平成 25 年度　博士学位論文

継続渡し形式の型主導部分評価器における正当性の証明

お茶の水女子大学大学院

人間文化創成科学研究科　理学専攻

廣田 知子

平成 26 年 3 月

要旨

本論文では，数学的定理の存在証明を定理証明支援システム Coq で定式化することにより，正当性の保証された部分評価器の抽出を行う．部分評価器は，与えられたプログラムを，関数の body 部分も含めてこれ以上簡約出来ない形（正規形）にまで評価するプログラム変換器である．簡約可能な部分は全て評価してくれる為，プログラムを部分評価器に通せば，意味は等価でありながらも元のプログラムよりも実行時間の短いプログラムが生成出来る．実際にユーザが部分評価器を使う際，その部分評価器の正当性が保証されているか否かは，部分評価器を通した後のプログラムの信頼性に大きく関わる．

本論文ではまず，強正規化性定理と評価器が Curry Howard 同型対応の関係にあるという，一般的に良く知られている事実に着目し，限定継続命令 shift/reset 付き λ 計算における強正規化性定理の証明を Coq で定式化することにより，評価器プログラムを抽出する．型システムを定式化するにあたっては，Locally Nameless 手法を用いることによって α 同値問題を回避する．定理の証明は Tait 流の論理述語を用いて行う．しかしながら，この方法で得られた評価器プログラムは非常に複雑であり，この手法を単純に拡張するだけでは実用的な部分評価器を得ることは出来ない．

上の問題を解決するために，本論文では以下の手順を行う．(i) 型主導部分評価器（type-directed partial evaluator，TDPE）と Curry Howard 同型対応となっている（論理述語の）completeness 定理の証明を行う．(ii) (i) の証明から得られる TDPE が正当性定理（TDPE 実行前後で入力値の意味が変化しない）を満たすことを証明する．(iii) (i) と (ii) の証明を Coq で定式化することにより，実用的で，かつ正しさの保証された TDPE プログラムを導出する．

TDPE は他の一般的な部分評価器と異なり，term の中身を全く見ずに計算を行う．そのため，実行が非常に高速であるという特徴を持つ．TDPE に関する先行研究として，Tsushima らは shift/reset 付き TDPE を提案しているが，その正当性に関しては未だ議論がされていなかった．他方で Ilik は Kripke モデルの推論規則に対する completeness 定理の証明から TDPE を抽出する方法を提示しており，shift/reset 付き λ 計算体系における TDPE の抽出も行っているが，そこで扱

われている shift/reset は,（Tsushima らが扱っているような）一般的に使われる shift/reset とは異なるものとなっている．そこで，Ilik と Tsushima らの仕事を結びつけることにより，通常の限定継続を扱えるように，Ilik の証明を構築し直す．それにより，Tsushima らの直接形式の shift/reset 付き TDPE を CPS 変換することにより得られる，継続渡し形式（CPS）の shift/reset 付き TDPE を，Coq を用いて抽出する．

しかしながら，上の completeness 定理から得られた TDPE は無限ループが起こらないことは保証されているものの，正当性定理を満たすことは保証されていない．故に本論文では，call-by-name と call-by-value の TDPE における Filinski の正当性定理の証明を Coq で定式化し，そしてそれら定式化手法を 2CPS に拡張することによって,（CPS 変換された）Tsushima らの shift/reset 付き TDPE の正当性定理を，Coq を用いて証明する．ここで使用する意味論の completeness 定理から得られる TDPE と，上の Kripke 意味論における completeness 定理から得られる TDPE との違いは，使用している意味論が異なるだけで，プログラムの挙動は同じものとなっている．又，型システムの定式化については parametric higher-order abstract syntax（PHOAS）を用いることにより，α 同値問題を回避しつつ非常に簡潔な定式化を実現している．但し PHOAS の higher-order の特性により Coq では証明が困難な性質が存在するため，その性質の証明のみ Coq には載せずに公理として定義する．

abstract

This thesis presents various methods of extracting a partial evaluator guaranteed to be correct from the formalization of a constructive existence proof utilizing a well-known proof assistant Coq. A partial evaluator reduces a given program to the normal form that cannot be further simplified even in a function body. Since the partial evaluator is able to evaluate all the reducible parts in a given program, the evaluator produces more efficient and fast program than the original program without changing its meaning. The correctness of partial evaluator is important in actual application to guarantee reliability of programs that have been generated by the partial evaluator.

In this thesis, we first attempt to extract an evaluator from formalization in Coq of strong normalization theorem of λ-calculus with delimited control operators shift/reset, paying attention to a well-known property that strong normalization theorem is Curry-Howard isomorphic to an evaluator. On formalizing of our type system, we adopt the Locally Nameless method which leads us to avoid difficulty of α-renaming problem. Our actual proof is done by fully utilizing logical predicate à la Tait. However, the evaluator obtained at this stage is so complicated that it seems much difficult to apply such an evaluator directly to actual problem.

In order to overcome this difficulty, in this thesis we prove (i) the completeness theorem of logical predicate from which type-directed partial evaluators (TDPEs) are actually derived, and (ii) the correctness property of such evaluators as is derived in (i), and finally, (iii) from the formalization process in Coq of the above, we deduce a concrete program for TDPE that should be correct in application.

Unlike standard partial evaluators, TDPE does not inspect the internal structure of the input program and thus is very fast. For the concept of TDPE, there are a couple of the proceeding studies by Tsushima et al., Ilik and others: Tsushima et al. proposed a TDPE with shift/reset;

Ilik proposed a method of extracting a TDPE from the proof process of the completeness theorem for inference rules in the Kripke model. However, Tsushima et al. unfortunately did not succeed to show the correctness property of their TDPE that seems very crucial in application. Also Ilik did not deal with standard shift/reset as is used in Tsushima et al. To fill this gap, we first re-formalize the method proposed by Ilik so as to deal with standard shift/reset. From this Ilik's method, we extract a continuation-passing style (CPS) TDPE with shift/reset in Coq which can be obtained by transforming direct-style TDPE of Tsushima et al.

However, although this TDPE extracted from completeness theorem is guaranteed to avoid infinite loop, the equivalence of the meaning of programs before and after performing this TDPE leaves still unclear. Therefore, we formalize, in Coq, the proof of Filinski's correctness theorem of the call-by-name TDPE and call-by-value TDPE, and then by extending this formalization to 2CPS, we prove the correctness theorem of CPS TDPE with shift/reset by which the equivalence of the meaning of the programs before and after the TDPE can be in fact guaranteed. The completeness theorem obtained from the semantics results in a similar TDPE to the above TDPE extracted from the completeness theorem of the Kripke semantics. Both the TDPEs have the same meaning in application, though the forms of semantics are slightly different. The use of parametric higher-order abstract syntax (PHOAS) makes it possible to establish the very simple formalization in Coq of our type system, leading in effect to avoid α-renaming problem. When proving the correctness theorem using PHOAS, we are confronted with a particular property that may be difficult to demonstrate in Coq because of the higher-order nature of PHOAS.

目 次

第1章　序論

部分評価器とは，与えられたプログラムを正規形（normal form，これ以上簡約出来ない形の term）にまで評価するプログラム変換器である．特に，単なる評価器では関数の body を触ることはないが，部分評価器では body 部分の簡約も行う．例えば $\lambda x.(\lambda y.y)\ x$ という関数が与えられたとき，関数の body $((\lambda y.y)\ x)$ も評価して，答えとして恒等関数 $\lambda x.x$ を返す．プログラムを部分評価器に通せば，意味は等価でありながらも元のプログラムよりも実行時間の短いプログラムが生成出来る．

実際にユーザが部分評価器を使う際，その部分評価器の正しさが保証されているか否か，つまり渡したプログラムの本当の正規形を（無限ループを起こすことなく）計算することが保証されているか否かは，部分評価器を通した後のプログラムの信頼性に大きく関わる．先行研究により，正当性の証明には様々なアプローチが考えられる．本論文では，Curry Howard 同型概念を利用した（部分評価器が持つべき）性質の存在証明から正しさの保証された部分評価器プログラムを抽出するというアプローチに着目し，実際に定理証明支援システム Coq を用いて存在証明を定式化することにより，正当性の保証された部分評価器プログラムの抽出を行う．本論文では，call-by-value（CBN）と call-by-name（CBV）の単純型付き λ 計算，そして限定継続命令 shift/reset 付き CBV の型付き λ 計算を対象とする．

プログラム抽出を行うにあたり，まず問題となるのは，どのような性質の存在証明を行えば部分評価器が抽出されるのかという点である．強正規化性定理から評価器が抽出されることは一般的に知られている．そのため本研究では，正当性の保証された部分評価器の抽出方法を考えるための足がかりとして，まず shift/reset 付き CBV の型付き λ 計算における評価器プログラムの抽出を行った．定理証明には Tait 流の論理述語を用いた．又，型システムの定式化の際に問題となるのが，如何にして変数名の衝突（α 同値）を避けるかであるが，ここでは Locally Nameless 手法 [3] を用いて α 同値問題を回避することに成功している．しかしこの方法で得られた評価器プ

ログラムは複雑で，さらには実行する際に term のみならず term の（非常に複雑な）型推論規則を引数として渡す必要があり，ユーザの使用に耐え得るプログラムとはなっていなかった．引数から型推論規則を排除したくとも，プログラムがあまりに複雑なため，プログラムの意味を変えずに削除出来るか否かの判別が出来ない．故にこの方法を単純に拡張しただけでは，実用的な部分評価器は得られないであろうことが分かった．

ではどのような証明を行えば実用に耐え得る簡潔な部分評価器が得られるのかという我々の模索に光明を投げかけたのが Ilik による先行研究 [18, 19, 20] である．Ilik [18, 19, 20] は Coquand [10] の研究をもとに，Kripke モデルの推論規則に対する完全性の証明から部分評価器の一種である型主導部分評価器（type-directed partial evaluator，以降は TDPE と略す）を抽出する方法を示し，Coq で定式化を行っている．TDPE は他の一般的な部分評価器と異なり，term の中身を全く見ずに計算を行う．そのため，実行が非常に高速であるという特徴を持つ．Ilik [20] は shift/reset 付きの体系における TDPE の抽出も行っているが，そ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，継続の返す型が常に固定されてしまっているのに加え，reset の持ち得る型が atomic type のみに限定されている．

一方，Tsushima ら [26] は限定継続命令 shift/reset [12] 付きの λ 計算体系における TDPE を提案しているが，その正当性に関しては議論しておらず，直感的な説明と実装を示しているのみであった．

そこで本研究では Ilik と Tsushima らの研究を融合させ，通常の限定継続を扱える形で Ilik の証明の再構築を行った．まず，Tsushima らの shift/reset 用の TDPE を継続渡し形式（continuation-passing style，CPS）に変換した．（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為，元の直接形式（direct-style，DS）の TDPE をどのように CPS 変換すれば良いのかは自明でない．）次に，適切な Kripke モデルを構築し，そのモデルの推論規則に対する completeness の定理を証明した．completeness 定理は，TDPE の中核である reify/reflect 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っている．証明は全て 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た．又，Kripke モデルを使って型システムの定式化を行っているため，Locally Nameless 手法よりも簡単に α 同値問題を回避することが出来ている．得られた TDPE はとてもすっきりした形をしており，これは，証明が複雑になり，TDPE の抽出を手動で行わなくてはならなかった Ilik の証明とは対照的となっている．

しかしながら，completeness 定理から TDPE を抽出しても，その TDPE の正しさが完全に保証されているわけではない．completeness 定理の定義により，得られた TDPE が入力プログラムと必ず同じ型を持つ正規形を導出することは保証されている．(加えて，Coq で抽出できている以上，その TDPE が無限ループを起こさないことも保証されている.）しかし導出された正規形と元の入力プログラムの意味が等価であることに関しては何の保証もされていないのである．

故に本研究では CBN と（shift/reset なし）CBV の TDPE における Filinski [15] の正当性定理の証明を Coq で定式化し直し，その定式化を拡張する形で，Tsushima ら [26] の shift/reset 付き CBV TDPE の正当性定理を，Coq を用いて示した．本論文における正当性定理とは，TDPE 実行後も入力値の意味（semantics）を変えないという性質を指す．又，この定式化において用いた意味論は Kripke 意味論ではないが，TDPE 抽出に使用した completeness 定理と Kripke 意味論において定義した completeness 定理は非常に近い定義となっている．型システムの定式化については，parametric higher-order abstract syntax（PHOAS）を用いることにより，α 同値問題を回避しつつ，Kripke モデルを用いるよりも簡潔な定式化を実現した．PHOAS は HOAS を拡張した定式化手法であり，Washburn ら [27] によって提案され，Chlipala [8, 9] によって Coq で定式化されている．変数名の生成を全てメタ言語（本研究では Coq）に任せてしまうため，こちらは一切変数名の生成に関して考慮する必要がなく，非常にシンプルに定式化を行うことが出来るのである．しかし証明において PHOAS の higher-order の特性により Coq では証明が困難な性質が存在するため，その性質の証明のみ Coq には載せずに公理として定義した．

本論文の構成は以下の如く．まず 2 章にて，強正規化性証明から shift/reset 付き CBV の λ 計算における評価器プログラムを抽出し，その問題点を明らかにする．次に 3 章にて，一般的に知られる Danvy [13] の CBN TDPE を紹介し，さらに CBV TDPE と Tsushima ら [26] の shift/reset 付き CBV TDPE の説明を行う．そして後者二つの TDPE に対して CPS 変換を行う．本研究で対象とする TDPE は，DS である CBN TDPE，そして CPS 変換された CBV TDPE，shift/reset 付き CBV TDPE である．4 章で Ilik の仕事 [18, 19, 20] を定式化し直し，Kripke モデルの推論規則に対する completeness 定理から，3 章で説明した CBN TDPE，CPS の CBV TDPE，そして 2CPS の shift/reset 付き CBV TDPE を抽出する．又，同章にて Ilik の仕事との差違に関して議論を行う．そして 5 章にて CBN TDPE と CPS の CBV TDPE における Filinski [15] の正当性定理の証明を，Coq で PHOAS を用いて定式化し，さらにその定式化を拡張することによって，2CPS の shift/reset 付き CBV TDPE の正当性定理の証明を，Coq を用いて行う．

第2章　Locally Nameless手法を用いたshift/reset付き評価器の抽出

本章では，定理証明系 Coq を用いて，call-by-value の，let 文及び限定継続命令 shift/reset 文を含んだ多相の型付き λ 計算（以降，$\lambda^{let}_{s/r}$ と表記）における評価器を抽出する．$\lambda^{let}_{s/r}$ を Coq で定式化する際，Aydemir ら [3] による Locally Nameless 手法を用いた λ 計算の定式化方法を参考にし，彼らのライブラリを利用することにより行う．Locally Nameless 手法とは，α-equality の問題を解消する為の変数の名前付けの手法の一つであり，自由変数には $x, y, z, ...$ 等の名前を付けるが，束縛変数には名前を付けずに de Bruijn index を用いて表現するというものである．de Bruijn index は束縛変数を 0 以上の整数で表し，その式における一番内側の binder の変数名を 0 と考え，外にいくに従って $0, 1, 2, ...$ と大きくしていくという表現方法である．例えば，λ 抽象 $\lambda x.\lambda y.\lambda z.y\ x\ w\ z$ は Locally Nameless 手法を用いて表すと $\lambda.\lambda.\lambda.1\ 2\ w\ 0$ となる．

そして，この型システムが強正規化性の性質を満たすことの constructive な証明を Coq にて定式化する．この証明は論理述語を用いて行われており，Asai [1] の証明方法に準ずる部分が多い．この証明の定式化により，この型システムにおける OCaml 言語の（正当性が保証された）評価器プログラムが自動的に抽出される．このプログラムは，入力値に型が付くならば，必ずプログラムの実行が無限ループを起こすことなく停止することが保証されている．

2.1　限定継続命令shift/reset

shift/reset は Danvy ら [12] によって提案された継続を扱う為の命令であり，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出来る．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 [24] 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．又，`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，一行目の `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，二行目の式となる．そして二行目の `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．

2.2　$\lambda_{s/r}^{let}$ と論理述語

本節では，型システム $\lambda_{s/r}^{let}$ と強正規化性の証明に使用する論理述語の定義，及びそれらをどの様にして Coq で定式化したかについて述べる．

2.2.1　構文

構文は以下で定義される．

$$
\begin{array}{rrcl}
\text{value}: & v & := & n \mid x \mid \lambda.e \\
\text{term}: & e & := & v \mid e_1\, e_2 \mid let\ e_1\ e_2 \mid S.e \mid \langle e \rangle \\
\text{pure evaluation context}: & F & := & 0 \mid F\, e \mid v\, F \mid S.F
\end{array}
$$

$$
\begin{array}{rrcl}
\text{monomorphic type}: & \alpha, \beta, \gamma, \delta & := & tb \mid tf \mid (\alpha/\gamma \to \beta/\delta) \\
\text{polymorphic type}: & A & := & \alpha \mid \forall.A
\end{array}
$$

value は変数又は λ 抽象である．本章では変数の表現に Locally Nameless 手法を用いている為，束縛変数と自由変数をそれぞれ別物として定義する．上の定義において，n は束縛変数，x は自由変数を表す．先述したように，前者は 0 以上の整数，後者は $x, y, z, ...$ 等のアルファベットを使って表される．そして term は value であるか，もしくは application $e_1\ e_2$，let 式 $let\ e_1\ e_2$，shift $S.e$，reset $\langle e \rangle$ のいずれかである．pure evaluation context は Kameyama ら [22] による shift と reset に関する公理を使用するために導入した．monomorphic type（単相型）の定義に登場する tb は de Bruijn index で表現される束縛型変数を意味し，n と同じく 0 以上の整数で表される．tf は自由型変数を表す．$(\alpha/\gamma \to \beta/\delta)$ は，引数の型を α，返り値の型を β とし，実行すると現在の継続

の返す型が γ から δ に変化するような関数の型である．この $(\alpha/\gamma \to \beta/\delta)$ は CPS 変換すると $\alpha \to (\beta \to \gamma) \to \delta$ という型に対応する．shift や reset が出てこなければ，γ と δ は常に等しくなるが，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 γ と δ が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 γ と δ）のことをアンサータイプと呼ぶ．$\forall.A$ は多相の型を表しており，$\forall$ で束縛される型変数や $S.e$ は λ 抽象と同じく de Bruijn index を使って表現される．

2.2.2　簡約規則

one-step の簡約規則は“ $\leadsto$ ”を，big-step の簡約規則は“ $\leadsto^*$ ”を用いて表現される．

one-step reduction rules (call-by-value left-to-right):

$$
\begin{array}{rcll}
e_1\, e_2 & \leadsto & e_1'\, e_2 & \text{if } e_1 \leadsto e_1' \\
v\, e_2 & \leadsto & v\, e_2' & \text{if } e_2 \leadsto e_2' \\
(\lambda.e)\, v & \leadsto & e^v & \\
let\ e_1\ e_2 & \leadsto & let\ e_1'\ e_2 & \text{if } e_1 \leadsto e_1' \\
let\ v\ e & \leadsto & e^v & \\
(S.e_1)\, e_2 & \leadsto & S.let\ (\lambda.\langle 1\, (0\, e_2)\rangle)\ e_1 & \\
v\, (S.e_1) & \leadsto & S.let\ (\lambda.\langle 1\, (v\, 0)\rangle)\ e_1 & \\
\langle e\rangle & \leadsto & \langle e'\rangle & \text{if } e \leadsto e' \\
\langle S.e\rangle & \leadsto & \langle let\ (\lambda.0)\ e\rangle & \\
\langle v\rangle & \leadsto & v &
\end{array}
$$

big-step reduction rules:

$$
\begin{array}{rcll}
e & \leadsto^* & e & \\
e_1 & \leadsto^* & e_3 & \text{if } e_1 \leadsto e_2\,,\ e_2 \leadsto^* e_3
\end{array}
$$

ここで e^v は e の中に含まれる束縛変数 0 に v を代入した式を表している．例えば $e = (0\, (\lambda.0\, 1))$ のとき，$e^v = (v\, (\lambda.0\, v))$ となる．

2.2.3　型付け規則

式の型規則は以下の様に表記する．

$$\Gamma;\, \alpha \vdash e : T;\, \beta$$

これは「型環境 Γ において，式 e は T 型を持ち，その実行によってアンサータイプが α から β に変化する」ことを表す．又，式 e を実行しても，常にアンサータイプが変化しない，即ち，どんな型 α においても $\Gamma;\ \alpha \vdash e : T;\ \alpha$ が得られる時は以下の様に書く．また，このようにアンサータイプが変化しないような型を持つ term のことを pure であると言う．

$$\Gamma \vdash_p e : T$$

上記二つの表記を用い，型の推論規則を以下の如く定義する：

$$\frac{ok\ \Gamma \quad (x : M) \in \Gamma}{\Gamma \vdash_p x : M^{U_s}}\ (var)$$

$$\frac{\forall x \notin L. \quad (\Gamma, x : \sigma;\ \alpha \vdash e^x : T;\ \beta)}{\Gamma \vdash_p \lambda.e : (\sigma/\alpha \to T/\beta)}\ (fun)$$

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to T/\beta);\ \delta \quad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1\, e_2 : T;\ \delta}\ (app)$$

$$\frac{\forall U_s. \forall x \notin L.\ (\Gamma \vdash_p e_1 : M^{U_s} \ \ \Gamma, x : M;\ \alpha \vdash e_2^x : T;\ \beta)}{\Gamma;\ \alpha \vdash let\ e_1\ e_2 : T;\ \beta}\ (let)$$

$$\frac{\forall x \notin L.\ (\Gamma, x : \forall.(T/0 \to \alpha/0);\ \sigma \vdash e^x : \sigma;\ \beta)}{\Gamma;\ \alpha \vdash S.e : T;\ \beta}\ (shift)$$

$$\frac{\Gamma;\ \sigma \vdash e : \sigma;\ T}{\Gamma \vdash_p \langle e \rangle : T}\ (reset) \quad \frac{\Gamma \vdash_p e : T}{\Gamma;\ \alpha \vdash e : T;\ \alpha}\ (exp)$$

