが変化しないような shift/reset は monadic effect の 1 つなので，Filinski の結果を使えば，そのような shift/reset 用の TDPE を得られるはずである．しかし，証明は簡単ではなく，現在のところ証明を再構成することができていない．我々の証明は Filinski の証明（の一部）を Coq で定式化した形になっているのではないかと予想しているが，詳細は不明である．

本論文の証明の足がかりを与えたのは Ilik の仕事である [12], [13]．まず初めに Coquand [4] によって，Kripke モデルの推論規則に対する completeness の証明と λ 計算における normalization by evaluation とが対応しうることが示され，Ilik はその研究を基に，Kripke モデルを構成し completeness の証明から TDPE を抽出するという着想をなした．彼は Kripke モデルが直観主義論理，さらには古典論理に対して完全であることを示し，その証明が TDPE に対応していることを示した．そしてそれを発展させて shift/reset に対する TDPE も得ている [14]．

本研究と Ilik の研究 [12], [13], [14] で異なる点は以下のとおりである．まず，shift/reset なしの call-by-value の TDPE の抽出については，TDPE 抽出のため

に使用する Kripke モデルの forcing の定義が大きく異なっている．本研究で定義した（call-by-value の）forcing の構造は $A \Rightarrow \forall T.(B \Rightarrow T) \Rightarrow T$ という一般的な CPS の形をしているが，Ilik による forcing の構造は $(\forall T.(A \Rightarrow T) \Rightarrow T) \Rightarrow \forall T'.(B \Rightarrow T') \Rightarrow T'$ となっており，これは一般的な CPS の形とは異なる．したがって導出される TDPE の構造も，本研究とは異なるものとなっている．次に shift/reset 付きの TDPE の抽出に関してだが，Ilik の定義する shift/reset 付き $\lambda$ 計算は論理学に基づいて考えられているもので，アンサタイプが固定されているうえに，reset の持つ型が atomic type のみに限定されている．一方で，本研究で使用した型システムは，通常扱われる Danvy によって提案された型システムに準拠した定義となっている．Kripke モデルにおける forcing については，本研究で shift/reset 付き TDPE の抽出に用いた forcing の構造は 2CPS の形をしているが，Ilik の forcing の構造は 1CPS の形になっており，結果として得られた TDPE プログラムは，本研究と Ilik ではまったく異なるものとなっている．彼の証明は Coq で定式化されているものの，証明の抽出は手で行われている．さらに彼の論文 [14] には，抽出した TDPE から得られた部分評価結果の例が載っているが，Kameyama らの公理系 [15] において同じとは認められないものが出てきている等不可解な点がある．本論文は，Ilik の仕事をプログラミング言語の立場から従来の shift/reset で再構築したものと考えることができる．

TDPE に関するより理論的な研究としては，Fiore [11] による圏論をベースとした意味論の構築，Balat ら [3] による sum 型をサポートしたもの等があげられる．また，より実際的な方向性として Lindley [16] は SML.NET の中間言語のコンパイラとして TDPE を使っている．

## 8. まとめ

本論文では，各種の TDPE を Kripke モデルの推論規則に対する completeness の証明から抽出した．証明は完全に Coq で定式化されており，Danvy のオリジナルの call-by-name の TDPE だけでなく，let-insertion を行う call-by-value の TDPE，さらに shift/reset を扱える TDPE についても定式化した．Coq による証明は複雑になりがちだが，本論文で紹介している証明は簡潔であり，証明を表す項はそのまま TDPE のプログラムと解釈できるレベルである．これをもとに，さらに TDPE のいろいろな性質を検討していく土台を築けたものと考えている．

抽出された TDPE のプログラムは，それ自体，興味深い格好をしている．特に，fresh な変数を生成する際には，持ち歩いている環境と重ならなければ大丈夫であることが理解できる．これまで fresh な変数を生成するには，グローバルなカウンタを用意するなどの方法がとられていたが，それとは別のアプローチを示唆するものとなっている．また，このように簡潔な形で TDPE を抽出できることが分かると，新たな TDPE を構築する際にも役に立つと思われる．TDPE プログラムにおいて static/dynamic な shift/reset を複雑に使うようになると，CPS 変換を行って static な shift/reset のみを取り除く作業も複雑になり，間違いも入りやすくなる．しかし，一度，このような形で TDPE を構築できることが分かると，逆に証明から正しい TDPE の形を割り出すことができるようになってくると思われる．

本論文では 3 つの体系の TDPE を抽出したが，それらは論理述語の定義を除くととても似た形をしていることが分かる．これらをたとえばモナドのような形で表現して，統一的に扱えるようになると面白い．これは今後の課題である．