(var) の規則に登場する「$ok\ \Gamma$」は，型環境 Γ 中に現れる変数名が全て異なることを表している．又，U_s は単相型の集合を表し，M^{U_s} は M 中に現れる束縛型変数を全て U_s 中の要素で置き換え，$\forall.$ を全て取り除いた型を表している．例えば $U_s = x, y, z, ...$，$M = \forall.\forall.(0/1 \to 0/1)$ のとき，$M^{U_s} = (y/x \to y/x)$ の如くとなる．(fun) の規則における $\forall x \notin L$ は，自由変数の集合 L に含まれていないような任意の自由変数 x を表す．この L には，いかなる制約も設けられていないため，自分で自由に L を定めることが出来る．自由変数の制限に関するこの手法は Cofinite Quantification と呼ばれ，詳しくは Aydemir ら [3] の論文にて述べられている．x を他のどこにも現れない変数とするのではなく，この様に L に関してのみの制限に留めることにより，型付け規則に関わる証明を行う際の複雑さを軽減することが可能となる．なお，Aydemir ら [3] は (let) における U_s にも x の場合と同様に Cofinite Quantification を使用しているが，本章では，定理 4 の証明の為に，任意の単相型の集合において成立する様変更を加えている．

Coq において定式化する際，簡約規則の返す型は Coq において命題の型を表す `Prop` 型で定義

したが，型付け規則の返す型は，後述する停止性と論理述語 R の定義に合わせ，Set 型となっている．

ここまでで述べてきた構文・簡約規則・型付け規則の定式化は全て，Inductive コマンドを用い，再帰のあるデータ型として定義を行った．構文の返り値の型は Set（又は Type でも構わない），簡約規則と型付け規則の返り値の型はそれぞれ Prop と Type で定義している．型付け規則に関しては，本来は簡約規則同様，Prop を返り値の型として指定したいところであるが，そうすると後述の証明において型付け規則に関する帰納法（場合分け）を用いることが出来なくなるため，やむなく Type で定義している．

2.2.4　停止性 $N(e)$ と論理述語 $R_T(e)$

先ず停止性を示す述語 N の定義を以下の如く設定する．

- $N(e) \equiv$ 「$e \rightsquigarrow^* v$ となるような value v が存在する」

この式を Coq にて定式化したものが以下となる．

```
Definition N (e: trm) : Set :=
  sig (fun v:trm => value v /\ e -->* v).
```

プログラム抽出を行うため，N の返す型は Coq において関数の型を表す Set 型で定義している．sig (...) は，命題 (value v /\ e -->* v)（「v は value であり，かつ $e \rightsquigarrow^* v$」の意味を表している）を満たす v の集合を表す.（sig (fun v:trm => ...) の代わりに，{v:trm | ...} と記述しても同じ意味である.）関数がこの型を持っているということは，その関数を実行すれば，$e \rightsquigarrow^* v$ を満たす value v を返してくれることを意味する．

次に，強正規化性の証明において要となる論理述語を，単相型に関する帰納法を用いて定義する．この定義は Asai [1] が使用しているものと同様である．

- $R_A(e) \equiv N(e)$
- $R_{(\sigma/\alpha \to \tau/\beta)}(e) \equiv$「$N(e)$ であり，かつ，$R_\sigma(v)$ を満たす任意の value v と，$\lambda.K \models \tau \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e\,v)\rangle)$ が成り立つ」

- $\lambda.K \models \tau \rightharpoonup \alpha \equiv$「$R_\tau(v)$ を満たす任意の v について，$R_\alpha(\langle(\lambda.K)v\rangle)$ が成立し，かつ，ある自由変数の集合 L に含まれる任意の要素 x（$\forall x \notin L$）について，$(\lambda.K)^x = \lambda.K$.」

R の定義において，e が T 型を持っているとする条件がついているケースもあるが，本章では（Asai [1] の証明と同様に,）その条件を設けることはしない．強正規化性を示す際，定理の条件部に型規則の成立を入れているため，そこで e と T の関係性を示すことになり，論理述語にて条件を入れる必要がないからである．又，$\lambda.K \models \tau \rightharpoonup \alpha$ の定義において $(\lambda.K)^x = \lambda.K$ の条件が加えられているのは，定理 4 における証明の際に必要となるためである．

尚，Coq では Inductive コマンドでデータ型を定義する際，自身の再帰が negative に現れる（含意の前件部の前提に相当する部分に,「任意の x に対して自身の定義が成り立つ」という条件が現れてしまう）と，帰納的な定義が出来なくなってしまう．故に今回，論理述語 R を定義するにあたり，通常ならば Inductive コマンドを使って R をデータ型として定義するところだが，R の再帰が negative に現れてしまう（「$R_\sigma(v)$ を満たす任意の value v」の部分）ため，データ型としては定義せずに Type 型を返す関数として以下の如く定義した．

```
Notation "x ** y" := (prod x y) (at level 40) : type_scope.
Reserved Notation "K ||= T ---> a" (at level 70).
Fixpoint R (T : typ) (e : trm) {struct T} : Type :=
  match T with
  | typ_bvar T1 => N e
  | typ_fvar T1 => N e
  | typ_arrow sigma T alpha beta =>
      (N e) **
      (forall v K, value v -> R sigma v ->
       (K ||= T ---> alpha) ->
       R beta (trm_reset (trm_app (trm_abs K) (trm_app e v))))
  end
where "K ||= T ---> alpha" :=
  Value (trm_abs K) ->
  (forall v, value v -> R T v ->
```

```
R alpha (trm_reset (trm_app (trm_abs K) v))).
```

関数として定義された R は，型 T と term e を引数として受け取り，本体では T の場合分けを行って返り値を求めている．`typ_fvar T1` が $T = A$ のケース，`typ_arrow sigma T alpha beta` が $T = (\sigma/T \to \alpha/\beta)$ のケースをそれぞれ指している．`forall v K, ...` の型が `Type` 型である為，この式に合わせるために `Set` と `Prop` のメタな型である `Type` を `R T e` が返す型として指定している．

2.3　強正規化性の証明

本節では $\lambda_{s/r}^{let}$ における強正規化性の証明を説明する．本章においては，Coq で証明を定式化する際，本節で述べる以外の諸々の補題が必要となってくる．しかしそれらは証明の本筋とは関係ない部分であるので，取り上げないこととする．

定理 1. $R_T(e)$ ならば，$N(e)$.

本章で使用している型システムは簡約順序が一意に定まっており，故に以下の補題が成立する．

補題 1. $e_1 \rightsquigarrow e_2$ かつ $e_1 \rightsquigarrow e_3$ ならば，$e_2 = e_3$.

Proof. e_1 に関する帰納法を用いる． □

補題 1 を用いることにより，以下の補題を証明することが出来る．

補題 2. $e \rightsquigarrow^* e'$ かつ $e \rightsquigarrow^* v$ かつ v が *value* であるならば，$e' \rightsquigarrow^* v$.

Proof. $e \rightsquigarrow^* e'$ の big-step の簡約規則に関する帰納法を用いる．

($e = e'$ **のとき**) 明らか.

($e \rightsquigarrow e$”，e” $\rightsquigarrow^* e'$ **のとき**) $e \rightsquigarrow^* v$ に関して場合分けを行う．

> $e = v$ の場合，$v \rightsquigarrow e$” となるので矛盾する．$e \rightsquigarrow e_1$，$e_1 \rightsquigarrow^* v$ の場合，$e \rightsquigarrow e$” かつ $e \rightsquigarrow e_1$ となるので，補題 1 により e” $= e_1$ である．故に $e_1 \rightsquigarrow^* v = e$” $\rightsquigarrow^* v$ と書ける．e” $\rightsquigarrow^* e'$ かつ e” $\rightsquigarrow^* v$ が得られたので，帰納法の仮定により $e' \rightsquigarrow^* v$ が成立.

□

補題 3. $e_1 \rightsquigarrow^* e_2$ かつ $e_2 \rightsquigarrow^* e_3$ ならば，$e_1 \rightsquigarrow^* e_3$.

Proof. $e_1 \rightsquigarrow^* e_2$ に関する場合分けを行う. □

これら諸補題を用いることにより，以下の定理 2・3 を証明することが出来る．これらの証明に関しては型に関する帰納法を用いた．証明の詳細は付録に載せる．

定理 2. $e \rightsquigarrow^* e'$ かつ $R_T(e)$ ならば，$R_T(e')$.

定理 3. $e \rightsquigarrow^* e'$ かつ $R_T(e')$ ならば，$R_T(e)$.

定理 3 の系として，次の系 1 を定義する．

系 1. $e \rightsquigarrow e'$ かつ $R_T(e')$ ならば，$R_T(e)$.

本章では，Kameyama ら [22] の shift と reset に関する二つの公理を用いる：

公理 1 (β_Ω). $R_T(\langle(\lambda.F)M\rangle)$ ならば，$R_T(\langle F^M\rangle)$.

公理 2 (reset-S). $R_T(\langle(\lambda.M)(\lambda.\langle F\rangle)\rangle)$ ならば，$R_T(\langle F^{S.M}\rangle)$.

そして，強正規化性の証明の根幹を成すのが下に示す定理 4 である．定義を書く前に，定理 4 で使用する表記について説明する．$\preceq$ は二つの型の関係を表す記号として用いる．例えば $T' \preceq T$ と書いたときには,「ある単相型の集合 U_s において，$T' = T^{U_s}$ である」ことを表している．例えば，$(\alpha/\tau \to \beta/\tau) \preceq \forall.(\alpha/0 \to \beta/0)$ と書ける．(このときの τ は単相型であるとする.) 又，$e\{n \mapsto e'\}$ は e 中の束縛変数 0 を全て e' で置き換えた式を表し，同様に，$e[x \mapsto e']$ は e 中の自由変数 x を全て e' で置き換えた式を表す．どちらも左側結合である．ここで，$\{n \mapsto e'\}$ を e の binder の中に移動する度に，束縛変数の値が 1 ずつ増えていくことに注意されたい．つまり $(\lambda.e)\{n \mapsto e'\} = \lambda.(e\{n{+}1 \mapsto e'\})$, $(S.e)\{n \mapsto e'\} = S.(e\{n{+}1 \mapsto e'\})$ となる．又，$e^x = e\{0 \mapsto x\}$ である．

定理 4. $\Gamma = x_1 : T_1, x_2 : T_2, ..., x_n : T_n$ とおく．又，以下に登場する $v_1 ..., v_n$ は全て，$T'_1 \preceq T_1, ..., T'_n \preceq T_n$ を満たす任意の型 $T'_1, ..., T'_n$ についてそれぞれ $R_{T'_1}(v_1), ..., R_{T'_n}(v_n)$ を満たす *value* であるとする．

- $\Gamma; \alpha \vdash e\{0 \mapsto y_0\}...\{m \mapsto y_m\} : T; \beta$ が成立するとき，上の条件を満たすような任意の $v_1 ..., v_n$ と，$\lambda.K \models T \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$ が成立する．

- $\Gamma \vdash_p e\{0 \mapsto y_0\}...\{m \mapsto y_m\} : T$ が成立するとき，上の条件を満たすような任意の $v_1 ..., v_n$ について，$R_T(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])$ が成立する．

この定理は項に関する帰納法を用いて証明を行った．詳細は付録に記す．定理 4 を Coq で定式化した際，条件部の「$v_1 ..., v_n$ は全て，$T'_1 \preceq T_1, ..., T'_n \preceq T_n$ を満たす任意の型 $T'_1, ..., T'_n$ について，それぞれ $R_{T'_1}(v_1), ..., R_{T'_n}(v_n)$ を満たす value である」は Inductive コマンドを用い，以下の如く再帰的に定義した．

```
Inductive R_lst : env -> list trm -> Type :=
  | R_empty : R_lst empty nil
  | R_list : forall x M E' v vs',
      ok (E' & x ~ M) ->
      (value v) ->
      (forall Xs, types (sch_arity M) Xs -> R (M ^^ Xs) v) ->
      (R_lst E' vs') ->
      R_lst (E' & x ~ M) (v :: vs').
```

$T_1, ..., T_n$ の情報は環境 Γ に格納されているため，R_lst は環境 E と $v_1, ..., v_n$ のリストを受け取るように定義している．

Coq で証明する際に注意しなくてはならないのは，命題の型である．現在証明したい式と異なる型を持つ条件式に対して場合分けや帰納法や推論を行うことが出来ない，つまり case，induction，inversion，destruct 等の tactic を使うことが許されていない．例えば，現在証明したい式の型が Set であるのに，Prop 型を持つ条件式に対してそれら tactic を使うことは出来ない．故に、手では行える証明手順が Coq で通用しない場合があり，そのことに注意を払いながら証明を行う必要がある．定理 4 のケースにおいては，現在証明したい式が R や N で記述されている場合，その式の型は Type や Set である．このときに，仮に型付け規則が Prop で定義されている場合，inversion 等を用いて型付け規則に対して推論を行うことが出来ない．しかし定理 4 の証明のためには必ず推論を行わなければならないため，先述の如く，型付け規則を Type で定義しなければならないのである．

定理 4 と定理 1 を用いることにより，以下の，停止性を示す強正規化性の定理を証明することが出来る．

定理 5 (強正規化性). $\vdash_p \langle e \rangle : T$ ならば，$N(\langle e \rangle)$.

2.4　プログラムの抽出

前節で述べた証明を Coq で定式化し、定理 5 に対して Extraction コマンドを用いることにより，OCaml 言語の評価器プログラムが抽出される．抽出の際，Coq にて `Set` や `Type` を返り値の型とした記述は，データ型や関数として抜き出される．一方で `Prop` を返り値の型とする記述に関しては抽出されることはない．抽出されたプログラムが満たしている性質（の証明）であると，Coq が自動的に判断するからである．

定理 5 以外で元の定式化からプログラムとして抜き出された主なものは，$\lambda^{let}_{s/r}$ の構文，型付け規則，変数の置換を行う為の関数，N，R，`R_lst`，定理 1・2・3・4，系 1 である．(簡約規則に関しては，`Set` や `Type` で定義せずとも証明を行えるため，`Prop` を返り値の型として記述した．そのため，抽出は行われなかった.）抽出されたそれぞれが，どのようにプログラムとして定義されているのかを，以下でみていく．(ただし，変数の置換を行う関数は，Coq で定義した関数がそのまま抜き出されているだけであるので，説明は省略する.）

2.4.1　構文，型付け規則，N，R，`R_lst` の抽出

まず，構文と型付け規則は，データ型として，抽出されたプログラム中で以下の如く定義される．

```
type trm =
  | Trm_bvar of nat
  | Trm_fvar of Variables.var
  | Trm_abs of trm
  | Trm_let of trm * trm
  | Trm_app of trm * trm
  | Trm_shift of trm
  | Trm_reset of trm
type typ =
  | Typ_bvar of nat
  | Typ_fvar of Variables.var
```

```
  | Typ_arrow of typ * typ * typ * typ
type typing =
  | Typing_app of env0 * typ * typ * typ * typ * typ *
    typ * trm * trm * typing * typing
  | Typing_exp of env0 * trm * typ * typ * typing_pure
  | Typing_shift of Variables.VarSet.S.t * sch Env.env *
    typ * typ * typ * trm * typ * (Variables.VarSet.S.elt -> __ -> typing)
  | Typing_let of sch * Variables.VarSet.S.t * env0 *
    typ * trm * trm * typ * typ * (typ list -> __ -> typing_pure) *
    (Variables.VarSet.S.elt -> __ -> typing)
and typing_pure =
  | Typing_var of sch Env.env * Variables.var * sch * typ list
  | Typing_fun of Variables.VarSet.S.t * sch Env.env *
    typ * typ * trm * typ * typ * (Variables.VarSet.S.elt -> __ -> typing)
  | Typing_reset of env0 * typ * trm * typ * typing
```

構文は Coq 上での定義がそのまま抜き出された形になっている．typing の記述は，項の証明木を全て記述することと等しくなっている．そして論理述語 N と R の定義は以下となっている．

```
type n = trm
type r = __
```

n は最後まで評価し終えた項の型として使われている．r の定義における__は Obj.t 型を表している．この定義では r の中身は記されていないが，r 型が現れている箇所をみると，

- n 型の項，或は，
- n 型の項と，この項を用いた式（R の定義における $\langle(\lambda.K)(e\ v)\rangle$ に相当）を実行した式を返すような関数のペア

を表す型として使われている．

又，定理 4 の定式化に使用した R_lst は元の定義に即したデータ型として抽出されている．

```
type r_lst =
  | R_empty
  | R_list of Variables.var * sch * sch Env.env * trm *
    trm list * (typ list -> __ -> r) * r_lst
```

2.4.2　定理の抽出

まず定理 3 を具体例として，定理とその証明からどの様な関数が抽出されるのかを見る．初めに定理 3 の Coq での定義を記す．

```
Lemma th3 : forall T e e', e -->* e' ->  R T e' -> R T e.
```

ここで“ -->* ”は“ $\rightsquigarrow^*$ ”を表している．そして以下が OCaml 言語で抽出された関数である．

```
(** val th3 : typ -> trm -> trm -> r -> r **)
let rec th3 t0 e e' h = match t0 with
| Typ_bvar n0 -> Obj.magic (Obj.magic h)
| Typ_fvar v -> Obj.magic (Obj.magic h)
| Typ_arrow (t1, t2, t3, t4) ->
    let Pair (n0, r0) = Obj.magic h in
    Obj.magic (Pair (n0, (fun v k _ x _ x0 -> th3 t4
                          (Trm_reset (Trm_app ((Trm_abs k), (Trm_app (e,v)))))
                          (Trm_reset (Trm_app ((Trm_abs k), (Trm_app (e', v)))))
                          (r0 v k __ x __ x0))))
```

この定理に限らず，抽出された定理の Coq における元の定義の結論部の型はどれも Set 或は Type である．関数 th3 が引数として受け取っている t0，e，e'，h は Coq での定式化における T, e，e'，R T e' つまり前節の定義における T，e，e'，$R_T(e')$ にそれぞれ対応している．このように，命題定義 $(\forall a.b.c....A \to ...B \to C)$ の内，$a, b, c...$ そして $A \to ...B$ において Set 或は Type 型を持つ条件式のみが引数として現れている．

そして関数本体部分は，Coq で行った証明に即した形で自動的に定義されている．例えば証明において，ある Set 又は Type 型を持つ式に関して場合分けを使えば（induction や case コマンド

を使えば)，抽出された関数ではその式に対して場合分けを行うよう定義されるし，証明にて帰納法の仮定を用いれば，抽出された関数ではその部分は関数の再帰呼び出しになる．上の定義と前節の証明を照らし合わせてみれば，確かにそのようになっていることが分かる．又，この関数本体では，型を任意に変換してくれる Obj.magic 関数が数カ所で使われている．th3 関数に限らず，抽出されたプログラム全体にわたって、至る所で型の不一致を解消するために Obj.magic が使われている．Obj.magic の出現に関する同様の記述は Berger ら [5] の論文にも見られる．

さて，この関数 th3 は具体的には何を行う関数であろうか．上の定義をみると，第二引数と第三引数は再帰の度に変化はしているものの実際には使われていない．そこでこの二つの引数を削除してみる．

```
let rec th3 t0 h = match t0 with
| Typ_bvar n0 -> Obj.magic (Obj.magic h)
| Typ_fvar v -> Obj.magic (Obj.magic h)
| Typ_arrow (t1, t2, t3, t4) ->
    let Pair (n0, r0) = Obj.magic h in
    Obj.magic (Pair (n0, (fun v k _ x _ x0 -> th3 t4 (r0 v k __ x __ x0))))
```

そして実は，この関数はどの様な値を受け取っても，第二引数 h を答えとして返す．

定理 6. $\forall t0.\forall h.\ th3\ t0\ h = h.$

Proof. t0 に関する帰納法を用いる．

t0 が Typ_bvar n0 あるいは Typ_fvar v のときは明らか．t0 が Typ_arrow (t1, t2) のときは帰納法の仮定により、再帰呼び出しの th3 t4 (r0 v k __ x __ x0) が (r0 v k __ x __ x0) に等しいと仮定すると，最後の 3 行は

```
let Pair (n0, r0) = Obj.magic h in
  Obj.magic (Pair (n0, (fun v k _ x _ x0 -> r0 v k __ x __ x0)))
```

となる．これは,

```
let Pair (n0, r0) = Obj.magic h in Obj.magic (Pair (n0, r0))
```

と同じで，これは h0 と等しい．故に成立．　□

よって th3 は let th3 t0 e e' h = Obj.magic (Obj.magic h) と変更出来ることが分かった．しかし何故抽出した関数がこのような挙動になるのだろうか．そもそも定理 3 の定義の意味は「e を実行した式 e' が $R_T(e')$ を満たすならば，e も $R_T(e)$ を満たす」であった．Coq で定式化する際，勿論この意味に即して定義は行われている．又 2.2 節での R の定義をみれば，R T e を満たすということは，引数 T，e を渡した関数 R の実行結果つまり N e もしくは N e と (forall t, ...) のペアが得られることを意味していることが分かる．そしてこの (forall t, ...) の中味も計算すれば結局は（term t などは残ってはいるが）N の集まりである．そして e -->* e'（$e \rightsquigarrow^* e'$）であるから，補題 1 により，R T e' と R T e は計算すれば必ず同じ値となることが分かる．故に上で示したように，抽出されたプログラムにおいては，引数として受け取った h（つまり Coq での定義における R T e'）がそのまま答えになるのである．

次にその他の諸定理に対応し抽出された関数をみていく．まずは定理 1 に対応して抽出された関数を以下に示す．

```
(** val th1 : typ -> trm -> r -> n **)
let rec th1 t0 e h = match t0 with
| Typ_bvar n0 -> Obj.magic h
| Typ_fvar v -> Obj.magic h
| Typ_arrow (t1, t2, t3, t4) -> let Pair (n0, r0) = Obj.magic h in n0
```

これは，h に格納されている e の実行結果を返す関数となっている．又，引数 e は実行に使われてない．次に定理 2 に対応して抽出された関数を示す．

```
(** val th2 : typ -> trm -> trm -> r -> r **)
let rec th2 t0 e e' h = match t0 with
| Typ_bvar n0 -> Obj.magic (Obj.magic h)
| Typ_fvar v -> Obj.magic (Obj.magic h)
| Typ_arrow (t1, t2, t3, t4) ->
    Obj.magic (Pair ((let Pair (n0, r0) = Obj.magic h in n0),
                     (fun v k _ x0 _ x1 -> th2 t4
                      (Trm_reset (Trm_app ((Trm_abs k), (Trm_app (e,v)))))
                      (Trm_reset (Trm_app ((Trm_abs k), (Trm_app (e', v))))))
```

```
                    (let Pair (n0, r0) = Obj.magic h in r0 v k __ x0 __ x1))))
```

この関数も th3 と同様にして，受け取る値に関わらず，Obj.magic (Obj.magic h) を答えとして返すことが分かる．

系 1 を抽出した関数は以下であり，th3 と全く等しい関数となっている．

```
(** val co1 : typ -> trm -> trm -> r -> r **)
let co1 t0 e e' x = th3 t0 e e' x
```

定理 4 に関しては，長いので一部分のみ載せる．

```
let rec main = function
| Trm_bvar n0 ->
  Pair ((fun e t1 a b nl vl k vs h _ _ x x0 h2 _ ->
         let ht =
         match trm_open_rec_rec nl vl (Trm_bvar n0) with
          | Trm_fvar v -> (match h with
                 | Typing_exp (e0, e1, t2, a0, h4) -> h4
                 | _ -> assert false (* absurd case *)
          | Trm_abs t2 -> (match h with
                 | Typing_exp (e0, e1, t3, a0, h4) -> h4
                 | _ -> assert false (* absurd case *))
          | _ -> assert false (* absurd case *)
         in
         x (all_substs e vs (trm_open_rec_rec nl vl (Trm_bvar n0))) __
         (match ht with
           | Typing_var (e0, x1, m, us) ->
               lemma0_3_2_1 vs e
               (lemma0_3_23 e t1 nl vl n0 ht h2) m us x0
           | _ -> assert false (* absurd case *))),
        (fun e t1 nl vl vs h _ x h1 _ ->
```

```
          match h with
           | Typing_var (e0, x0, m, us) ->
             lemma0_3_2_1 vs e
             (lemma0_3_23 e t1 nl vl n0 h h1) m us x
           | _ -> assert false (* absurd case *)))
| Trm_fvar v ->
  Pair ((fun e t1 a b nl vl k vs h _ _ x x0 h2 _ ->
          match h with
           | Typing_exp (e0, e1, t2, a0, h4) ->
             x (all_substs e vs (Trm_fvar v)) __
             (match h4 with
               | Typing_var (e2, x1, m, us) -> lemma0_3_2_1 vs e v m us x0
               | _ -> assert false (* absurd case *))
           | _ -> assert false (* absurd case *)),
         (fun e t1 nl vl vs h _ x h1 _ ->
          match h with
           | Typing_var (e0, x0, m, us) -> lemma0_3_2_1 vs e v m us x
           | _ -> assert false (* absurd case *)))
| Trm_abs t1 -> ...
| Trm_let (t1, t2) -> ...
| Trm_app (t1, t2) -> ...
| Trm_shift t1 -> ...
| Trm_reset t1 -> ...
```

上の関数は項を引数として受け取り，この項が$\vdash$で型付けされる場合と$\vdash_p$で型付けされる場合のそれぞれに対応した関数のペアを答えとして返すよう定義されている．上の定義から分かるように，引数として受け取った項について場合分けを行い，さらにその中で引数 h として受け取った typing 或は typing_pure に関しても場合分けを行っている．そして場合分けされたそれらが持つデータを実行に引き渡している．これは Trm_bvar n0 や Trm_fvar x 以外のケースでも同様である．

定理 5 の抽出により最終的に得られた評価器は以下である.

```
(** val normalization : trm -> typ -> typing_pure -> n **)
let normalization_trm e t0 h =
  th1 t0 (Trm_reset e) (let Pair (r0, r1) = main (Trm_reset e) in
                        r1 Env.empty t0 Nil Nil Nil h __ R_empty
                        Vars_empty __)
```

この評価器をユーザが使用する場合，評価したい項とその型のみならず，その項の型に関する証明木（typing_pure）を引数（h）として渡さなくてはならない．これは実際的ではないので引数から外したいところであるが，そのためには，関数本体中の r1 の実行において h が使用されないことを示さなくてはならない．つまり関数 main において h が実質上は使用されていないことを示す必要があるが，上述の様に h が持つデータを実行に引き渡している上，プログラム自体が非常に複雑な形をしている為，示すのは簡単ではなく，h を引数から外せるのか否かは未だ結論が出せていない.

2.5　まとめ

本章では，論理述語を用いて強正規化性を証明することにより，shift/reset 及び let 文を含んだ多相の型システムにおける評価器の抽出を行い，さらに抽出したプログラムの挙動の解釈を試みた.

本章の定式化においては型付け規則 (let) に関して問題点があり，単相型の集合 U_s に関して何の制限も加えていないため，本章の定義では，型が付く let 文の存在が非常に制限されてしまっていると考えられる．しかし現在のままでは，U_s に $U_s \not\subset L$ といった制限を設けてしまうと，定理 4 を証明出来なくなってしまう.

又，本章では正当性の保証された（つまり停止性と正当性定理が成立する）評価器を得られてはいるものの，main 関数と normalization 関数は引数として型付け規則を必要とするため，ユーザが使用するにはあまりに複雑である．プログラムの意味を変えずに，手動で main 関数と normalization の引数から型付け規則を外せるのであれば,（手動で変更を加えたプログラムと元のプログラムの等価性を示すことにより,）ユーザの使用に耐えうる評価器となり得るが，プログラムが複雑なため，その可否の特定は出来ていない.

第3章　型主導部分評価器（Type-directed partial evaluator, TDPE）

型主導部分評価器（type-directed partial evaluator，TDPE）は 96 年に初めて Danvy [13] によって提案された部分評価器であり，コンパイルされたコードとそのコードの型が渡されたとき，元のプログラムの標準形（normal form）となっているソースプログラムを答えとして返す．ソースプログラムそのものを評価する一般的な部分評価器 [21] とは異なり，TDPE はプログラムの中身を全く調べない為，実行が非常に高速であるという利点がある．TDPE が調べるのは入力として受け取った（プログラムの）型のみである．この TDPE の効率性はコンパイラの高速な最適化器として利用するのに魅力的であり，Lindley [23] は SML.NET コンパイラの最適化に TDPE を使用している．

本研究では，call-by-name（CBN）, call-by-value（CBV）, CBV に shift/reset を加えたそれぞれの体系における三つの TDPE を扱っている．以下，それぞれの TDPE に関して説明する．

3.1　Danvy の call-by-name の TDPE

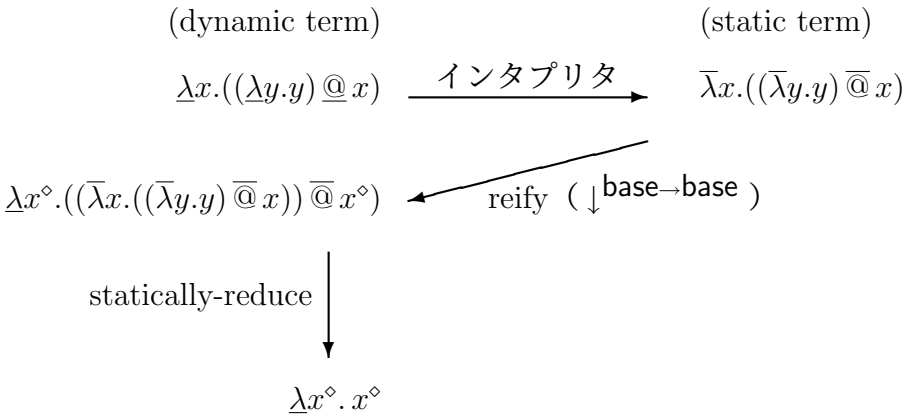
まず一番基本形となる，Danvy[13] により提案された CBN の TDPE について説明する．定義を図 3.1 に示す．この TDPE は直接形式の関数定義となっている．型は，型変数 base か関数型 $t_1 \to t_2$ である．TDPE を行う際には，入力の term を部分評価時に実行してしまう式（$\in$ Static）と実行しない式（$\in$ Dynamic）に分解する．前者は static な式，後者は dynamic な式と呼ばれ，本章ではそれぞれをオーバーライン，アンダーラインを付けて表すものとする．また，両者が混ざった式（$\in$ TLT）は 2 レベルの式（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，インタプリタによっ

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

図 3.1: Danvy の call-by-name の TDPE

て，完全に static な式（つまりオーバーラインのついた式）に変換される．例えば $\underline{\lambda} x.((\underline{\lambda} y.y) \mathbin{\underline{@}} x)$ 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 な部分を全て実行（statically-reduce）すると TDPE 結果が得られる．$\underline{\lambda} x.((\underline{\lambda} y.y) \mathbin{\underline{@}} x)$ に対する TDPE の実行を図で表すと以下のようになる．



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わかる．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

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

図 3.2: call-by-value の TDPE

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．

3.2 Call-by-value の TDPE

次に CBV における TDPE の定義を示す．Tsushima らが先行研究 [26] にて記している CBV の λ 計算における TDPE の定義は，let-insertion を用いて行われており，その let-insertion に shift/reset を用いた定義が図 3.2 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，CBN では $\overline{\lambda} v.\ \uparrow_B (e \,\underline{@}\, \downarrow^A v)$ と定義されているが，CBV 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，即ち 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，$\underline{\lambda} x^{\diamond}.\underline{\mathrm{let}}\ g^{\diamond} = \ldots\ \underline{\mathrm{in}}\ \ldots$ といった形で出力される．又，Danvy [14] 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 Kameyama ら [22] の諸公理を使って証明することが可能.）

この TDPE の定義には，static な shift/reset が使われている．Coq は shift/reset を提供していな

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

図 3.3: call-by-value の CPS TDPE

い為，図 3.2 の TDPE 定義を直接 Coq で定式化することは出来ない.（さらに，static な shift/reset が入ってくると Kripke モデルその他論理関係の構築が複雑になり扱いにくくなる.）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 変換すれば良いのかが明らかではなかったが，Ilik の先行研究 [20] を詳細に検討した結果，static な部分のみを CPS 変換すれば良いことが分かった.）図 3.2 の static な term を CPS 変換したものが，図 3.3 で定義した関数である．この定義は，先の CBV 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.

3.3　Shift/reset 付き call-by-value の TDPE

前節までの定義を踏まえた上で，本節では CBV の λ 計算に shift/reset を加えた体系における TDPE はどの様に定義されるかを説明する．まずは，Tsushima らの CBV の shift/reset 付き λ 計算における TDPE の定義を図 3.4 に示す．前節までとは違い，shift/reset が入力のプログラムに入ってくると，関数の型は 2 章と同様に四つ組となり，$t_1/t_3 \to t_2/t_4$ という形になる [11].

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

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

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

図 3.4: shift/reset 付き call-by-value の TDPE

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 [12] 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 3.4 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．この図ではさらに 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 では提供されていない．そこで，この定義を二回，CPS 変換することでこれら命令を除去する．最初の CPS 変換では，shift/reset が除去され，$\mathrm{shift}_2/\mathrm{reset}_2$ 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．そのようにして二回 CPS 変換をかけて，static な shift/reset をはずしたのが図 3.5 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．

又，どちらの定義においても，reify の $A/\alpha \to B/\beta$ のケースにおける変数 $g^\diamond$ に束縛されてい

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

図 3.5: shift/reset 付き call-by-value の 2CPS TDPE

る式 $\underline{\langle} k_1^\diamond \,\underline{@}\, \downarrow^B v_1 \underline{\rangle}$ は,（dynamic な）reset で囲っても囲まなくとも，式の意味は変わらない．この式の k_1 には capture した継続が入るが，そのさらに外側が reset で囲まれているためである．しかし 5 章にて TDPE の正当性定理において TDPE 実行前後の式が semantic に等価であることを示そうとすると，$\underline{\langle} k_1^\diamond \,\underline{@}\, \downarrow^B v_1 \underline{\rangle}$ の様に reset で囲んでいないと（少なくとも現段階では）証明が出来ない．

以上の本章で示した図 3.1，3.3，3.5 の TDPE 定義をモデルとして，次章以降では定式化を行っていく．

第4章　Kripke意味論を用いたTDPEの抽出

2 章で抽出した shift/reset 付き CBV の評価器は非常に複雑であり，実用にはほど遠いものであった．先行研究にて Ilik [18, 19] は Kripke モデルの推論規則に対する完全性の証明を Coq で定式化することにより，TDPE を抽出している．Ilik の定式化は Kripke モデルを使っているおかげで Locally Nameless 手法を用いるよりも簡単かつきれいな形で α 同値問題を避けることが出来ている．又，term の定義に型情報を加えているため，型推論規則を必要としない.（言い換えると，term の定義自体が型推論規則になっている.）この為，評価器に型推論規則を渡す必要がなく，簡潔な TDPE プログラムの抽出に成功している．さらに Ilik [20] は同様の手法を用いて shift/reset 付き λ 計算体系の TDPE も扱っているが，そ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あり，一般的に使われている限定継続とは異なるものとなっている．具体的には，継続の返す型が常に固定されてしまっており，さらには reset の持ち得る型が atomic type のみに限定されている.

一方，Tsushima ら [26] 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.

本章では，Tsushima らの仕事と Ilik の仕事を結びつけ，通常の限定継続を扱える形で Ilik の証明を再構築する．大まかには以下の手順となる.

- まず，Tsushima らの shift/reset 用の TDPE を継続渡し形式（CPS）に変換する.（shift/reset を扱うために CPS 変換を行うのは常套手段である．しかし TDPE は static/dynamic な term が混ざった式で定義されている為，元の直接形式（DS）の TDPE をどのように CPS 変換するかを判断するのは簡単でない.）

- 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 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．

- 証明を全て Coq で定式化することにより，Tsushima らの TDPE を CPS 変換した関数に直接対応している関数プログラムが得られる．

本章で得られる TDPE は簡潔な形をしており，これは，証明が複雑になり，TDPE の抽出を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．

以降，本章で定義する CBN の（shift/reset なし）λ 計算を λ_{cbn}，CBV の（shift/reset なし）λ 計算を λ_{cbv}，そして CBV の shift/reset 付き λ 計算を $\lambda_{cbv}^{S/R}$ と表記する．

4.1　Kripke モデルと TDPE 抽出の概要

本章では，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．ここでは本章で必要な Kripke モデルを導入する．

定義 1. *Kripke* モデルは以下のふたつからなる．

- $(K, \leq)$ 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，本章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．例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 $w' \models B$.

本章では，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 λ 計算における自由変数の型（＝論理式）の列とする．（本章では，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．その上で，$w \models X$ 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，本章で対象とする λ 計算においても自明に成り立つ．従って，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わかる．

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ことにより，元の入力の部分評価結果を導出する．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．本章では，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．

4.2　λ_{cbn} における TDPE の抽出

本節では CBN の（shift/reset を含まない）λ 計算 λ_{cbn} において，Kripke モデルを用いてどのように TDPE を抽出したかについて説明する．まずは λ_{cbn} を定義する．

4.2.1　λ_{cbn} の定義

本章における Coq での定式化に合わせて，Ilik [20] の定式化を参考にして λ_{cbn} を定義する．(λ_{cbn} の term が，図 3.1 における dynamic な式に相当している．) λ_{cbn} の型システムは図 4.1 の如くに定義される．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 3.1 の様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．この様に定義すると，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．但し，型情報が implicit に推論出来る場合（もしくは型情報に関する議論が不必要な場合）には，本章内においてコロンから右側の式（$: w \vdash_{\mathsf{t}} A$）は省

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

$$
\text{term} :
$$

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

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

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

図 4.1: λ_{cbn} と λ_{cbv} の型システム

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，もしくは `Arguments` コマンドで implicit な式を具体的に指定することにより，自明な環境と型を省略して記述することが出来る.）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する．本章においては，束縛変数は de Bruijn index で表される．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一つ外側の binder に繰り上げるような命令を意味している.（定義をみれば分かる様に，$\mathsf{wkn}(p)$ の規則は weakening を表すものとなっている.）例えば恒等関数 $\underline{\lambda}x.x$ は，$\mathsf{lam}(\mathsf{hyp})$，$\underline{\lambda}x.\underline{\lambda}y.y \mathbin{\underline{@}} x$ 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 hyp 或いは $\mathsf{wkn}(\_)$ 以外の式の外に付くような，例えば $\mathsf{wkn}(\mathsf{app}(\mathsf{hyp}, \mathsf{hyp}))$ のような式を許している．直感的には，application や λ 抽象の外側に $\mathsf{wkn}(\_)$ が付いている様な 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：

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

但し，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．

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

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

一般の de Bruijn index term への変換関数は簡単に定義することが出来る．その変換プログラムは付録に記す.

Coq で term を定式化する際は，[20] を参考に，以下のように定義した．これは図 4.1 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，例えば恒等関数は（world w において (base $\to$ base) 型を持つとすると），
`(tm_Lam (tm_Hyp w base))`
と記述できる.（`Arguments` コマンドや `Set Implicit Arguments` を使用しない場合は，全ての情報を記述しないといけない為,
`(tm_Lam w base base (tm_Hyp w base))`
と記述する.）

4.2.2　λ_{cbn} の Kripke モデル

λ_{cbn} の Kripke モデルを定義するために，まず normal form と neutral term の定義を行う．normal form は正規形であり，これ以上簡約出来ない 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 4.2 に載せる．term の場合と同様，normal form と neutral term も構文木の形で定義される.

本章で使用する 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.

normal form ：

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

neutral term ：

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

図 4.2: λ_{cbn} の normal form と neutral term

$$\begin{aligned} w \models \mathsf{base} &\iff w \vdash_{\mathsf{ne}} \mathsf{base} \\ w \models A \to B &\iff \forall w', w \leq w' \ \Rightarrow\ w' \models A \ \Rightarrow\ w' \models B \end{aligned}$$

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

図 4.3: λ_{cbn} で使用する論理述語

$$\begin{aligned} &(i) \quad w \leq w \\ &(ii) \quad w \leq w' \ \Rightarrow\ w \leq (A, w') \end{aligned}$$

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 4 (transitivity, $\leq$)**.**

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

図 4.3 に，本節で使用する λ_{cbn} 用の論理述語 $\models$ と $\models_s$ を記載する．前者は単純型付き λ 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 [25] を Kripke モデル用に拡張したものであり，$A \to B$ のケースは 4.1 節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 の集合をとってあげれば良い．ただし，型が base の場合は関数型にはなり得ないので，normal form の中でも neutral term のみを考慮すれば良い．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 5 (monotonicity, λ_{cbn})**.**

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

この補題は $w \leq w'$ 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，$\leq$ 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 $\leq$ 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 又は Type 型である場合は，帰納法を用いる式も Set 又は Type 型でないといけない」という制約があるため，$\leq$ の定義を Set 型にしないと $\leq$ に関する帰納法を使えない．$\leq$ に関する帰納法を使わずに上の補題を証明できれば，$\leq$ 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．

4.2.3　λ_{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 7 (Soundness, λ_{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 Γ 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，「Γ のもとで A が成り立つなら，Kripke モデル上でも Γ のもとで A が成り立つ（言い換えると，Γ のもとで A 型になるような term が存在するなら，Γ 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出来ていない．又，Coquand [10] が定理

7 と同様の性質を Kripke モデルの推論規則に対する soundness と呼んでいる為，本章もそれに倣う形をとっている．

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（λ_{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上で，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 4.3）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 $\mathsf{app}(t_1, t_2)$ の場合を見るとわかる通り，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 β 簡約が許されているので，これで CBN 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．

4.2.4　λ_{cbn} の completeness の証明

λ_{cbn} の completeness の定理は，本章では以下の如く定義する．

定理 8 (Completeness, λ_{cbn})**.**

$$\begin{array}{lll} \forall A, & (i) & \forall w, w \models A \;\Rightarrow\; w \vdash_{\mathsf{nf}} A \\ & (ii) & \forall w, w \vdash_{\mathsf{ne}} A \;\Rightarrow\; w \models A \end{array}$$