**謝辞**　査読者からいただいた種々のコメントは本論文全般の改善にとって誠に有益でありました．ここに深く感謝申し上げます．なお，本研究は科研費（25280020）の助成を受けたものであります．

### 参考文献

[1] Asai, K.: Logical Relations for Call-by-value Delimited Continuations, *Trends in Functional Programming* (*TFP 2005*), Vol.6, pp.63–78, Intellect (2007).
[2] Asai, K. and Kameyama, Y.: Polymorphic Delimited Continuations, *Proc. 5th Asian Symposium on Programming Languages and Systems* (*APLAS'07*), LNCS 4807, pp.239–254 (Nov. 2007).
[3] Balat, V., Cosmo, R.D. and Fiore, M.: Extensional Normalization and Type-Directed Partial Evaluation for Typed Lambda Calculus with Sums, *Proc. 31st ACM Symposium on Principles of Programming Languages*, pp.64–76 (Jan. 2004).
[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*, Vol.15, Issue 1, pp.57–90 (Mar. 2002).
[5] Danvy, O. and Filinski, A.: A Functional Abstraction of Typed Contexts, Technical Report 89/12, DIKU, University of Copenhagen (July 1989).
[6] Danvy, O. and Filinski, A.: Abstracting Control, *ACM Conference on Lisp and Functional Programming*, pp.151–160 (June 1990).
[7] Danvy, O.: Type-Directed Partial Evaluation, *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp.242–257 (Jan. 1996).
[8] Danvy, O.: Type-Directed Partial Evaluation, *Partial Evaluation, Practice and Theory*, Hatcliff, J., Mogensen, T.Æ. and Thiemann, P. (Eds.), LNCS 1706, pp.367–411 (1999).
[9] Filinski, A.: A Semantic Account of Type-Directed Partial Evaluation, *Principles and Practice of Declarative Programming*, Nadathur, G. (Ed.), LNCS 1702, pp.378–395 (Sep. 1999).
[10] Filinski, A.: Normalization by Evaluation for the Computational Lambda-Calculus, *Typed Lambda Calculi and Applications*, Abramsky, S. (Ed.), LNCS 2044, pp.151–165 (May 2001).
[11] Fiore, M.: Semantic Analysis of Normalisation by Eval-

uation for Typed Lambda Calculus, *Proc. 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (*PPDP'02*), pp.26–37 (Oct. 2002).
[12] Ilik, D.: Constructive Completeness Proofs and Delimited Control, Ph.D. thesis, Ecole Polytechnique X (Oct. 2010).
[13] Ilik, D.: Continuation-passing style models complete for intuitionistic logic, *Annals of Pure and Applied Logic, Special issue: Classical logic and computation 2010*, pp.651–662 (June 2013).
[14] Ilik, D.: A formalized type-directed partial evaluator for shift and reset, *Control Operators and their Semantics*, p.18 (June 2013), available from ⟨http://arxiv.org/abs/1210.2094⟩.
[15] Kameyama, Y. and Hasegawa, M.: A Sound and Complete Axiomatization of Delimited Continuations, *Proc. 8th ACM SIGPLAN International Conference on Functional Programming* (*ICFP'03*), pp.177–188 (Aug. 2003).
[16] Lindley, S.: Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages, Ph.D. thesis, University of Edinburgh (2005).
[17] Masuko, M. and Asai, K.: Caml Light + shift/reset = Caml Shift, *Theory and Practice of Delimited Continuations* (*TPDC 2011*), pp.33–46 (May 2011).
[18] Pierce, B.C.: *Types and Programming Languages*, MIT Press, Cambridge (2002).
[19] Tsushima, K. and Asai, K.: Towards Type-Directed Partial Evaluation for Shift and Reset, *Proc. 2009 Workshop on Normalization by Evaluation*, pp.57–64 (Aug. 2009).
[20] Yang, Z.: Encoding Types in ML-like Languages, *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming* (*ICFP'98*), pp.289–300 (Sep. 1998).

## 付　　録

### A.1 一般の de Bruijn index term への変換プログラム

本研究の de Bruijn index term を一般の de Bruijn index term へと変換する（Coq 上の）関数 `normalize` は以下のように定義できる．

```
Fixpoint add_wkn w1 w2 A (H: Rs w1 w2)
 : Rs (A :: w1) w2 :=
  match H with
| Rs_nil w => Rs_nil (A :: w)
| Rs_cons _ _ _ t X =>
    Rs_cons (tm_Wkn A t) (add_wkn A X)
  end.

Fixpoint normalize (G: world) (A: typ)
                   (t: tm G A)
 : forall (w: world), Rs w G -> tm w A :=
  match t with
| tm_Hyp _ _ => fun _ env =>
    get_first env
| tm_Wkn _ _ _ t1 => fun _ env =>
    normalize t1 (get_rest env)
| tm_Lam _ A _ t1 => fun w1 env =>
    tm_Lam (normalize t1
            (Rs_cons (tm_Hyp w1 A)
                     (add_wkn A env)))
| tm_App _ _ _ t1 t2 => fun _ env =>
    tm_App (normalize t1 env) (normalize t2 env)
  end.
```

また，本研究で用いる normal form，neutral term は，通常の de Bruijn index を用いた表現に変換したあとも normal form，neutral term となっている．特に，neutral term に出てくる wkn(_)（複数の wkn(_) が入れ子になっていてもよい）が app(_, _) にかかっていた場合にも，wkn(_) を中に潜らせれば最終的に app(_, _) となるため，neutral term になっていることが確認できる．

**廣田 知子**

2008 年お茶の水女子大学理学部情報科学科卒業．2010 年同大学大学院博士前期課程修了．同年同大学院博士後期課程入学．

**浅井 健一**　（正会員）

1992 年東京大学大学院理学系研究科情報科学専攻修士課程修了．1994 年同大学助手．博士（理学）．1998 年より 2001 年まで科学技術振興事業団さきがけ研究 21 研究員を兼任．2001 年お茶の水女子大学助教授（現，准教授）．現在に至る．2004 年より 1 年間，米国ノースイースタン大学客員助教授．関数型言語，限定継続，部分評価，自己反映言語等に興味を持つ．ソフトウェア科学会，ACM 各会員．