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わかる．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，本章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，上記の様に限定して $w \vdash_{\mathsf{nf}} A$ を結論部としている．又，(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 $w \vdash_{\mathsf{ne}} \mathsf{base}$ となっている為，neutral

term よりも制約の緩い項を前件部にて許してしまうと $A = \mathsf{base}$ のケースの証明が上手くいかなくなってしまう為である．

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 3.1 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 のひとつなので $w \vdash_{\mathsf{nf}} \mathsf{base}\ [v]$ 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 $w \vdash_{\mathsf{nf}} A \to B\ [\underline{\lambda} x^\diamond.\ \downarrow^B (v@ \uparrow_A x^\diamond)]$ 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．このふたつに対して $\mathsf{app}(\_, \_)$ 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．

```
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 4.4: λ_{cbn} の抽出された TDPE (reify/reflect)

□

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 の定義と一対一の関係にあり，Coq の証明もそれに 沿って行った．その結果，得られた証明は図 4.4 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 の定義と一対一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わかる．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` が使われるかも知れない．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 $w \leq w'$ の証明だったとき，$w \leq (A, w')$ の証明を表す.）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

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

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

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

図 4.5: λ_{cbv} の normal form と neutral term

ている．`wkn_ne` は補題 5 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 [28]）直接は表現できないので，`Obj.magic` が現れる格好になる．

4.2.5　λ_{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出来る．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．

4.3　λ_{cbv} における TDPE の抽出

前節を受けて，本節では CBV の（shift/reset を含まない）λ 計算 λ_{cbv} における TDPE の抽出を説明する．TDPE の抽出手法は，前節のそれをきれいな形で拡張したものになっている．

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

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

図 4.6: λ_{cbv} で使用する論理述語

4.3.1　λ_{cbv} の定義と Kripke モデル

λ_{cbv} の term の定義は λ_{cbn} と同じである（図 4.1）．λ_{cbv} にて使用する Kripke モデルは，λ_{cbn} で定義したモデルにて使用した 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．

λ_{cbv} における normal form，neutral term の定義に関しては図 4.5 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，見やすさの為に導入した．この式は CBN ではさらに β 簡約が出来る．しかし CBV では $\mathsf{lam}(p)$ に渡される項が（これ以上簡約出来ないような）application だと簡約が出来ない．図 4.5 の let 式の定義をみれば分かる通り q は neutral term であるので，$\mathsf{app}(q, q')$ はこれ以上簡約出来ず，故に $\mathsf{app}(\mathsf{lam}(p), \mathsf{app}(q, q'))$ はこれ以上簡約出来ない項となっている．その為，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 4.6 である．この定義が λ_{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，λ_{cbn} では $w \models A \ \Rightarrow \ w \models B$ となっていた式が，λ_{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わかる．

λ_{cbv} のときと同様，論理述語が Kripke モデルの forcing の関係を満たす為には，base のケースに

おいて monotone でなくてはならない．λ_{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 6 (monotonicity, λ_{cbv})**.**

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

この性質は,（前節のときと同様に,）$w \leq w'$ 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 を得ることが出来る．以上のセッティングにより，λ_{cbn} のときと同様に，Kripke モデルを定義することが出来た．この諸定義を用いて，soundness と completeness の証明を行う．

4.3.2　λ_{cbv} の soundness の証明

λ_{cbv} での soundness 定理は以下の如く，λ_{cbn} の場合よりも複雑な定義となる．

定理 9 (Soundness, λ_{cbv})**.**

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

定理 9 の定義が意味するところは基本的に定理 7 と同じであるが，結論部「Γ 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,「Γ 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 7 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．

4.3.3　λ_{cbv} の completeness の証明

λ_{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）λ_{cbn} での定義と全く同じである．

```
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 4.7: λ_{cbv} の抽出された TDPE (reify/reflect)

定理 10 (Completeness, λ_{cbv})**.**

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

定理 10 の証明は，図 3.3 の CPS TDPE のプログラムと一対一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 4.7 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 4.7 と図 3.2 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，λ_{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，λ_{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，λ_{cbv} では let-insertion により全ての部分式に名前がつくため，変数を扱えさえすれば十分だからである．

4.3.4　λ_{cbv} 用の reify への入力について

以上で λ_{cbv} 用の TDPE が得られたが，この TDPE が受け取る入力について，ひとつ面白いことが起きている．λ_{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，λ_{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．例えば $\underline{\lambda}x.((\underline{\lambda}y.y)\,\underline{@}\,x) : \mathsf{base} \to \mathsf{base}$ を TDPE にかけた結果が欲しければ，λ_{cbn} 用の TDPE を使うときには $\overline{\lambda}x.((\overline{\lambda}y.y)\,\overline{@}\,x)$ を `reify` に渡せば良かったが，λ_{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知れないが，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．λ_{cbn} 用の TDPE では内部表現として直接形式を使っていたが，λ_{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，λ_{cbv} 用の soundness の定理は，CPS インタプリタになっていた．従って，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．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．

4.4　$\lambda_{cbv}^{S/R}$ における TDPE の抽出

λ_{cbn} と λ_{cbv} の TDPE の抽出方法を受けて，本節では $\lambda_{cbv}^{S/R}$ における TDPE の抽出について説明を行う．基本的には前節と同じアプローチで，さらにもう一度 CPS 変換を行う．

本節では図 3.5 をモデルと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全てが λ_{cbn}，λ_{cbv} のそれと全く同じ流れになっており，λ_{cbv}（さらに遡れば λ_{cbn}）の定式化の拡張となっていることに注意されたい．

4.4.1　$\lambda_{cbv}^{S/R}$ の定義

まず $\lambda_{cbv}^{S/R}$ を図 4.8 にて定義する．これは Danvy と Filinski による shift/reset に対する型システム [11] を多相に拡張した Asai と Kameyama の型システム [2] に基づいており，λ_{cbn} や λ_{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ータイプが a から b へと変化する．

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

term :

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

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

図 4.8: $\lambda_{cbv}^{S/R}$ の型システム

(pure) normal form :

$$\frac{p : w \vdash_{\mathsf{ne}} A}{p : w \vdash_{\mathsf{nf}} A} \qquad \frac{p : B/a \to a/a, A, w \vdash_{\mathsf{nf}} b}{\mathsf{lam}(\mathsf{shift}(p)) : w \vdash_{\mathsf{nf}} A/a \to B/b} \qquad \frac{}{\mathsf{hyp} : A, w \vdash_{\mathsf{ne}} A} \qquad \frac{p : w \vdash_{\mathsf{ne}} A}{\mathsf{wkn}(p) : B, w \vdash_{\mathsf{ne}} A}$$

pure neutral term :

$$\frac{q : w \vdash_{\mathsf{ne}} c'/c \to c/c \quad q' : w \vdash_{\mathsf{nf}} c' \quad p : c, w \vdash_{\mathsf{nf}} A}{\mathsf{let}(\mathsf{reset}(\mathsf{app}(q, q')), p) : w \vdash_{\mathsf{ne}} A} \qquad \frac{p : w; a' \vdash_{\mathsf{nex}} a'; c \quad q : c, w \vdash_{\mathsf{nf}} A}{\mathsf{let}(\mathsf{reset}(p), q) : w \vdash_{\mathsf{ne}} A}$$

impure neutral term :

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

図 4.9: $\lambda_{cbv}^{S/R}$ の normal form と neutral term

4.4.2　$\lambda_{cbv}^{S/R}$ の Kripke モデル

$\lambda_{cbv}^{S/R}$ にて使用する Kripke モデルを定める．$\lambda_{cbv}^{S/R}$ にて定義する Kripke モデルは，λ_{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 4.9 の如く定義する．$\lambda_{cbv}^{S/R}$ では，neutral term が pure である（つまりアンサータイプが変化しないような型を持つ neutral term）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 は登場しない為，normal form は pure の場合のみあれば十分である.）$\mathsf{lam}(\mathsf{shift}(\_))$ は λ 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 λ 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．

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 4.10 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（アンサー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 4.10 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 $w \vdash_{\mathsf{ne}} \mathsf{base}$ である為，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．λ_{cbn}，λ_{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 7 (monotonicity, $\lambda_{cbv}^{S/R}$)**.**

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

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．故に図 4.10 で定義した $\models$ は Kripke モデル上の forcing の関係となっている．

4.4.3　$\lambda_{cbv}^{S/R}$ の soundness の証明

Soundness の定義は以下の如く，λ_{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 11 (Soundness, $\lambda_{cbv}^{S/R}$)**.**

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

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，本節の 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．

$$
\begin{array}{lcl}
w \models \mathsf{base} & \iff & w \vdash_{\mathsf{ne}} \mathsf{base} \\
w \models A/a \to B/b & \iff & \forall w_1, w \le w_1 \ \Rightarrow\ w_1 \models A \ \Rightarrow \\
 & & (\forall w_2, w_1 \le w_2 \ \Rightarrow\ w_2 \models B \ \Rightarrow \\
 & & \ \forall T, (\forall w_3, w_2 \le w_3 \ \Rightarrow\ w_3 \models a \ \Rightarrow\ w_3 \vdash_{\mathsf{nf}} T) \ \Rightarrow \\
 & & \qquad w_2 \vdash_{\mathsf{nf}} T) \ \Rightarrow \\
 & & \forall T', (\forall w_2, w_1 \le w_2 \ \Rightarrow\ w_2 \models b \ \Rightarrow\ w_2 \vdash_{\mathsf{nf}} T') \ \Rightarrow \\
 & & \qquad w_1 \vdash_{\mathsf{nf}} T'
\end{array}
$$

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

図 4.10: $\lambda_{cbv}^{S/R}$ で使用する論理述語

4.4.4　$\lambda_{cbv}^{S/R}$ の completeness の証明

次に，TDPE を抽出する為の completeness の証明を行う．定理の定義自体は，本質的に λ_{cbn}，λ_{cbv} の定義と同じである．

定理 12 (Completeness, $\lambda_{cbv}^{S/R}$)**.**

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

定理 12 の証明は，本節のはじめに図 3.5 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 4.11 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．又，`ne_LetApp` は pure な neutral term である $\mathsf{let}(\mathsf{reset}(\mathsf{app}(\_, \_)), \_)$ を，`nex_LetApp` は impure な neutral term である $\mathsf{let}(\mathsf{reset}(\mathsf{app}(\_, \_)), \_)$ をそれぞれ表している．`arrow` は $(\_/\_ \to \_/\_)$ を表す．図 4.11 の関数定義においても，図 3.5 の CPS の shift/reset 付 TDPE と等しいであろうことが見てとれる．

4.4.5　$\lambda_{cbv}^{S/R}$ 用の reify への入力について

λ_{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．従って，shift/reset を含んだ入力プログラムを TDPE にかけたければ，一度，soundness の証明

```
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 (arrow A2 A3 A3 A3 :: A1 :: w) A4
                  (lew_trans (lew_cons (arrow A2 A3 A3 A3) (lew_cons A1 (lew_refl w)))
                             (lew_refl (arrow A2 A3 A3 A3 :: A1 :: w)))
                  (reflect (ne_Wkn (arrow A2 A3 A3 A3) (ne_Hyp w A1)))
                  (fun w2 _ (H: arrow A2 A3 A3 A3 :: A1 :: w <== w2) v1 k2 =>
                   nf_ne (ne_LetApp (wkn_ne H (ne_Hyp (A1 :: w) (arrow A2 A3 A3 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))
  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))))
  end.
```

図 4.11: $\lambda_{cbv}^{S/R}$ の抽出された TDPE (reify/reflect)

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．

4.5　関連研究

TDPE の定義は非常に単純であるが，その正しさを示すのは簡単でない．Filinski [15] は CBN と CBV それぞれの TDPE の正当性を表示的意味論と Kripke 論理関係を用いて示し，さらにそれを任意の monadic effect を持つ体系を対象とする様拡張している [16]．その論文 [16] では計算順序を保存する為の let-insertion を含めて証明がなされている．又，Fiore [17] は圏論的かつ代数的観点から TDPE を研究しており，同様のアプローチを用いて Balat ら [4] は sum 型を持つ TDPE を定義している．

本章の証明の足がかりを与えたのは Ilik の仕事である [18, 19]．まず初めに Coquand [10] 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 とが対応し得ることが示され，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為した．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 [20].

本章と Ilik の研究 [18, 19, 20] で異なる点は以下の如くである．まず，shift/reset なしの CBV 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．本章で定義した（CBV 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．従って導出される TDPE の構造も，本章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 λ 計算は論理学に基づいて考えられているもので，アンサータイプが固定されている上に，reset の持つ型が atomic type のみに限定されている．一方で，本章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，本章で shift/reset 付 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 では全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なわれている．さらに彼の論文 [20] 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 [22] の公理系において同じとは認められないものが出てきているなど不可解な点がある．本章は，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．

より理論的な TDPE に関する研究としては，Fiore [17] による圏論をベースとした意味論の構築，Balat ら [4] による sum 型をサポートしたものなどがあげられる．また，より実際的な方向性として Lindley [23] は SML.NET の中間言語のコンパイラとして TDPE を使っている．

4.6　まとめ

本章では，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 オリジナルの CBN TDPE だけでなく，

let-insertion を行う CBV 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，本章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．

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．又，本章で行った簡潔な形での TDPE 抽出は，他のシステムにおいて TDPE を構築する際にも役に立つと思われる．実際，5 章においては Kripke モデルを使用していないものの，本章と類似の手順を踏むことで TDPE を得ることが出来る．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わかると，逆に証明から正しい TDPE の形を割り出すことが出来るようになってくると思われる．本章では 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わかる．

第5章　PHOASを用いたTDPEの抽出と正当性の証明

本章ではPHOASを用いてCBN，CBV，Tsushimaらのshift/reset付きCBVそれぞれのTDPEを抽出し，さらに各TDPEが，その実行において入力値の意味を変えない即ち正当性定理を満たすことを証明する．TDPEの抽出までは4章と同じ流れとなっていることに注意したい．又，CBNとCBVのTDPEの正当性定理の証明は，Filinski [15] の証明を参考にしており，特にCBNのTDPEの正当性定理証明に関してはFilinskiの証明と同じものとなっている．以下，本章にて定義するCBN，CBV，shift/reset付きCBVのλ計算をそれぞれΛ_{cbn}，Λ_{cbv}，$\Lambda_{cbv}^{S/R}$と書くこととする．

5.1　Call-by-nameのTDPEの正当性証明

本節ではCBNのTDPEの正当性を示す．まずはΛ_{cbn}を定義する．

5.1.1　Λ_{cbn}の定式化

まず，型とtermを以下の如くに定義する．

$$\begin{array}{rcrcrcl}
\text{type} & : & \mathsf{typ} & \ni & A, B & := & \mathsf{base} \mid \mathsf{arrow}(A, B) \\
\text{pre-term} & : & \mathsf{tm}\ \mathcal{V}\ A & \ni & t & := & \mathsf{var}(x) \mid \mathsf{lam}(\lambda x.t) \mid \mathsf{app}(t_1, t_2) \\
\text{term} & : & \mathsf{TM}(A) & \ni & T & := & \lambda\mathcal{V}.t : \mathsf{tm}\ \mathcal{V}\ A
\end{array}$$

ここではtermをWashburnら [27] の提案したparametric higher-order abstract syntax（PHOAS）を用いて定義している．$\mathcal{V}$は$\mathsf{typ} \to *$型の関数であれば何でも良い．ここで$*$はtypよりもメタな型を表す.（つまり$\mathsf{typ} : *$であり，Coqでは Type に相当する.）typ型を持つのはbaseもしくは関数型を表す$\mathsf{arrow}(_, _)$のみである．$\mathsf{var}(x)$は（束縛）変数を表す．$\mathsf{var}(x)$と$\mathsf{lam}(\lambda x.t)$で使われている$x$と$\lambda x....$は，以下本章においてはstatic termにおける（つまりメタ言語における）変数

と λ 抽象（無名関数）を表す．故に上では便宜上 $\mathsf{var}(x)$ も独立して表記しているが，本研究ではメタ言語 Coq を用いて定式化しているため，実際は λ 抽象を表す $\mathsf{lam}(\lambda x.t)$ の t の内部にしか登場し得ない．つまり Λ_{cbn} では closed term のみしか作り得ない．(Λ_{cbv}，$\Lambda_{cbv}^{S/R}$ でも同様である．）そして変数の型は必ず $\mathcal{V}$ に typ 型の式を渡して得られる値（つまり $\mathcal{V}(A)$）となっている，つまり（HOAS と異なり）変数の型が関数 $\mathcal{V}$ によってパラメータ化されている．Coq では自身が negative に登場するような再帰的定義は行えない為，HOAS は定式化出来ない．しかし PHOAS を用いた上の定義においては，$\mathcal{V}$ が tm に置換され得ないが故に tm の定義において tm 自身が negative に登場することがないため，Coq で定式化することが出来る．

上の表記を用いると，例えば恒等関数は

$$\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{var}(x)),$$

3 章の表記で書く所の $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$ は

$$\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{app}(\mathsf{lam}(\lambda y.\mathsf{var}(y)),\mathsf{var}(x))),$$

S コンビネータは

$$\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{lam}(\lambda y.\mathsf{lam}(\lambda z.\mathsf{app}(\mathsf{app}(\mathsf{var}(x),\mathsf{var}(z)),\mathsf{app}(\mathsf{var}(y),\mathsf{var}(z))))))$$

とそれぞれ書ける．しかし上述の如く，本章では closed term 以外は定義出来ないので，例えば $\lambda\mathcal{V}.\mathsf{app}(\mathsf{var}(x),\mathsf{var}(y))$ といった式は書けない．

又，t は必ず以下の型を持つものと定める．

$$\begin{aligned}
\mathsf{tm}\ \mathcal{V} &: \mathsf{typ} \to * \\
\mathsf{var} &: \forall\{A:\mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{tm}\ \mathcal{V}\ A \\
\mathsf{lam} &: \forall\{A:\mathsf{typ}\}, \forall\{B:\mathsf{typ}\}, (\mathcal{V}(A) \to \mathsf{tm}\ \mathcal{V}\ B) \to \mathsf{tm}\ \mathcal{V}\ \mathsf{arrow}(A,B) \\
\mathsf{app} &: \forall\{A:\mathsf{typ}\}, \forall\{B:\mathsf{typ}\}, \mathsf{tm}\ \mathcal{V}\ \mathsf{arrow}(A,B) \to \mathsf{tm}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ B
\end{aligned}$$

ここで $\{A:\mathsf{typ}\}$ 等 {} で囲われている引数は，他の引数から推論可能な式であるため，t を書く際には記述を省いて構わないものとする．以降，本論文では term や定理の定義にて，他の式から推論可能な引数を記号 {} を用いて表し，{} で囲われた引数は，term や定理を実際に使用する際には記述を省略するものとする．(それ故，上述の t の定義において型の記述を省いていた.)

Coq で term を定義する際は，この型に関する記述そのままの形で定式化が行える：

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

この tm と TM A が受け取っている引数 var が，関数 $\mathcal{V}$ を表している．又，Coq で term を記述する際，以下のように予め implicit arguments（明示せずとも他から推論可能な引数）を宣言しておくことで，var，A，B を省略することが出来る．

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

以上の定義により，例えば上の例で挙げた恒等関数と $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$，そして S コンビネータを以下のように Coq で記述することが出来る．

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


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


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

TDPE が出力するのは $\beta\eta$-long normal form であるため，4 章と同じく normal form と neutral term を定義する必要がある．ここでは PHOAS を使って以下の如くに normal form，neutral term を定義する．

$$
\begin{array}{rrcccl}
 & \mathsf{nf}\ \mathcal{V}\ A & \ni & nf & := & \mathsf{lam}(\lambda x.nf) \mid \mathsf{nf\_ne}(ne) \\
 & \mathsf{ne}\ \mathcal{V}\ A & \ni & ne & := & \mathsf{var}(x) \mid \mathsf{app}(ne, nf) \\
\text{normal form} : & \mathsf{NF}(A) & \ni & Nf & := & \lambda\mathcal{V}.nf : \mathsf{nf}\ \mathcal{V}\ A \\
\text{neutral term} : & \mathsf{NE}(A) & \ni & Ne & := & \lambda\mathcal{V}.ne : \mathsf{ne}\ \mathcal{V}\ A
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{nf}\ \mathcal{V} & : & \mathsf{typ} \to * \\
\mathsf{lam} & : & \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, (\mathcal{V}(A) \to \mathsf{nf}\ \mathcal{V}\ B\ ) \to \mathsf{nf}\ \mathcal{V}\ \mathsf{arrow}(A, B) \\
\mathsf{nf\_ne} & : & \mathsf{ne}\ \mathcal{V}\ \mathsf{base} \to \mathsf{nf}\ \mathcal{V}\ \mathsf{base} \\
\mathsf{ne}\ \mathcal{V} & : & \mathsf{typ} \to * \\
\mathsf{var} & : & \forall\{A : \mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{ne}\ \mathcal{V}\ A \\
\mathsf{app} & : & \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, \mathsf{ne}\ \mathcal{V}\ \mathsf{arrow}(A, B) \to \mathsf{nf}\ \mathcal{V}\ A\ \to \mathsf{ne}\ \mathcal{V}\ B
\end{array}
$$

nf は base 型の ne であるか（識別子 nf_ne(_) で表す），body 部分が nf となっている λ 抽象のどちらかである．nf となり得る ne は必ず base 型を持つ様制限してあるので，上の定義で作られる正規形は $\beta\eta$-long normal form となる．又，ne は変数であるか，変数を nf で呼び出した形の application であり，先頭の変数を λ 抽象に置換しない限り，これ以上簡約出来ない式となっている．この定義の場合，例えば恒等関数は $\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{nf\_ne}(\mathsf{var}(x)))$ と書ける．上の定義そのままに，Coq では以下の如くに normal form，neutral term，そして恒等関数を定義出来る．

```
Inductive nf (var: typ -> Type) : typ -> Type :=
| nf_Lam : forall A B, (var A -> nf var B) -> nf var (arrow A B)
| nf_ne  : ne var base -> nf var base
with ne (var: typ -> Type) : typ -> Type :=
| ne_Var : forall A, var A -> ne var A
| ne_App : forall A B, ne var (arrow A B) -> nf var A -> ne var B.
Arguments nf_Lam [var A B] _.
Arguments nf_ne [var] _.
Arguments ne_Var [var A] _.
Arguments ne_App [var A B] _ _.
Definition NF A := forall var, nf var A.
Definition NE A := forall var, ne var A.
```

```
Example NF_id : NF (arrow base base) := fun var =>
  nf_Lam (fun x => nf_ne (ne_Var x)).
```

以下の tm_of_nf(_), tm_of_ne(_) は nf : nf $\mathcal{V}$ A と ne : ne $\mathcal{V}$ A それぞれを，対応する e : tm $\mathcal{V}$ A へと変換する関数である．

$$
\begin{aligned}
\mathsf{tm\_of\_nf}(\_) &: \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{nf}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ A \\
\mathsf{tm\_of\_nf}(\mathsf{lam}(\lambda x.nf)) &= \mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nf}((\lambda x.nf)x')) \\
\mathsf{tm\_of\_nf}(\mathsf{nf\_ne}(ne)) &= \mathsf{tm\_of\_ne}(ne) \\
\\
\mathsf{tm\_of\_ne}(\_) &: \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{ne}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ A \\
\mathsf{tm\_of\_ne}(\mathsf{var}(x)) &= \mathsf{var}(x) \\
\mathsf{tm\_of\_ne}(\mathsf{app}(ne, nf)) &= \mathsf{app}(\mathsf{tm\_of\_ne}(ne), \mathsf{tm\_of\_nf}(nf))
\end{aligned}
$$

これら関数は Coq では以下の様に定義出来る．

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

5.1.2　Soundness

型に関する semantics は以下の論理関係 V によって定義されるが，この論理関係が 4 章における（λ_{cbn} の）$\models$ に相当している．$\models$ と異なる点は，world を引数として持たず，base 型における semantics b でパラメータ化されているところである．

$$
\begin{aligned}
V\ b\ \mathsf{base} &\iff b \\
V\ b\ \mathsf{arrow}(A, B) &\iff V\ b\ A \Rightarrow V\ b\ B
\end{aligned}
$$

arrow(A, B) のケースにおける式 $(V\ b\ A\ \Rightarrow\ V\ b\ B)$ は，$V\ b\ A$ 型の式を受け取って $V\ b\ B$ 型の式を返す（Coq 上の）static な関数の型となっている．Coq では V を以下の如くに `Fixpoint` を使って定義出来る：

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

そして λ_{cbn} の場合と同様にして，V を用いて soundness 定理を定義することが出来る．

定理 13 (Soundness, Λ_{cbn})**.**

$$\forall b, \forall \{A\}, \mathsf{tm}\ (V\ b)\ A\ \Rightarrow\ V\ b\ A$$

この soundness 定理が意味する所は，どんな b が与えられても，もし A が型付け規則（tm　が相当している）から導出可能であるならば，semantics において A は真であるということである．この定理は（4 章と同様,）tm $(V\ b)\ A$ に関する帰納法を用いることにより簡単に証明することが出来る．

この soundness 定理は A 型の term t の semantics を定義しており，実際，この定理の proof term は以下の DS インタプリタとなっている：

```
Fixpoint soundness b {A} (t: tm (V b) A) : V b A :=
  match t with
| tm_Var _ x => x
| tm_Lam _ _ t1 => fun x => soundness b (t1 x)
| tm_App _ _ t1 t2 => (soundness b t1) (soundness b t2)
  end.
```

A 型の typing derivation t が与えられると，`soundness` 関数は t を Coq のコードにコンパイルした結果を返す．λ_{cbn} で定義した soundness 関数のときと同じく，メタ言語である Coq が一般の β 簡約を許している為，上の `soundness` 関数も又，call-by-name のインタプリタになっているが，

このインタプリタが call-by-name となっているそもそもの原因は，V の $\mathsf{arrow}(A, B)$ のケースにおける定義が Coq の関数型そのものとなっていることにある．

$\mathsf{TM}(A)$ 型の term T のためのインタプリタ soundness2 は，soundness 関数を使って以下の如くに定義出来る．

$$\mathsf{soundness2}\ b\ T := \mathsf{soundness}\ b\ (T\ (V\ b))$$

上の関数は，Coq ではそのまま以下のように定義される．

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

5.1.3　Completeness

semantics V における completeness 定理は以下の如く定める．

定理 14 (Completeness (reify/reflect), Λ_{cbn})**.**

$$\forall \mathcal{V}, \forall A,\quad (i)\quad V\ (\mathsf{ne}\ \mathcal{V}\ \mathsf{base})\ A\ \Rightarrow\ \mathsf{nf}\ \mathcal{V}\ A$$
$$(ii)\quad \mathsf{ne}\ \mathcal{V}\ A\ \Rightarrow\ V\ (\mathsf{ne}\ \mathcal{V}\ \mathsf{base})\ A$$

4 章と比較して，使っている論理関係（semantics）が異なる以外は同様の形をしており，この定理の proof term が reify/reflect そのものとなり得る点も 4 章と同じである．証明は型 A の帰納法を用いて行えるが，以下の様に直接 completeness（reify/reflect）の proof term を定義することも出来る：

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

上の定義をみると，図 3.1 の Danvy の定義と全く同一となっていることが分かる．Danvy の定義においてオーバーラインを付けて表されていた term がメタ言語である Coq のコードとして，アンダーラインで表されていた term はデータ型としてそれぞれ表されている．

図 3.1 の $x^\diamond$ は fresh な変数である．fresh な変数をとってくるという操作を正しく行う為に，4 章の λ_{cbn} における `reify/reflect` 関数では環境 world を常に持ち歩き，環境を実際に使う際には定義時と呼び出し時の間にどれ位のずれがあるかを示す必要があった．しかし上の定義では PHOAS を用いているおかげで，fresh な変数を全く簡単に生成している．`nf_Lam` には変数 `x` で束縛された Coq の（static な）関数 (`fun x => ...`) が渡されているが，この変数 `x` が fresh であることはメタ言語である Coq によって保証されている．

`A` 型の term `T` の正規形を得るには，まず `T` を `soundness2` 関数に（但し，引数 `b` には任意の `var` における `ne var base` を代入する）渡して，`T` の semantics を得る．そしてその結果を `A` と `var` と共に `reify` 関数に渡して実行することによって `T` の正規形が得られる．

上の `reify/reflect` の定義（と定理 14）をみると，V の引数 b には $\mathsf{ne}\ \mathcal{V}\ \mathsf{base}$ を代入しているが，これは V の base case においては b がそのまま返ってくることを考えれば当然である．`reify` は入力された式の型が base だった場合には，その式が既に正規形になっていると判断してそのまま返すので，b が normal form 以外のものが来ては正常に動作しないであろうことは明らかである．(neutral term は normal form の部分集合であるので，b に $\mathsf{nf}\ \mathcal{V}\ \mathsf{base}$ でなく $\mathsf{ne}\ \mathcal{V}\ \mathsf{base}$ を代入しても問題ない.）reification が行われる際，base 型の value は neutral term として実行されるが，Filinski はこれを residualizing interpretation と呼んでいる [15]．又，`reify` とは逆に `soundness` 関数にはどんな b がきても構わない．Filinski [15] は soundness 関数を evaluating interpretation と呼んでいる．

上で定義した term `TM_id'` を使って TDPE の実行例を示す.（既に説明した通り，この term は $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$ を表している．又，この term は $\mathsf{TM}(\mathsf{arrow}(\mathsf{base},\mathsf{base}))$ 型を持つ.）以下にもう一度定義を載せる．

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

この term の normal form を得るには，任意の `var` において，

```
Axiom var: typ -> Type.
```

以下の如くに実行すれば良い：

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

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

λ 抽象の body 部分が簡約され，normal form $\lambda x.\, x$ が得られている．又，`TM_id'` は型が明示的に定義されているので，`reify` に渡す型と `soundness` に渡す b を省いて記述しても，Coq が勝手に型推論を行ってくれるので実行が可能である：

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

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

もう一つ TDPE の実行例を示す．$\underline{\lambda} x.\underline{\lambda} y.(\underline{\lambda} z.z)\,\underline{@}\,(x\,\underline{@}\,y)$ は，CBN の TDPE においては関数の body が簡約されて，$\underline{\lambda} x.\underline{\lambda} y.x\,\underline{@}\,y$ が得られるはずである．評価前の式は Coq で以下のように定義出来る．

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

この式を TDPE に渡して実行すると，実際に $\underline{\lambda} x.\underline{\lambda} y.x\,\underline{@}\,y$ 即ち本節の表記における $\mathsf{lam}(\lambda x.\mathsf{lam}(\lambda y.\mathsf{nf\_ne}(\mathsf{app}(\mathsf{var}(x), \mathsf{var}(y)))))$ が得られる：

```
Eval compute in (reify var _ (soundness2 _ TM_t)).

= nf_Lam (fun x :  var (arrow base base) =>
          nf_Lam (fun x0 :  var base =>
                  nf_ne (ne_App (ne_Var x) (nf_ne (ne_Var x0)))))
:  nf var (arrow (arrow base base) (arrow base base))
```

5.1.4　正当性定理の証明

前節にて reify 関数が completeness 定理の proof term であることを示したが，これは TDPE の正当性（つまり，TDPE の実行前後で semantics が変化しない）を直接示したことにはなっていない．A 型を持つ normal form を導出しても，それが soundness 関数を使って $V\ (\mathsf{ne}\ \mathcal{V}\ \mathsf{base})\ A$ に変換する前の dynamic term を簡約して得られる normal form となっているかは completeness 定理の定義において問題としていないからである．この節では論理関係を用いて TDPE の正当性を証明する．

まずは二項関係を二つ定義する：

$$
\begin{array}{lcl}
\texttt{interp\_nf} & : & \forall b, \forall \{A\}, \mathsf{nf}\ (V\ b)\ A\ \to V\ b\ A \to \mathit{Prop} \\
\texttt{interp\_nf}\ b\ f\ v & \Longleftrightarrow & \mathsf{soundness}\ b\ (\mathsf{tm\_of\_nf}(f)) = v \\
\texttt{interp\_ne} & : & \forall b, \forall \{A\}, \mathsf{ne}\ (V\ b)\ A \to V\ b\ A \to \mathit{Prop} \\
\texttt{interp\_ne}\ b\ e\ v & \Longleftrightarrow & \mathsf{soundness}\ b\ (\mathsf{tm\_of\_ne}(e)) = v
\end{array}
$$

$\texttt{interp\_nf}\ b\ f\ v$ は，normal form f が value v へと評価されることを意味している．同様に，$\texttt{interp\_ne}\ b\ e\ v$ は，neutral term e が value v へと評価されることを意味する．Prop は命題論理の型を表しており，Coq においても同じ名前 `Prop` が使われている．（Prop は proposition から名付けられている．）

本節の目的は，以下の正当性定理を証明することにある．

定理 15 (Correctness, Λ_{cbn})**.**

$$
\begin{array}{l}
\forall b, \forall A, \forall (T : \mathsf{TM}(A)), \\
\texttt{interp\_nf}\ b\ (\texttt{reify}\ (V\ b)\ A\ (\mathsf{soundness2}\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ T)) \\
\qquad\qquad (\mathsf{soundness2}\ b\ T)
\end{array}
$$

term T に型が付いているならば，その TDPE 結果も元の T と同じ振る舞いをする，つまり同じ semantics を持っているはずである．上の正当性定理において `soundness2` が二カ所に登場しているが，二番目の引数には同じ T が与えられているにも関わらず，第一引数にはそれぞれ異なる式（`(ne (V b) base)` と `b`）が与えられていることに注意したい．これは base 型のケースにおいてそれぞれ異なる実行を行う為であり，特に前者は実行結果を `reify`（この関数では，base ケースにおいては neutral term として実行される必要がある）に渡す必要があるためである．（言い換えれば，T は前者においては residualizing interpretation を行う必要がある．）

この正当性定理を証明する為には，論理関係を用いる必要がある．論理関係 R は reification-time value e と run-time value v が以下の如くに関係付いていることを定義している：

$$\begin{array}{lcl} \mathsf{R} & : & \forall b, \forall A, V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ A \to V\ b\ A \to * \\ \mathsf{R}\ b\ \mathsf{base}\ e\ v & \iff & \texttt{interp\_ne}\ b\ e\ v \\ \mathsf{R}\ b\ \mathsf{arrow}(A, B)\ e\ v & \iff & \forall e', \forall v', \mathsf{R}\ b\ A\ e'\ v' \Rightarrow \mathsf{R}\ b\ B\ (e\ e')\ (v\ v') \end{array}$$

この論理関係は型（typ）に関する帰納法を用いて定義されている．型が base のケースでは，入力値 e が v へと評価されるならば，e と v は R で関係付いていることを意味している．前述したように base のケースにおいては，e は neutral term であるので，reification 時には簡約が行われないことに注意したい．そして型が関数型の場合には，関係付いている term e' と v' をそれぞれ e と v に渡した式同士が関係付いているならば，e と v も R で関係付いていることを意味している．定義を比較すると分かるように，論理関係 R $b\ A\ e\ v$ は $V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ A$ に，e が v に評価されるという情報を加えた定義となっている．Coq では，R は 4 章の $\models$ と同じく `Fixpoint` を使って定義される：

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

正当性定理を証明するには補題を用いる必要がある．その内の二つは completeness 定理と R の関係に対する補題である：

定理 16 (`reify_R`/`reflect_R`, Λ_{cbn})**.**

$$\begin{array}{ll} (\texttt{reify\_R}) & \forall b, \forall A, \forall v, \forall f, \mathsf{R}\ b\ A\ v\ f \Rightarrow \texttt{interp\_nf}\ b\ (\texttt{reify}\ (V\ b)\ A\ v)\ f \\ (\texttt{reflect\_R}) & \forall b, \forall A, \forall e, \forall v, \texttt{interp\_ne}\ b\ e\ v \Rightarrow \mathsf{R}\ b\ A\ (\texttt{reflect}\ (V\ b)\ A\ e)\ v \end{array}$$

この補題は completeness 定理を拡張したものであり，term の実行に関する情報が与えられている．例えば，`reify_R` は v と v に reification を行った式は共に同じ value f へと評価されることを表す補題である．`reflect_R` も同様である．これら補題の証明は completeness 定理と同様の流れで（型 A に関する帰納法を用いることによって）行うことが出来る．注意しなくてはならないのは，term の構成を把握しておかなくてはいけないことと，`interp_nf` と `interp_ne` が互いに影響し合っている点である．この補題を証明する為に，以下の諸補題を使用する．

補題 8 (interp_nf_ne, Λ_{cbn}).

$$\forall b, \forall e, \forall v, \texttt{interp\_ne}\ b\ e\ v\ \Rightarrow\ \texttt{interp\_nf}\ b\ \mathsf{nf\_ne}(e)\ v$$

補題 9 (interp_nf_Lam, Λ_{cbn}).

$$\forall b, \forall f, \forall v, (\forall v', \texttt{interp\_nf}\ b\ (f\ v')\ (v\ v'))\ \Rightarrow\ \texttt{interp\_nf}\ b\ \mathsf{lam}(f)\ v$$

補題 10 (interp_ne_Var, Λ_{cbn}).

$$\forall b, \forall v, \texttt{interp\_ne}\ b\ \mathsf{var}(v)\ v$$

補題 11 (interp_ne_App, Λ_{cbn}).

$$\forall b, \forall e, \forall e', \forall f, \forall f', \texttt{interp\_ne}\ b\ e\ e'\ \Rightarrow\ \texttt{interp\_nf}\ b\ f\ f'\ \Rightarrow\ \texttt{interp\_ne}\ b\ \mathsf{app}(e, f)\ (e'\ f')$$

これら四つの補題は式を展開すれば簡単に証明することが出来る．これらは PHOAS の term を，first-order で定義しているかの如くに扱う為のものである．二番目の補題 interp_nf_Lam では，PHOAS で定義した λ 抽象がどのように interp_nf で関係付いているかを示している．ここで interp_nf_Lam の証明について留意すべきことは，Coq で証明する際に tactic extensionality を使う必要があることである．何故ならば，extensionality は二つの（Coq の）関数が等しいことを示すために使う tactic であるが，この tactic を使う必要が生じるのは，λ 抽象を PHOAS で表しているが故に，Coq の関数が定義に使われているからである．interp_nf_Lam は，reify_R の型が $A = \mathsf{arrow}(A, B)$ のケースにおける証明において使われる．v が f へと評価される（つまり，$\mathsf{R}\ b\ A\ v\ f$ が成立する）とき，v の reification が行われた式は f そのものへと評価されるのではなく，引数を渡されたときに f と同じ振る舞いをする．故に soundness 定理や completeness 定理の証明では必要ないにも関わらず，TDPE の正当性定理を Coq で証明する為には，extensionality tactic が必要となる．

正当性定理を証明する為には，さらに以下の論理関係 R の成立に関する補題が必要となる．

補題 12 (main, Λ_{cbn}).

$$\forall b, \forall A, \forall (T : \mathsf{TM}(A)), \mathsf{R}\ b\ A\ (\mathsf{soundness2}\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ T)\ (\mathsf{soundness2}\ b\ T)$$

この補題は，A 型を持つ dynamic term T に対して reification を行って得られた value と standard interpretation を行って得られた value が R で関係付いていることを意味している．この補題が証明出来れば，正当性定理は単に補題を二つ適用するだけで証明出来る：

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

しかし一見すると，main は Coq では証明出来ないように思える．証明は T の構造に関する帰納法を用いて行いたいが，T は PHOAS で定義されているため，V でパラメータ化されており，帰納的構造をしていない．故に帰納法が使えるようにするためには，まず T の定義を展開する必要がある．補題 main の soundness2 関数を展開することによって，以下の式が得られる：

$$\begin{aligned}&\mathsf{R}\ b\ A\ (\mathsf{soundness}\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ (T\ (V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))))\\&\quad(\mathsf{soundness}\ b\ (T\ (V\ b)))\end{aligned}$$

ここで，T が $\mathsf{TM}(A)$ 型を持つならば，$(T\ (V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})))$ と $(T\ (V\ b))$ はそれぞれ $\mathsf{tm}\ (V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\ A$ 型と $\mathsf{tm}\ (V\ b)\ A$ 型を持ち，共に帰納的構造をしている．この二つの term は base type の interpretation を除けば全く同一の形をしている．この性質を前提条件として得られるように，main を修正した補題を定義する：

補題 13 (main', Λ_{cbn})**.**

$$\begin{aligned}&\forall b, \forall A, \forall t_1, \forall t_2,\\&\mathsf{related\_term}\ b\ t_1\ t_2\ \Rightarrow\ \mathsf{R}\ b\ A\ (\mathsf{soundness}\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})\ t_1)\ (\mathsf{soundness}\ b\ t_2)\end{aligned}$$

この補題で使われている論理関係 related_term は，以下の如くに定義される：

$$\begin{array}{lcl}\mathsf{related\_term} & : & \forall b, \forall \{A\},\\ & & \mathsf{tm}\ (V\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\ A \to \mathsf{tm}\ (V\ b)\ A \to *\\ \mathsf{related\_term}\ b\ \mathsf{var}(v)\ \mathsf{var}(v') & \iff & \mathsf{R}\ b\ A\ v\ v'\\ \mathsf{related\_term}\ b\ \mathsf{lam}(t)\ \mathsf{lam}(t') & \iff & \forall v, \forall v', \mathsf{R}\ b\ A\ v\ v'\ \Rightarrow\ \mathsf{related\_term}\ b\ (t\ v)\ (t'\ v')\\ \mathsf{related\_term}\ b\ \mathsf{app}(t_1, t_2)\ \mathsf{app}(t_1', t_2') & \iff & \mathsf{related\_term}\ b\ t_1\ t_1' \wedge \mathsf{related\_term}\ b\ t_2\ t_2'\end{array}$$

related_term $b\ t_1\ t_2$ は，二つの式 t_1 と t_2 が同じ構造をしており，それぞれの sub term についても適切な関係が成立していることを意味している．補題 main' は，related_term $b\ t_1\ t_2$ に関する帰納法を用いることによって証明することが出来る．related_term は，Coq では Inductive コマンドを用いて以下の如くに定義される：

```
Inductive related_term b : forall {A},
                           tm (V (ne (V b) base)) A -> tm (V b) A -> Type :=
| related_Var : forall A (v: V (ne (V b) base) A) (v': V b A),
                R b A v v' -> related_term b (tm_Var v) (tm_Var v')
```

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

以下の性質が成り立つならば，main' を使って簡単に main が証明出来る．この性質は，同じ term T を異なるインタプリタに通しても，それぞれで得られた式同士には related_term の関係が成立していることを表している．

性質 1.

$$\forall b, \forall T, \mathsf{related\_term}\ b\ \ (T\ (V\ \mathsf{ne}\ (V\ b)\ \mathsf{base}))\ (T\ (V\ b))$$

この性質は，related_term の定義を考えれば簡単に証明出来るように思えるが，実際には Coq で証明することは出来ない．何故なら term の定義に higher-order を用いている為，T に関して場合分けを行いたくとも，T は（static な，つまり本研究ではメタ言語 Coq の）関数である為，それは叶わない．Coq で証明を行う代わりに，次節で手作業の証明を示す．正当性証明を Coq で定式化する際には，上の性質は（成立するものと仮定して）公理として定義する．(Chlipala [8] も，PHOAS を用いた CPS 変換の正当性を証明する際に，類似の性質を公理として仮定している．)

5.1.5　性質の証明

この節では，前節で定義した性質 1 を手作業で証明する．手順としては，まず性質 1 を open term に関して一般化した定理を証明し，そしてその定理の系として性質 1 を示す．

型の付いている term T はそれぞれ $A_1, ..., A_n$ 型を持つ自由変数 $x_1, ..., x_n$ を含む，つまり，sub term $\mathsf{var}(x_i)$ を含んでいるとする．(ここで x_i はメタ言語 Coq の自由変数であるとする．) $T[t_i/x_i]$ は，T の中の x_i を t_i に置換した式を表す．以下が，open term について一般化した定理である：

定理 17. 各 i について $\mathsf{R}\ b\ A_i\ t_i\ t'_i$ が成り立つならば，以下の式が成立する．

$$\mathsf{related\_term}\ b\ (T[t_i/x_i]\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\ (T[t'_i/x_i]\ (V\ b))$$

Proof. T の構造に関する帰納法を用いる．本証明では，メタ言語（Coq）が変数名の衝突を自動的に避けてくれることを利用する．

$(T = \lambda\mathcal{V}.\mathbf{var}(x_i))$ 証明したい式は以下．

$$\begin{array}{l}\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.\mathsf{var}(t_i))\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ \qquad((\lambda\mathcal{V}.\mathsf{var}(t'_i))\ (V\ b))\end{array}$$

この式は以下の式に簡約出来る：

$$\mathsf{related\_term}\ b\ (\mathsf{var}(t_i))\ (\mathsf{var}(t'_i))$$

related_term の定義から，$\mathsf{R}\ b\ A_i\ t_i\ t'_i$ を証明する必要があるが，この式は既に仮定として成立している．

$(T = \lambda\mathcal{V}.\mathbf{lam}(\lambda x.t))$ 証明したい式は以下．

$$\begin{array}{l}\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.\mathsf{lam}(\lambda x.t[t_i/x_i]))\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ \qquad((\lambda\mathcal{V}.\mathsf{lam}(\lambda x.t[t'_i/x_i]))\ (V\ b))\end{array}$$

この式は以下の式に変形出来る：

$$\begin{array}{l}\mathsf{related\_term}\ b\ (\mathsf{lam}(\lambda x.((\lambda\mathcal{V}.t[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))))\\ \qquad(\mathsf{lam}(\lambda x.((\lambda\mathcal{V}.t[t'_i/x_i])\ (V\ b))))\end{array}$$

related_term の定義により，A 型を持ち $\mathsf{R}\ b\ A\ t\ t'$ を満たす任意の t と t' について，

$$\begin{array}{l}\mathsf{related\_term}\ b\ ((\lambda x.((\lambda\mathcal{V}.t[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})))\ t)\\ \qquad((\lambda\mathcal{V}.((\lambda\mathcal{V}.t[t'_i/x_i])\ (V\ b)))\ t')\end{array}$$

が成立することを示さなくてはならない．この式は以下に簡約出来る：

$$\begin{array}{l}\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.t[t_i/x_i, t/\mathtt{x}])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ \qquad((\lambda\mathcal{V}.t[t'_i/x_i, t'/\mathtt{x}])\ (V\ b))\end{array}$$

この式は帰納法の仮定により成立する．

$(T = \lambda\mathcal{V}.\mathbf{app}(t_1, t_2))$ 証明したい式は以下.

$$\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.\mathsf{app}(t_1[t_i/x_i], t_2[t_i/x_i]))\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ ((\lambda\mathcal{V}.\mathsf{app}(t_1[t'_i/x_i], t_2[t'_i/x_i]))\ (V\ b))$$

この式は以下の式に変形出来る：

$$\mathsf{related\_term}\ b\ (\mathsf{app}(((\lambda\mathcal{V}.t_1[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base})), ((\lambda\mathcal{V}.t_2[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))))\\ (\mathsf{app}(((\lambda\mathcal{V}.t_1[t'_i/x_i])\ (V\ b)), ((\lambda\mathcal{V}.t_2[t'_i/x_i])\ (V\ b))))$$

related_term の定義により，以下の二式が成立することを示さなくてはならない.

$$\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.t_1[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ ((\lambda\mathcal{V}.t_1[t'_i/x_i])\ (V\ b))$$

$$\mathsf{related\_term}\ b\ ((\lambda\mathcal{V}.t_2[t_i/x_i])\ (\mathsf{ne}\ (V\ b)\ \mathsf{base}))\\ ((\lambda\mathcal{V}.t_2[t'_i/x_i])\ (V\ b))$$

両式共に，帰納法の仮定により成立する.

□

T はメタ言語 Coq の関数であるため，上の証明における T の構造に関する場合分けは，恐らくはそうなるだろうと予想されるものの，本当に正しい場合分けであるという保証はない．故に，厳密にいえば上の証明は完全ではない．証明を完全なものにするために，場合分けの正しさを証明したいが，本研究ではどのような方法をとれば良いのか未だ分かっていない.

上の定理を T が closed である（つまり，$n = 0$）場合に限定することによって，前節で示したかった以下の系を得ることが出来る.

系 2. 任意の closed term T について, 以下の式が成立する.

$$\mathsf{related\_term}\ b\ (T\ (V\ \mathsf{ne}\ (V\ b)\ \mathsf{base}))\ (T\ (V\ b))$$

よって，上の系を Coq で以下の如くに公理として仮定することが出来る．この公理は補題 main を証明するために使われる.

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

5.2　Call-by-value の TDPE の正当性証明

本節では CBV の TDPE における正当性を示す．4 章と同じく，本節の定式化は前節の定式化を CPS 変換することによって行う．正当性定理の証明自体は Filinski [15, 16] の証明を参考にして行っている．まずは PHOAS を用いた CBV の λ 計算（Λ_{cbv}）を定義する．

5.2.1　Λ_{cbv} の定式化

型と term は以下の如く定義する．型は Λ_{cbn} と同じであるが，term は value をきちんと区別する．value を区別する必要があるのは，本当に TDPE が CBV になっているのか（言い換えると，β 簡約が行われる際の関数に渡される引数が value になっているか）に関しても正当性定理の中で示す必要があるからである．

$$
\begin{array}{rcrcrcl}
\text{type} & : & \mathsf{typ} & \ni & A, B & := & \mathsf{base} \mid \mathsf{arrow}(A, B) \\
\text{pre-value} & : & \mathsf{vl}\ \mathcal{V}\ A & \ni & v & := & \mathsf{var}(x) \mid \mathsf{lam}(\lambda x.t) \\
\text{pre-term} & : & \mathsf{tm}\ \mathcal{V}\ A & \ni & t & := & \mathsf{val}(v) \mid \mathsf{app}(t_1, t_2) \\
\text{value} & : & \mathsf{VL}(A) & \ni & Vl & := & \lambda\mathcal{V}.v : \mathsf{vl}\ \mathcal{V}\ A \\
\text{term} & : & \mathsf{TM}(A) & \ni & Tm & := & \lambda\mathcal{V}.t : \mathsf{tm}\ \mathcal{V}\ A
\end{array}
$$

value は変数または λ 抽象，term は value である（識別子 $\mathsf{val}(\_)$ で表す）か application のどちらかである．Λ_{cbn} と同じく PHOAS で定義している為，定義出来る term（と value）は closed term のみである．例えば恒等関数と value $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$ は，上の定義を使うとそれぞれ以下の如くに書ける：

$$\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{val}(\mathsf{var}(x)))$$

$$\lambda\mathcal{V}.\mathsf{lam}(\lambda x.\mathsf{app}(\mathsf{val}(\mathsf{lam}(\lambda y.\mathsf{val}(\mathsf{var}(y)))), \mathsf{val}(\mathsf{var}(x))))$$

これら value は，Coq においてはそれぞれ以下の如くに定義出来る：

```
Example VL_id : VL (arrow base base) := fun var =>
  vl_Lam (fun x => tm_Val (vl_Var x)).

Example VL_id': VL (arrow base base) := fun var =>
  vl_Lam (fun x => tm_App (tm_Val (vl_Lam (fun y => (tm_Val (vl_Var y)))))
                          (tm_Val (vl_Var x))).
```

又，各 v，t は必ず以下の型を持つとする：

$$
\begin{aligned}
\mathsf{vl}\ \mathcal{V} &: \mathsf{typ} \to * \\
\mathsf{var} &: \forall\{A : \mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{vl}\ \mathcal{V}\ A \\
\mathsf{lam} &: \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, (\mathcal{V}(A) \to \mathsf{tm}\ \mathcal{V}\ B) \to \mathsf{vl}\ \mathcal{V}\ \mathsf{arrow}(A, B) \\
\mathsf{tm}\ \mathcal{V} &: \mathsf{typ} \to * \\
\mathsf{val} &: \forall\{A : \mathsf{typ}\}, \mathsf{vl}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ A \\
\mathsf{app} &: \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, \mathsf{tm}\ \mathcal{V}\ \mathsf{arrow}(A, B) \to \mathsf{tm}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ B
\end{aligned}
$$

ここでも，t，v を記述する際，型に関する記述は省略していることに注意したい．

次に正規形を定義する．ここでは Filinski [16] に倣い，normal value と normal computation によって正規形を定義する．

$$
\begin{aligned}
& & \mathsf{nv}\ \mathcal{V}\ A \ni nv &:= \mathsf{var}(x) \mid \mathsf{lam}(\lambda x.nc) \\
& & \mathsf{nc}\ \mathcal{V}\ A \ni nc &:= \mathsf{nc\_nv}(nv) \mid \mathsf{let\_app}(x, nv, \lambda y.nc) \\
\text{normal value} &: \mathsf{NV}(A) \ni Nv &:= \lambda\mathcal{V}.nv : \mathsf{nv}\ \mathcal{V}\ A \\
\text{normal computation} &: \mathsf{NC}(A) \ni Nc &:= \lambda\mathcal{V}.nc : \mathsf{nc}\ \mathcal{V}\ A
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{nv}\ \mathcal{V}\ \_ &: \mathsf{typ} \to * \\
\mathsf{var} &: \forall\{A : \mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{nv}\ \mathcal{V}\ A \\
\mathsf{lam} &: \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, (\mathcal{V}(A) \to \mathsf{nc}\ \mathcal{V}\ B) \to \mathsf{nv}\ \mathcal{V}\ \mathsf{arrow}(A, B) \\
\mathsf{ne}\ \mathcal{V}\ \_ &: \mathsf{typ} \to * \\
\mathsf{nc\_nv} &: \forall\{A : \mathsf{typ}\}, \mathsf{nv}\ \mathcal{V}\ A \to \mathsf{nc}\ \mathcal{V}\ A \\
\mathsf{let\_app} &: \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, \forall\{C : \mathsf{typ}\}, \\
&\quad \mathcal{V}(\mathsf{arrow}(A, B)) \to \mathsf{nv}\ \mathcal{V}\ A \to (\mathcal{V}(B) \to \mathsf{nc}\ \mathcal{V}\ C) \to \mathsf{nc}\ \mathcal{V}\ C
\end{aligned}
$$

名前の通り，normal value がこれ以上（λ 抽象の body 部分も含めて）簡約出来ない value，normal computation がこれ以上実行を進められない term を意味している．$\mathsf{let\_app}(x, nv, \lambda y.nc)$ は $\mathsf{app}(\mathsf{var}(x), nv)$ を変数 y で束縛して nc を実行する let 式であり，つまり変数を正規形で呼び出した（これ以上実行を進められない）application を，let 文で残している．この式の意味は term の定義における $\mathsf{app}(\mathsf{lam}(\lambda y.nc), \mathsf{app}(\mathsf{var}(x), nv))$ と同じであるが，4 章で let 文を導入したときと同じく，見やすさのために導入している．そして以下は normal value，normal computation を，それぞれ対応する value，term に変換する関数である：

$$
\begin{aligned}
\mathsf{vl\_of\_nv}(\_) &: \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{nv}\ \mathcal{V}\ A \to \mathsf{vl}\ \mathcal{V}\ A \\
\mathsf{vl\_of\_nv}(\mathsf{var}(x)) &= \mathsf{var}(x) \\
\mathsf{vl\_of\_nv}(\mathsf{lam}(\lambda x.nc)) &= \mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nc}((\lambda x.nc)x')) \\
\\
\mathsf{tm\_of\_nc}(\_) &: \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{nc}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ A \\
\mathsf{tm\_of\_nc}(\mathsf{nc\_nv}(nv)) &= \mathsf{val}(\mathsf{vl\_of\_nv}(nv)) \\
\mathsf{tm\_of\_nc}(\mathsf{let\_app}(x, nv, \lambda y.nc)) &= \mathsf{app}(\mathsf{val}(\mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nc}((\lambda y.nc)\ x'))), \\
&\qquad \mathsf{app}(\mathsf{val}(\mathsf{var}(x)), \mathsf{val}(\mathsf{vl\_of\_nv}(nv))))
\end{aligned}
$$

5.2.2 Soundness/completeness

本節では型に関する二種類の semantics を使用する．V_s が standard interpretation に対応する semantics を定義する論理関係，V_r が residualing interpretation に対応する semantics を定義する論理関係である．

$$
\begin{aligned}
V_s\ b\ \mathcal{V}\ \mathsf{base} &\iff b\ \mathcal{V}\ \mathsf{base} \\
V_s\ b\ \mathcal{V}\ \mathsf{arrow}(A, B) &\iff V_s\ b\ \mathcal{V}\ A \Rightarrow (V_s\ b\ \mathcal{V}\ B \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat}
\end{aligned}
$$

$$
\begin{aligned}
V_r\ \mathcal{V}\ \mathsf{base} &\iff \mathsf{nv}\ \mathcal{V}\ \mathsf{base} \\
V_r\ \mathcal{V}\ \mathsf{arrow}(A, B) &\iff V_r\ \mathcal{V}\ A \Rightarrow \forall D, (V_r\ \mathcal{V}\ B \Rightarrow \mathsf{nc}\ \mathcal{V}\ D) \Rightarrow \mathsf{nc}\ \mathcal{V}\ D
\end{aligned}
$$

両定義共に $\mathsf{arrow}(A, B)$ のケースの定義が CPS となっているが，特に V_r が 4 章の λ_{cbv} で使用した論理関係 $\models$ に対応している．V_s の定義は本来ならば answer type に任意の型が入るようにしたいところであるが，そうすると answer type に自分自身が入り得るため，Coq では定義出来ない．故にここでは V_s の answer type を暫定的に nat に固定している．(nat でなく bool 等の別の型に置き換えても構わない.)

V_s と V_r について，それぞれ以下の soundness 定理が成立する．証明は共に前件部に関する帰納法を用いることによって簡単に示せる．

定理 18 (Soundness, V_s, Λ_{cbv})**.**

$$
\begin{aligned}
&(\text{soundness\_s\_tm}) \quad \forall\mathcal{V}, \forall\{A\}, \forall\{b\}, \mathsf{tm}\ (V_s\ b\ \mathcal{V})\ A \Rightarrow (V_s\ b\ \mathcal{V}\ A \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat} \\
&(\text{soundness\_s\_vl}) \quad \forall\mathcal{V}, \forall\{A\}, \forall\{b\}, \mathsf{vl}\ (V_s\ b\ \mathcal{V})\ A \Rightarrow V_s\ b\ \mathcal{V}\ A
\end{aligned}
$$

定理 19 (Soundness, V_r, Λ_{cbv})**.**

$$
\begin{aligned}
&(\text{soundness\_r\_tm}) \quad \forall\mathcal{V}, \forall\{A\}, \mathsf{tm}\ (V_r\ \mathcal{V})\ A \Rightarrow \forall D, (V_r\ \mathcal{V}\ A \Rightarrow \mathsf{nc}\ \mathcal{V}\ D) \Rightarrow \mathsf{nc}\ \mathcal{V}\ D \\
&(\text{soundness\_r\_vl}) \quad \forall\mathcal{V}, \forall\{A\}, \mathsf{vl}\ (V_r\ \mathcal{V})\ A \Rightarrow V_r\ \mathcal{V}\ A
\end{aligned}
$$

定理 18，定理 19 の proof term がそれぞれ standard interpreter，residualizing interpreter となっている：

```
Fixpoint soundness_s_tm var {A b} (e: tm (Vs b var) A)
        : (Vs b var A -> nat) -> nat :=
  match e with
| tm_Val _ v => fun k => k (soundness_s_vl var v)
| tm_App _ _ e1 e2 => fun k =>
  soundness_s_tm var e1 (fun x1 =>
  soundness_s_tm var e2 (fun x2 =>
  x1 x2 k))
  end
with soundness_s_vl var {A b} (e: vl (Vs b var) A) : Vs b var A :=
  match e with
| vl_Var _ v => v
| vl_Lam _ _ t => fun x k => soundness_s_tm var (t x) k
  end.

Fixpoint soundness_r_tm var {A} (e: tm (Vr var) A)
        : forall D, (Vr var A -> nc var D) -> nc var D :=
  match e with
| tm_Val _ v => fun _ k => k (soundness_r_vl var v)
| tm_App _ _ e1 e2 => fun _ k =>
  soundness_r_tm var e1 _ (fun x1 =>
  soundness_r_tm var e2 _ (fun x2 =>
  x1 x2 _ k))
  end
with soundness_r_vl var {A} (e: vl (Vr var) A) : Vr var A :=
  match e with
| vl_Var _ v => v
```

```
| vl_Lam _ _ t => fun x _ k => soundness_r_tm var (t x) _ k
  end.
```

これら関数を使って，term と value における各 interpretation は以下の如くに定義出来る：

```
Definition soundness_s_TM {A b} var (T: TM A) :=
  soundness_s_tm var (T (Vs b var)).
Definition soundness_s_VL {A b} var (T: VL A) :=
  soundness_s_vl var (T (Vs b var)).
Definition soundness_r_TM {A} var (T: TM A) :=
  soundness_r_tm var (T (Vr var)).
Definition soundness_r_VL {A} var (T: VL A) :=
  soundness_r_vl var (T (Vr var)).
```

V_r について，(Λ_{cbv} の，つまり CBV の）reify/reflect と Curry Howard 同型の completeness 定理が成立する．定理の形自体は，前節と同じである．

定理 20 (Completeness (reify/reflect), Λ_{cbv})**.**

$$\forall \mathcal{V}, \forall A, \quad \begin{array}{ll} (i) & V_r\ \mathcal{V}\ A \Rightarrow \mathsf{nv}\ \mathcal{V}\ A \\ (ii) & \mathcal{V}(A) \Rightarrow V_r\ \mathcal{V}\ A \end{array}$$

この completeness 定理の証明から，以下の reify/reflect 関数が抽出できる．

```
Fixpoint reify (var: typ -> Type) (A: typ) :=
  match A return Vr var A -> nv var A with
| base => fun v => v
| arrow A B => fun v =>
    nv_Lam (fun x => v (reflect var A x) B (fun x' => nc_nv (reify var B x')))
  end
with reflect (var: typ -> Type) (A: typ) :=
  match A return var A -> Vr var A with
| base => fun e => nv_Var e
| arrow A B => fun e v _ k =>
```

```
    nc_LetApp e (reify var A v) (fun g => k (reflect var B g))
  end.
```

比較すれば，PHOAS で定義している為に reify/reflect 関数の引数として var（$\mathcal{V}$）を受け取る必要があること以外は，図 3.3 と正に同じプログラムになっていることが分かる．

本節で得られた CBV の TDPE の実行例を示す．まず，CBN の TDPE の実行例で用いた $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$ と $\underline{\lambda}x.\underline{\lambda}y.(\underline{\lambda}z.z)\,\underline{@}\,(x\,\underline{@}\,y)$ を表す式は，それぞれ本節では以下のごとく定義出来る：

```
Example VL_id': VL (arrow base base) := fun var =>
  vl_Lam (fun x => tm_App (tm_Val (vl_Lam (fun y => (tm_Val (vl_Var y)))))
                          (tm_Val (vl_Var x))).


Example VL_t : VL (arrow (arrow base base) (arrow base base)) := fun var =>
  vl_Lam (fun x => tm_Val (vl_Lam (fun y =>
    tm_App (tm_Val (vl_Lam (fun z => tm_Val (vl_Var z))))
           (tm_App (tm_Val (vl_Var x)) (tm_Val (vl_Var y)))))).
```

そして，これらの式を TDPE で実行する．つまり，residualizing interpreter である soundness_r_VL に上の式を渡して，対応する semantics へと変換し，その結果を reify 関数に渡して実行する．すると，それぞれ以下の結果が得られる：

```
Eval compute in (reify var _ (soundness_r_VL _ VL_id')).

= nv_Lam (fun x :  var base => nc_nv (nv_Var x))
:  nv var (arrow base base)

Eval compute in (reify var _ (soundness_r_VL _ VL_t)).

= nv_Lam (fun x :  var (arrow base base) =>
          nc_nv
          (nv_Lam (fun x0 :  var base =>
                   nc_LetApp x (nv_Var x0)
```

```
                                    (fun g :  var base => nc_nv (nv_Var g))))
:  nv var (arrow (arrow base base) (arrow base base))
```

前者は前節の実行例同様，恒等関数へと部分評価されている．後者は前節と異なり，application の引数部分が value になっていないため，関数の body は簡約されず，application $(x \mathbin{\underline{@}} y)$ を let 文で残した式 $\underline{\lambda} x.\underline{\lambda} y.\underline{\mathrm{let}}\ z = x \mathbin{\underline{@}} y\ \underline{\mathrm{in}}\ z$（$= \underline{\lambda} x.\underline{\lambda} y.(\underline{\lambda} z.z) \mathbin{\underline{@}} (x \mathbin{\underline{@}} y)$）が実行結果として得られている．

5.2.3　正当性定理の証明

上述の reify/reflect 関数が実行前後で term の semantics を変化させないことを示すために，Λ_{cbn} のケースと同様にして二項関係を nv と nc のそれぞれに対して定義する：

$$
\begin{array}{lcl}
\texttt{interp\_nv} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \mathsf{nv}\ (V_s\ b\ \mathcal{V})\ A \to V_s\ b\ \mathcal{V}\ A \to \mathit{Prop} \\
\texttt{interp\_nv}\ b\ \mathcal{V}\ nv\ v & \Longleftrightarrow & \mathsf{soundness\_s\_vl}\ \mathcal{V}\ (\mathsf{vl\_of\_nv}(nv)) = v \\
\texttt{interp\_nc} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ A \to ((V_s\ b\ \mathcal{V}\ A \to \mathsf{nat}) \to \mathsf{nat}) \to \mathit{Prop} \\
\texttt{interp\_nc}\ b\ \mathcal{V}\ nc\ v & \Longleftrightarrow & \mathsf{soundness\_s\_tm}\ \mathcal{V}\ (\mathsf{tm\_of\_nc}(nc)) = v
\end{array}
$$

ここで注意すべきは，interp_nv が受け取る v の型は（受け取る nv が $\mathsf{nv}\ (V_s\ b\ \mathcal{V})\ A$ 型を持つとき）$V_s\ b\ \mathcal{V}\ A$ である一方で，interp_nc が受け取る v の型は（受け取る nc が $\mathsf{nc}\ (V_s\ b\ \mathcal{V})\ A$ 型を持つとき）$((V_s\ b\ \mathcal{V}\ A \to \mathsf{nat}) \to \mathsf{nat})$ と CPS になっている点である．

この interp_nv を用いて正当性定理が以下のように定義出来る：

定理 21 (Correctness, Λ_{cbv})**.**

$$
\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall A, \forall (Vl : \mathsf{VL}(A)), \\
\texttt{interp\_nv}\ b\ \mathcal{V}\ (\texttt{reify}\ (V_s\ b\ \mathcal{V})\ A\ (\mathsf{soundness\_r\_VL}\ (V_s\ b\ \mathcal{V})\ Vl)) \\
\qquad\qquad (\mathsf{soundness\_s\_VL}\ \mathcal{V}\ Vl)
\end{array}
$$

正当性定理の証明の手順は Λ_{cbn} のときと同じである．まず，reification-time value e と run-time value v の関係性を表す論理関係 R を定義する．Λ_{cbn} のときと同様，R の定義は，residualizing interpretation に対応している V_s に「e が v に評価される」という情報を加えて拡張したものと捉えることが出来る．

$$
\begin{array}{lcl}
\mathsf{R} & : & \forall b, \forall \mathcal{V}, \forall A, V_r\ (V_s\ b\ \mathcal{V})\ A \to V_s\ b\ \mathcal{V}\ A \to * \\
\mathsf{R}\ b\ \mathcal{V}\ \mathsf{base}\ e\ v & \Longleftrightarrow & \mathtt{interp\_nv}\ b\ \mathcal{V}\ e\ v \\
\mathsf{R}\ b\ \mathcal{V}\ \mathsf{arrow}(A, B)\ e\ v & \Longleftrightarrow & \forall e', \forall a', \mathsf{R}\ b\ \mathcal{V}\ A\ e'\ a' \Rightarrow \forall D, \\
 & & \forall (k : V_r\ (V_s\ b\ \mathcal{V})\ B \to \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ D), \\
 & & \forall (k' : V_s\ b\ \mathcal{V}\ B \to (V_s\ b\ \mathcal{V}\ D \to \mathsf{nat}) \to \mathsf{nat}), \\
 & & (\forall e'', \forall a'', \mathsf{R}\ b\ \mathcal{V}\ e''\ a'' \Rightarrow \mathtt{interp\_nc}\ b\ \mathcal{V}\ (k\ e'')\ (\lambda c.\, k'\ a''\ c)) \Rightarrow \\
 & & \mathtt{interp\_nc}\ b\ \mathcal{V}\ (e\ e'\ D\ k)\ (\lambda c.\, v\ a'\ (\lambda x.\, k'\ x\ c))
\end{array}
$$

この論理関係は，前節と同様の性質が成り立つ：

定理 22 ($\mathtt{reify\_R}/\mathtt{reflect\_R}$, Λ_{cbv})**.**

$$
\begin{array}{ll}
(\mathtt{reify\_R}) & \forall b, \forall \mathcal{V}, \forall A, \forall v, \forall f, \mathsf{R}\ b\ \mathcal{V}\ A\ v\ f \Rightarrow \mathtt{interp\_nv}\ b\ \mathcal{V}\ (\mathtt{reify}\ (V_s\ b\ \mathcal{V})\ A\ v)\ f \\
(\mathtt{reflect\_R}) & \forall b, \forall \mathcal{V}, \forall A, \forall x, \forall v, \mathtt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(x)\ v \Rightarrow \mathsf{R}\ b\ \mathcal{V}\ A\ (\mathtt{reflect}\ (V_s\ b\ \mathcal{V})\ A\ x)\ v
\end{array}
$$

使っている論理関係の定義が異なるだけで，形自体は定理 16 と同様である．この定理は，以下の諸補題を使えば，completeness 定理の証明と同じストーリーで証明が出来る．これら諸補題は，展開すれば簡単に成立を示すことが出来る．

補題 14 ($\mathtt{interp\_nc\_nv}$, Λ_{cbv})**.**

$$\forall b, \forall \mathcal{V}, \forall e, \forall v, \mathtt{interp\_nv}\ b\ \mathcal{V}\ e\ v \Rightarrow \mathtt{interp\_nc}\ b\ \mathcal{V}\ \mathsf{nc\_nv}(e)\ (\lambda c.\, c\ v)$$

補題 15 ($\mathtt{interp\_nv\_Lam}$, Λ_{cbv})**.**

$$\forall b, \forall \mathcal{V}, \forall f, \forall v, (\forall x, \mathtt{interp\_nc}\ b\ \mathcal{V}\ (f\ x)\ (v\ x)) \Rightarrow \mathtt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{lam}(f)\ v$$

補題 16 ($\mathtt{interp\_nv\_Var}$, Λ_{cbv})**.**

$$\forall b, \forall \mathcal{V}, \forall v, \mathtt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(v)\ v$$

補題 17 ($\mathtt{interp\_nc\_LetApp}$, Λ_{cbv})**.**

$$
\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall x, \forall v, \mathtt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(x)\ v \Rightarrow \\
\forall e, \forall a, \mathtt{interp\_nv}\ b\ \mathcal{V}\ e\ a \Rightarrow \\
\forall f, \forall f', (\forall x', \forall v', \mathtt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(x')\ v' \Rightarrow \mathtt{interp\_nc}\ b\ \mathcal{V}\ (f\ x')\ (\lambda c.\, f'\ c\ v')) \Rightarrow \\
\mathtt{interp\_nc}\ b\ \mathcal{V}\ \mathsf{let\_app}(x, e, f)\ (\lambda c.\, v\ a\ (f'\ c))
\end{array}
$$

$\mathtt{reify\_R}$ を用いて正当性定理を示すには，やはり以下の補題が成立している必要がある：

補題 18 ($\mathtt{main}$, Λ_{cbv})**.**

$$
\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall A, \forall (Vl : \mathsf{VL}(A)), \\
\mathsf{R}\ b\ \mathcal{V}\ A\ (\mathsf{soundness\_r\_VL}\ (V_s\ b\ \mathcal{V})\ Vl) \\
\qquad (\mathsf{soundness\_s\_VL}\ \mathcal{V}\ Vl)
\end{array}
$$

前節で定義した `main` の term T 同様，ここで使われている Vl は PHOAS で定義されている．故に Vl は $\mathcal{V}$ でパラメータ化されており，Vl 自身に対して帰納法を用いることが出来ない．その為，本節でも `main` を修正した補題を作る必要がある．以下がその補題である：

補題 19 (`main'_vl/main'_tm`, Λ_{cbv})**.**

$$
\begin{array}{ll}
(\text{main'\_vl}) & \forall b, \forall \mathcal{V}, \forall A, \forall v_1, \forall v_2, \\
& \mathsf{related\_vl}\ b\ \mathcal{V}\ v_1\ v_2\ \Rightarrow \\
& \mathsf{R}\ b\ \mathcal{V}\ A\ (\mathsf{soundness\_r\_vl}\ (V_s\ b\ \mathcal{V})\ v_1)\ (\mathsf{soundness\_s\_vl}\ \mathcal{V}\ v_2) \\
(\text{main'\_tm}) & \forall b, \forall \mathcal{V}, \forall A, \forall t_1, \forall t_2, \\
& \mathsf{related\_tm}\ b\ \mathcal{V}\ t_1\ t_2\ \Rightarrow \\
& \forall D, \forall k, \forall k', \\
& (\forall e, \forall a, \mathsf{R}\ b\ \mathcal{V}\ A\ e\ a\ \Rightarrow\ \texttt{interp\_nc}\ b\ \mathcal{V}\ (k\ e)\ (\lambda c.\, k'\ c\ a))\ \Rightarrow \\
& \texttt{interp\_nc}\ b\ \mathcal{V}\ (\mathsf{soundness\_r\_tm}\ (V_s\ b\ \mathcal{V})\ t_1\ D\ k)\ (\lambda c.\, \mathsf{soundness\_s\_tm}\ \mathcal{V}\ t_2\ (k'\ c))
\end{array}
$$

`main` を示す為に使うのは `main'_vl` であるが，この補題を証明するには `main'_tm` と連立させる必要がある．related_tm は前節と同じく，二つの pre-term が同じ構造をしており，かつそれぞれの sub term についても適切な関係が成立していることを表す論理関係である．但し，本節では pre-term を pre-value を用いて定義しているため，pre-value に関しても同様の論理関係を定義する必要がある．それが related_vl である．これら二つの論理関係は以下の如く相互再帰的に定義される：

$$
\begin{array}{lcl}
\mathsf{related\_vl} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \\
& & \mathsf{vl}\ (V_r\ (V_s\ b\ \mathcal{V}))\ A \to \mathsf{vl}\ (V_s\ b\ \mathcal{V})\ A \to * \\
\mathsf{related\_vl}\ b\ \mathcal{V}\ \mathsf{var}(v)\ \mathsf{var}(v') & \Longleftrightarrow & \mathsf{R}\ b\ \mathcal{V}\ A\ v\ v' \\
\mathsf{related\_vl}\ b\ \mathcal{V}\ \mathsf{lam}(t)\ \mathsf{lam}(t') & \Longleftrightarrow & \forall v, \forall v', \mathsf{R}\ b\ \mathcal{V}\ A\ v\ v'\ \Rightarrow\ \mathsf{related\_tm}\ b\ \mathcal{V}\ (t\ v)\ (t'\ v') \\
\mathsf{related\_tm} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \\
& & \mathsf{tm}\ (V_r\ (V_s\ b\ \mathcal{V}))\ A \to \mathsf{tm}\ (V_s\ b\ \mathcal{V})\ A \to * \\
\mathsf{related\_tm}\ b\ \mathcal{V}\ \mathsf{val}(v)\ \mathsf{val}(v') & \Longleftrightarrow & \mathsf{related\_vl}\ b\ \mathcal{V}\ v\ v' \\
\mathsf{related\_tm}\ b\ \mathcal{V}\ \mathsf{app}(t_1, t_2)\ \mathsf{app}(t_1', t_2') & \Longleftrightarrow & \mathsf{related\_tm}\ b\ \mathcal{V}\ t_1\ t_1' \wedge \mathsf{related\_tm}\ b\ \mathcal{V}\ t_2\ t_2'
\end{array}
$$

related_vl と related_tm について以下の性質が成立するならば，`main'_vl` から `main` が証明出来るので，正当性定理が成立することになる．

公理 3 (`Vr_related/T_related`, Λ_{cbv})**.**

$$
\begin{array}{ll}
(\text{Vr\_related}) & \forall b, \forall \mathcal{V}, \forall Vl, \mathsf{related\_vl}\ b\ \mathcal{V}\ (Vl\ (V_r\ (V_s\ b\ \mathcal{V})))\ (Vl\ (V_s\ b\ \mathcal{V})) \\
(\text{T\_related}) & \forall b, \forall \mathcal{V}, \forall Tm, \mathsf{related\_tm}\ b\ \mathcal{V}\ (Tm\ (V_r\ (V_s\ b\ \mathcal{V})))\ (Tm\ (V_s\ b\ \mathcal{V}))
\end{array}
$$

上記性質は，前節と同様にして手作業での証明を行うことが出来る．しかし，やはり higher-order を用いて value と term を定義しているために，Vl と Tm について場合分けを行うことが出来ないため，Coq で証明することは出来ない．そのため，ここでも公理として定義している．

5.3　Shift/reset 付き call-by-value の TDPE の正当性証明

本節では Tsushima らの shift/reset 付き CBV の TDPE の正当性を示す．4 章では $\lambda_{cbv}^{S/R}$ における TDPE の抽出を，λ_{cbv} の定式化を 2CPS に拡張することによって行った．本節も同様に，Λ_{cbv} の定式化を 2CPS に拡張することによって，$\Lambda_{cbv}^{S/R}$ における TDPE の抽出及び正当性の証明を行う．

5.3.1　$\Lambda_{cbv}^{S/R}$ の定式化

まずは shift/reset 付き CBV の λ 計算（$\Lambda_{cbv}^{S/R}$）を定義する．

$$
\begin{array}{rrcrcl}
\text{type} & : & \mathsf{typ} & \ni & A,B,c,d & := & \mathsf{base} \mid A/c \to B/d \\
\text{pre-value} & : & \mathsf{vl}\ \mathcal{V}\ A & \ni & v & := & \mathsf{var}(x) \mid \mathsf{lam}(\lambda x.t) \\
\text{pre-term} & : & \mathsf{tm}\ \mathcal{V}\ A\ c\ d & \ni & t & := & \mathsf{val}(v) \mid \mathsf{app}(t_1,t_2) \mid \mathsf{shift}(\lambda k.t) \mid \mathsf{reset}(t) \\
\text{value} & : & \mathsf{VL}(A) & \ni & Vl & := & \lambda\mathcal{V}.v : \mathsf{vl}\ \mathcal{V}\ A \\
\text{term} & : & \mathsf{TM}(A,c,d) & \ni & Tm & := & \lambda\mathcal{V}.t : \mathsf{tm}\ \mathcal{V}\ A\ c\ d
\end{array}
$$

PHOAS を用いて定義していること以外，基本的には $\lambda_{cbv}^{S/R}$ と同じであるが，Λ_{cbv} のケースと同様の理由により，上の定義においても term と value を区別している．value は変数又は λ 抽象のどちらかであり，term は value であるか，又は application，shift，reset のいずれかである．又，value は pure であり，term は impure である．そのため，$\mathsf{TM}(\_,\_,\_), \mathsf{VL}(\_), \mathsf{tm}, \mathsf{vl}$ で指定する型は，$\mathsf{VL}(\_)$ と vl では（項自身の）型だけ指定すれば良いが，$\mathsf{TM}(\_,\_,\_)$ と tm では項の持つ型の他，その項を実行することによってアンサータイプが何から何に変わるかも記述する必要があるため，三つの型が渡される必要がある．例えば，$\mathsf{TM}(A,c,d)$ 型を持つ term は，その term の型自体は A 型であり，その term を実行するとアンサータイプが c から d に変化することを意味する．$\mathsf{tm}\ \mathcal{V}\ A\ c\ d$ 型についても同様である．継続を取得する命令である $\mathsf{shift}(\_)$ に関しては，$\mathsf{lam}(\_)$ と同じく PHOAS で定義されているため，受け取る式はメタ言語（本研究では Coq）の関数となっていることに注意したい．value と term は必ず以下の型を持つものとする：

$$
\begin{array}{rcl}
\mathsf{vl}\ \mathcal{V} & : & \mathsf{typ} \to * \\
\mathsf{var} & : & \forall\{A : \mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{vl}\ \mathcal{V}\ A \\
\mathsf{lam} & : & \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, \forall\{c : \mathsf{typ}\}, \forall\{d : \mathsf{typ}\}, \\
 & & (\mathcal{V}(A) \to \mathsf{tm}\ \mathcal{V}\ B\ c\ d) \to \mathsf{vl}\ \mathcal{V}\ A/c \to B/d \\
\mathsf{tm}\ \mathcal{V} & : & \mathsf{typ} \to \mathsf{typ} \to \mathsf{typ} \to * \\
\mathsf{val} & : & \forall\{A : \mathsf{typ}\}, \forall\{c : \mathsf{typ}\}, \mathsf{vl}\ \mathcal{V}\ A \to \mathsf{tm}\ \mathcal{V}\ A\ c\ c \\
\mathsf{app} & : & \forall\{A : \mathsf{typ}\}, \forall\{a : \mathsf{typ}\}, \forall\{b : \mathsf{typ}\} \forall\{c : \mathsf{typ}\}, \forall\{d : \mathsf{typ}\}, \\
 & & \mathsf{tm}\ \mathcal{V}\ c/a \to A/b\ r\ d \to \mathsf{tm}\ \mathcal{V}\ c\ b\ r \to \mathsf{tm}\ \mathcal{V}\ A\ a\ d \\
\mathsf{shift} & : & \forall\{A : \mathsf{typ}\}, \forall\{a : \mathsf{typ}\}, \forall\{r : \mathsf{typ}\}, \forall\{b : \mathsf{typ}\}, \forall\{c : \mathsf{typ}\}, \\
 & & (\mathcal{V}(A/r \to a/r) \to \mathsf{tm}\ \mathcal{V}\ c\ c\ b) \to \mathsf{tm}\ \mathcal{V}\ A\ a\ b \\
\mathsf{reset} & : & \forall\{A : \mathsf{typ}\}, \forall\{a : \mathsf{typ}\}, \forall\{r : \mathsf{typ}\}, \mathsf{tm}\ \mathcal{V}\ a\ a\ A \to \mathsf{tm}\ \mathcal{V}\ A\ r\ r
\end{array}
$$

次に normal value，normal computation を定義する．term は impure だが，normal value，normal computation は全て pure であることに注意したい．

$$
\begin{array}{rcl}
 & & \mathsf{nv}\ \mathcal{V}\ A \ \ni\ nv \ := \ \mathsf{var}(x) \mid \mathsf{lam\_shift}(\lambda x.\lambda k.\, nc) \\
 & & \mathsf{nc}\ \mathcal{V}\ A \ \ni\ nc \ := \ \mathsf{nc\_nv}(nv) \mid \mathsf{let\_reset\_app}(x, nv, \lambda y.nc) \\
 & & \qquad\qquad \mid \ \mathsf{let\_reset\_let\_app}(x, nv, \lambda y.nc_1, \lambda z.nc_2) \\
\text{normal value} & : & \mathsf{NV}(A) \ \ni\ Nv \ := \ \lambda\mathcal{V}.nv : \mathsf{nv}\ \mathcal{V}\ A \\
\text{normal computation} & : & \mathsf{NC}(A) \ \ni\ Nc \ := \ \lambda\mathcal{V}.nc : \mathsf{nc}\ \mathcal{V}\ A
\end{array}
$$

$\mathsf{lam\_shift}(\_)$ は λ 抽象の真下に shift が入っている項を表しており，上図の x が λ 抽象によって束縛されている変数，k が shift によって束縛されている変数を表す．例えば 3 章の定義を使って書ける式 $\underline{\lambda}x.\underline{S}k.x$ は，上の定義を使えば $\mathsf{lam\_shift}(\lambda x.\lambda k.\, \mathsf{nc\_nv}(\mathsf{var}(x)))$ と書ける．又，$\mathsf{let\_reset\_app}(x, nv, \lambda y.nc)$ は $\mathsf{reset}(\mathsf{app}(\mathsf{var}(x), nv))$ を変数 y で束縛して nc を実行する let 式である．同様に $\mathsf{let\_reset\_let\_app}(x, nv, \lambda y.nc$ は，$\mathsf{app}(\mathsf{var}(x), nv)$ を変数 y で束縛して nc_1 を実行する let 式が reset で囲われた式を，変数 z で束縛して nc_2 を実行するような let 式である．又，nv, nc は以下の型を持つと定める．

$$
\begin{array}{rcl}
\mathsf{nv}\ \mathcal{V} & : & \mathsf{typ} \to * \\
\mathsf{var} & : & \forall\{A : \mathsf{typ}\}, \mathcal{V}(A) \to \mathsf{nv}\ \mathcal{V}\ A \\
\mathsf{lam\_shift} & : & \forall\{A : \mathsf{typ}\}, \forall\{B : \mathsf{typ}\}, \forall\{a : \mathsf{typ}\}, \forall\{b : \mathsf{typ}\}, \\
 & & (\mathcal{V}(A) \to \mathcal{V}(B/a \to a/a) \to \mathsf{nc}\ \mathcal{V}\ B) \to \\
 & & \mathsf{nv}\ \mathcal{V}\ A/a \to B/b \\
\mathsf{ne}\ \mathcal{V} & : & \mathsf{typ} \to * \\
\mathsf{nc\_nv} & : & \forall\{A : \mathsf{typ}\}, \mathsf{nv}\ \mathcal{V}\ A \to \mathsf{nc}\ \mathcal{V}\ A \\
\mathsf{let\_reset\_app} & : & \forall\{A : \mathsf{typ}\}, \forall\{c' : \mathsf{typ}\}, \forall\{c : \mathsf{typ}\}, \\
 & & \mathcal{V}(c'/c \to c/c) \to \mathsf{nv}\ \mathcal{V}\ c' \to \\
 & & (\mathcal{V}(c) \to \mathsf{nc}\ \mathcal{V}\ A) \to \mathsf{nc}\ \mathcal{V}\ A \\
\mathsf{let\_reset\_let\_app} & : & \forall\{c' : \mathsf{typ}\}, \forall\{c : \mathsf{typ}\}, \forall\{a' : \mathsf{typ}\}, \forall\{r : \mathsf{typ}\}, \forall\{A : \mathsf{typ}\}, \\
 & & \mathcal{V}(c'/a' \to c/r) \to \mathsf{nv}\ \mathcal{V}\ c' \to \\
 & & (\mathcal{V}(c) \to \mathsf{nc}\ \mathcal{V}\ a') \to (\mathcal{V}(r) \to \mathsf{nc}\ \mathcal{V}\ A) \to \mathsf{nc}\ \mathcal{V}\ A
\end{array}
$$

normal value，normal computation を value と term にそれぞれ変換する関数 $\mathsf{vl\_of\_nv}(\_)$，$\mathsf{tm\_of\_nc}(\_)$ は以下のごとくに相互再帰的に定義される：

$$
\begin{array}{rcl}
\mathsf{vl\_of\_nv}(\_) & : & \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{nv}\ \mathcal{V}\ A \to \mathsf{vl}\ \mathcal{V}\ A \\
\mathsf{vl\_of\_nv}(\mathsf{var}(x)) & = & \mathsf{var}(x) \\
\mathsf{vl\_of\_nv}(\mathsf{lam\_shift}(\lambda x.\lambda k.\, nc)) & = & \mathsf{lam}(\lambda x'.\mathsf{shift}(\lambda k'.\mathsf{tm\_of\_nc}((\lambda x.\lambda k.\, nc)\ x'\ k'))) \\
\mathsf{tm\_of\_nc}(\_) & : & \forall\{\mathcal{V}\}, \forall\{A\}, \mathsf{nc}\ \mathcal{V}\ A \to \forall\{r\}, \mathsf{tm}\ \mathcal{V}\ A\ r\ r \\
\mathsf{tm\_of\_nc}(\mathsf{nc\_nv}(nv)) & = & \mathsf{val}(\mathsf{vl\_of\_nv}(nv)) \\
\mathsf{tm\_of\_nc}(\mathsf{let\_reset\_app}(x, nv, \lambda y.nc)) & = & \mathsf{app}(\mathsf{val}(\mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nc}((\lambda y.nc)\ x'))), \\
 & & \quad \mathsf{reset}(\mathsf{app}(\mathsf{val}(\mathsf{var}(x)), \mathsf{val}(\mathsf{vl\_of\_nv}(nv))))) \\
\mathsf{tm\_of\_nc}(\mathsf{let\_reset\_let\_app}(x, nv, f, g)) & = & \mathsf{app}(\mathsf{val}(\mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nc}(g\ x'))), \\
 & & \quad \mathsf{reset}(\mathsf{app}(\mathsf{val}(\mathsf{lam}(\lambda x'.\mathsf{tm\_of\_nc}(f\ x'))), \\
 & & \quad\quad \mathsf{app}(\mathsf{val}(\mathsf{var}(x)), \mathsf{val}(\mathsf{vl\_of\_nv}(nv)))))) \\
\end{array}
$$

5.3.2 Soundness/completeness

型に関する standard semantics を定義する論理関係 V_s と，型に関する residualizing semantics を定義する論理関係 V_r はそれぞれ以下の如くに定義する．どちらも，Λ_{cbv} で定義した V_s と V_r をそれぞれ CPS から 2CPS にそのまま拡張した定義となっている．又，V_r は 4 章で定義した $\lambda_{cbv}^{S/R}$ における論理関係 $\models$ に対応している．

$$
\begin{array}{lcl}
V_s\ b\ \mathcal{V}\ \mathsf{base} & \Longleftrightarrow & b\ \mathcal{V}\ \mathsf{base} \\
V_s\ b\ \mathcal{V}\ A/c \to B/d & \Longleftrightarrow & V_s\ b\ \mathcal{V}\ A\ \Rightarrow \\
 & & (V_s\ b\ \mathcal{V}\ B\ \Rightarrow\ (V_s\ b\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nat})\ \Rightarrow\ \mathsf{nat})\ \Rightarrow \\
 & & (V_s\ b\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nat})\ \Rightarrow\ \mathsf{nat}
\end{array}
$$

$$
\begin{array}{lcl}
V_r\ \mathcal{V}\ \mathsf{base} & \Longleftrightarrow & \mathsf{nv}\ \mathcal{V}\ \mathsf{base} \\
V_r\ \mathcal{V}\ A/c \to B/d & \Longleftrightarrow & V_r\ \mathcal{V}\ A\ \Rightarrow \\
 & & (V_r\ \mathcal{V}\ B\ \Rightarrow\ \forall D, (V_r\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)\ \Rightarrow \\
 & & \forall D', (V_r\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D')\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D'
\end{array}
$$

V_s の $(V_s\ b\ \mathcal{V}\ B\ \Rightarrow\ (V_s\ b\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nat})\ \Rightarrow\ \mathsf{nat})$ と V_r の $(V_r\ \mathcal{V}\ B\ \Rightarrow\ \forall D,\ (V_r\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)$ が継続，$(V_s\ b\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nat})$ と $(V_r\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D')$ がメタ継続となっている．

上のそれぞれの論理関係について，以下の soundness 定理が成立する：

定理 23 (Soundness, V_s, $\Lambda_{cbv}^{S/R}$).

$$
\begin{array}{ll}
(\text{soundness\_s\_tm}) & \forall \mathcal{V}, \forall b, \forall A, \forall c, \forall d, \mathsf{tm}\ (V_s\ b\ \mathcal{V})\ A\ c\ d\ \Rightarrow \\
 & (V_s\ b\ \mathcal{V}\ A\ \Rightarrow\ (V_s\ b\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nat})\ \Rightarrow\ \mathsf{nat})\ \Rightarrow \\
 & (V_s\ b\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nat})\ \Rightarrow\ \mathsf{nat} \\
(\text{soundness\_s\_vl}) & \forall \mathcal{V}, \forall b, \forall A, \mathsf{vl}\ (V_s\ b\ \mathcal{V})\ A\ \Rightarrow\ V_s\ b\ \mathcal{V}\ A
\end{array}
$$

定理 24 (Soundness, V_r, $\Lambda_{cbv}^{S/R}$).

$$
\begin{array}{ll}
(\text{soundness\_r\_tm}) & \forall \mathcal{V}, \forall A, \forall c, \forall d, \mathsf{tm}\ (V_r\ \mathcal{V})\ A\ c\ d\ \Rightarrow \\
 & (V_r\ \mathcal{V}\ A\ \Rightarrow\ \forall D, (V_r\ \mathcal{V}\ c\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D)\ \Rightarrow \\
 & \forall D', (V_r\ \mathcal{V}\ d\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D')\ \Rightarrow\ \mathsf{nc}\ \mathcal{V}\ D' \\
(\text{soundness\_r\_vl}) & \forall \mathcal{V}, \forall A, \mathsf{vl}\ (V_r\ \mathcal{V})\ A\ \Rightarrow\ V_r\ \mathcal{V}\ A
\end{array}
$$

どちらの定理も，定理 18・定理 19 と比較して，前半部の定義が 2CPS の形に拡張されている．これら定理の proof term が図 5.1 の諸インタプリタである．

term と value のためのインタプリタは，それぞれ図 5.1 の関数を使って以下のように定義出来る：

```
Definition soundness_s_TM {A c d b} var (T: TM A c d) :=
  soundness_s_tm var (T (Vs b var)).
Definition soundness_s_VL {A b} var (T: VL A) :=
  soundness_s_vl var (T (Vs b var)).
```

```
Fixpoint soundness_s_tm var {A c d b} (e: tm (Vs b var) A c d)
  : (Vs b var A -> (Vs b var c -> nat) -> nat) ->
    (Vs b var d -> nat) -> nat :=
  match e with
| tm_App _ _ _ _ _ _ t1 t2 => fun k1 k2 =>
  soundness_s_tm var t1 (fun h5 h6 =>
  soundness_s_tm var t2 (fun h7 h8 => h5 h7 k1 h8) h6) k2
| tm_Shift _ _ _ _ _ t => fun k1 k2 =>
  soundness_s_tm var (t (fun h3 h4 h5 => k1 h3 (fun h6 => h4 h6 h5)))
  (fun h8 h9 => h9 h8) k2
| tm_Reset _ _ _ t => fun k1 k2 =>
  soundness_s_tm var t (fun h4 h5 => h5 h4) (fun h6 => k1 h6 k2)
| tm_Val _ _ v => fun  k1 k2 => k1 (soundness_s_vl var v) k2
  end
with soundness_s_vl var {A b} (e: vl (Vs b var) A) : Vs b var A :=
  match e with
| vl_Var _ v => v
| vl_Lam _ _ _ _ t => fun x k1 k2 => soundness_s_tm var (t x) k1 k2
  end.

Fixpoint soundness_r_tm var {A c d} (e: tm (Vr var) A c d)
  : (Vr var A -> forall D, (Vr var c -> nc var D) -> nc var D) ->
    forall D', (Vr var d -> nc var D') -> nc var D' :=
  match e with
| tm_App _ _ _ _ _ _ t1 t2 => fun k1 D' k2 =>
  soundness_r_tm var t1 (fun h5 D h6 =>
  soundness_r_tm var t2 (fun h7 D'' h8 => h5 h7 k1 D'' h8) D h6) D' k2
| tm_Shift _ _ _ _ _ t => fun k1 D' k2 =>
  soundness_r_tm var (t (fun h3 h4 D h5 => k1 h3 D (fun h6 => h4 h6 D h5)))
  (fun h8 D h9 => h9 h8) D' k2
| tm_Reset _ _ _ t => fun k1 D' k2 =>
  soundness_r_tm var t (fun h4 D h5 => h5 h4) D' (fun h6 => k1 h6 D' k2)
| tm_Val _ _ v => fun k1 D' k2 => k1 (soundness_r_vl var v) D' k2
  end
with soundness_r_vl var {A} (e: vl (Vr var) A) : Vr var A :=
  match e with
| vl_Var _ v => v
| vl_Lam _ _ _ _ t => fun x k1 D' k2 => soundness_r_tm var (t x) k1 D' k2
  end.
```

図 5.1: $\Lambda_{cbv}^{S/R}$ の抽出されたインタプリタ

```
Definition soundness_r_TM {A c d} var (T: TM A c d) :=
  soundness_r_tm var (T (Vr var)).
Definition soundness_r_VL {A} var (T: VL A) :=
  soundness_r_vl var (T (Vr var)).
```

次に V_r を用いて reify/reflect を抽出する．$\Lambda_{cbv}^{S/R}$ における reify/reflect と Curry Howard 同型の completeness 定理は，前節までと同一の形をしている：

定理 25 (Completeness (reify/reflect), $\Lambda_{cbv}^{S/R}$)**.**

$$\forall \mathcal{V}, \forall A, \quad \begin{array}{ll} (i) & V_r\ \mathcal{V}\ A \Rightarrow \mathsf{nv}\ \mathcal{V}\ A \\ (ii) & \mathcal{V}(A) \Rightarrow V_r\ \mathcal{V}\ A \end{array}$$

この completeness 定理の proof term となっている reify/reflect 関数が，図 5.2 である．nv_LamShift，nc_LetResetApp，nc_LetResetLetApp がそれぞれ lam_shift，let_reset_app，let_reset_let_app を表している．図 3.5 と比較して，見るからに同一の定義となっていることが分かる．

以上により，shift/reset 付き CBV の TDPE が得られた．この TDPE の実行例を示す．まずは $\underline{\lambda}x.(\underline{\lambda}y.y)\,\underline{@}\,x$ に対応する式を定義する．

```
Example VL_id': VL (arrow base base base base) := fun var =>
  vl_Lam (fun x => tm_App (tm_Val (vl_Lam (fun y => (tm_Val (vl_Var y)))))
                          (tm_Val (vl_Var x))).
```

この式を本節で得られた TDPE にかけると，以下の答えが得られる：

```
Eval compute in (reify var _ (soundness_r_VL _ VL_id')).
```

```
= nv_LamShift (fun (x1 :  var base) (k1 :  var (arrow base base base base)) =>
               nc_LetResetApp k1 (nv_Var x1)
                              (fun g :  var base => nc_nv (nv_Var g)))
:  nv var (arrow base base base base)
```

この実行結果を 3 章の表記に直すと，$\underline{\lambda}x_1.\underline{S}k_1.\underline{\mathrm{let}}\ g = \underline{\langle} k_1\,\underline{@}\,x_1 \underline{\rangle}\ \underline{\mathrm{in}}\ g$ となる．この式は，$(\underline{\lambda}y.y)\,\underline{@}\,x$ を β 簡約して得られる変数 x を変数名 x_1 に付け替え，さらに，x_1 を shift が取ってくる継続 k_1 に渡した application（を reset で括った式）を，let 文で残している．故に，正しく元の式の部分評価結果が得られている．

```
Fixpoint reify (var: typ -> Type) (A: typ) :=
  match A return Vr var A -> nv var A with
| base => fun v => v
| arrow A1 A2 A3 A4 => fun v =>
  nv_LamShift (fun x1 k1 =>
  v (reflect var A1 x1)
    (fun v1 D k2 => nc_LetResetApp k1 (reify var A2 v1)
                                    (fun g => k2 (reflect var A3 g)))
     A4
     (fun v2 => nc_nv (reify var A4 v2)))
  end
with reflect (var: typ -> Type) (A: typ) :=
  match A return var A -> Vr var A with
| base => fun e => nv_Var e
| arrow A1 A2 A3 A4 => fun e v1 k1 D k2 =>
  nc_LetResetLetApp e
                    (reify var A1 v1)
                    (fun x1 => k1 (reflect var A2 x1) A3
                                  (fun v2 => nc_nv (reify var A3 v2)))
                    (fun g => k2 (reflect var A4 g))
  end.
```

図 5.2: $\Lambda_{cbv}^{S/R}$ の抽出された reify/reflect 関数

もう一つ，shift/reset を用いた実行例を示す．例えば $\underline{\lambda}x.\underline{\langle}\underline{S}k.k\,\underline{@}\,x\underline{\rangle}$ を TDPE で実行することを考える．この式を部分評価する場合，reset の真下に shift が来ているため，shift で束縛されている変数 k には恒等関数が代入される．故に $k\,\underline{@}\,x$ は x へと評価される．よって TDPE 実行結果は上の例と同じ式 $\underline{\lambda}x_1.\underline{S}k_1.\underline{\text{let}}\ g = \underline{\langle}k_1\,\underline{@}\,x_1\underline{\rangle}\ \underline{\text{in}}\ g$ になる筈である．$\underline{\lambda}x.\underline{\langle}\underline{S}k.k\,\underline{@}\,x\underline{\rangle}$ は以下のごとく定義出来る：

```
Example VL_id'2 : VL (arrow base base base base) := fun var =>
  vl_Lam (fun x =>
    tm_Reset (tm_Shift (fun k =>
      tm_App (tm_Val (vl_Var k)) (tm_Val (vl_Var x))))).
```

この式の TDPE 実行結果が以下であり，正しい答えが得られていることが分かる：

```
Eval compute in (reify var _ (soundness_r_VL _ VL_id'2)).
```

```
= nv_LamShift (fun (x1 :  var base) (k1 :  var (arrow base base base base)) =>
                 nc_LetResetApp k1 (nv_Var x1)
                                (fun g :  var base => nc_nv (nv_Var g)))
:  nv var (arrow base base base base)
```

5.3.3　正当性定理の証明

TDPE の正当性定理の証明は，証明に使用する諸々の道具が 2CPS に拡張されているだけで，手順は前節までと同様である．まずは正当性定理の定義に用いる述語，interp_nv と interp_nc を定義する：

$$
\begin{array}{lcl}
\texttt{interp\_nv} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \mathsf{nv}\ (V_s\ b\ \mathcal{V})\ A \to V_s\ b\ \mathcal{V}\ A \to Prop \\
\texttt{interp\_nv}\ b\ \mathcal{V}\ nv\ v & \Longleftrightarrow & \mathsf{soundness\_s\_vl}\ \mathcal{V}\ (\mathsf{vl\_of\_nv}(nv)) = v \\
\texttt{interp\_nc} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \forall r, \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ A \to \\
 & & ((V_s\ b\ \mathcal{V}\ A \to (V_s\ b\ \mathcal{V}\ r \to \mathsf{nat}) \to \mathsf{nat}) \to \\
 & & (V_s\ b\ \mathcal{V}\ r \to \mathsf{nat}) \to \mathsf{nat}) \to Prop \\
\texttt{interp\_nc}\ b\ \mathcal{V}\ r\ nc\ v & \Longleftrightarrow & \mathsf{soundness\_s\_tm}\ \mathcal{V}\ (\mathsf{tm\_of\_nc}(nc)) = v
\end{array}
$$

Λ_{cbv} にて定義したときとの違いは，interp_nc が受け取る v が 2CPS の形をしており，新たに継続が返す型 r も interp_nc が引数として受け取っている点である．

正当性定理に関しては，前節までと同一の定義の形をしている：

定理 26 (Correctness, $\Lambda_{cbv}^{S/R}$)**.**

$$
\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall A, \forall (Vl : \mathsf{VL}(A)), \\
\texttt{interp\_nv}\ b\ \mathcal{V}\ (\texttt{reify}\ (V_s\ b\ \mathcal{V})\ A\ (\mathsf{soundness\_r\_VL}\ (V_s\ b\ \mathcal{V})\ Vl)) \\
\qquad\qquad (\mathsf{soundness\_s\_VL}\ \mathcal{V}\ Vl)
\end{array}
$$

正当性定理の証明に必要となる論理関係 R は，Λ_{cbv} で定義した R を 2CPS に対応する形で拡張

することによって定義される：

$$
\begin{array}{lcl}
\mathsf{R} & : & \forall b, \forall \mathcal{V}, \forall A, V_r\ (V_s\ b\ \mathcal{V})\ A \to V_s\ b\ \mathcal{V}\ A \to * \\
\mathsf{R}\ b\ \mathcal{V}\ \mathsf{base}\ e\ v & \Longleftrightarrow & \texttt{interp\_nv}\ b\ \mathcal{V}\ e\ v \\
\mathsf{R}\ b\ \mathcal{V}\ A/c \to B/d\ e\ v & \Longleftrightarrow & \forall e_1, \forall a_1, \mathsf{R}\ b\ \mathcal{V}\ A\ e_1\ a_1\ \Rightarrow \\
 & & \forall (k_1 : V_r\ (V_s\ b\ \mathcal{V})\ B \to \forall D, (V_r\ (V_s\ b\ \mathcal{V})\ c \to \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ D) \to \\
 & & \qquad \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ D), \\
 & & \forall (k_1' : V_s\ b\ \mathcal{V}\ B \to (V_s\ b\ \mathcal{V}\ c \to (V_s\ b\ \mathcal{V}\ c \to \mathsf{nat}) \to \mathsf{nat}) \to \\
 & & \qquad (V_s\ b\ \mathcal{V}\ c \to \mathsf{nat}) \to \mathsf{nat}), \\
 & & (\forall e_2, \forall a_2, \mathsf{R}\ b\ \mathcal{V}\ B\ e_2\ a_2\ \Rightarrow\ \forall D, \forall k_3, \forall k_3', \\
 & & \ (\forall e_3, \forall a_3, \forall r, \mathsf{R}\ b\ \mathcal{V}\ c\ e_3\ a_3\ \Rightarrow \\
 & & \ \ \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ (k_3\ e_3)\ (\lambda k.\lambda k'.k_3'\ a_3\ (\lambda v.k\ v\ k'))) \Rightarrow \forall r, \\
 & & \ \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ (k_1\ e_2\ D\ k_3) \\
 & & \qquad\qquad (\lambda k.\lambda k'.k_1'\ a_2\ \mathrm{id}\ (\lambda v.k_3'\ v\ (\lambda v'.k\ v'k')))) \Rightarrow \\
 & & \forall D', \\
 & & \forall (k_2 : V_r\ (V_s\ b\ \mathcal{V})\ d \to \mathsf{nc}\ (V_s\ b\ \mathcal{V})\ D', \\
 & & \forall (k_2' : V_s\ b\ \mathcal{V}\ d \to (V_s\ b\ \mathcal{V}\ D' \to \mathsf{nat}) \to \mathsf{nat}), \\
 & & (\forall e_4, \forall a_4, \forall r, \mathsf{R}\ b\ \mathcal{V}\ d\ e_4\ a_4\ \Rightarrow \\
 & & \ \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ (k_2\ e_4)\ (\lambda k.\lambda k'.k_2'\ a_4\ (\lambda v.k\ v\ k'))) \Rightarrow \forall r, \\
 & & \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ (e\ e_1\ k_1\ D'\ k_2) \\
 & & \qquad\qquad (\lambda k.\lambda k'.v\ a_1\ (\lambda x.\lambda f.k_1'\ x\ \mathrm{id}\ f) \\
 & & \qquad\qquad\qquad (\lambda x.k_2'\ x\ (\lambda v.k\ v\ k')))
\end{array}
$$

ここで登場している id は 2CPS における恒等関数を表しており，$\mathrm{id} = \lambda x.\lambda f.f\ x$ である．又，$A/c \to B/d$ のケースにおける式の二行目から六行目が継続，八行目と九行目がメタ継続に対応する式となっている．Λ_{cbv} で定義した R 同様，上の論理関係についても以下の定理が成立する：

定理 27 ($\texttt{reify\_R}/\texttt{reflect\_R}$, $\Lambda_{cbv}^{S/R}$)**.**

($\texttt{reify\_R}$)　$\forall b, \forall \mathcal{V}, \forall A, \forall v, \forall f, \mathsf{R}\ b\ \mathcal{V}\ A\ v\ f\ \Rightarrow\ \texttt{interp\_nv}\ b\ \mathcal{V}\ (\texttt{reify}\ (V_s\ b\ \mathcal{V})\ A\ v)\ f$
($\texttt{reflect\_R}$)　$\forall b, \forall \mathcal{V}, \forall A, \forall x, \forall v, \texttt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(x)\ v\ \Rightarrow\ \mathsf{R}\ b\ \mathcal{V}\ A\ (\texttt{reflect}\ (V_s\ b\ \mathcal{V})\ A\ x)\ v$

この定理を証明する際に使用した諸補題が以下となる．(これら諸補題は，Λ_{cbn}，Λ_{cbv} のケース同様，展開すれば簡単に証明可能である．)

補題 20 ($\texttt{interp\_nc\_nv}$, $\Lambda_{cbv}^{S/R}$)**.**

$$\forall b, \forall \mathcal{V}, \forall e, \forall v, \texttt{interp\_nv}\ b\ \mathcal{V}\ e\ v\ \Rightarrow\ \forall r, \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ \mathsf{nc\_nv}(e)\ (\lambda k.\lambda k'.k\ v\ k')$$

補題 21 ($\texttt{interp\_nv\_LamShift}$, $\Lambda_{cbv}^{S/R}$)**.**

$$
\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall f, \forall v, \\
(\forall x, \forall y, \forall r, \texttt{interp\_nc}\ b\ \mathcal{V}\ r\ (f\ x\ y)\ (\lambda k.\lambda k'.v\ x\ (\lambda x'.\lambda f.y\ x'\ \mathrm{id}\ f)(\lambda x'.k\ x'\ k'))) \Rightarrow \\
\texttt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{lam\_shift}(f)\ v
\end{array}
$$

補題 22 (interp_nv_Var, $\Lambda_{cbv}^{S/R}$).

$$\forall b, \forall \mathcal{V}, \forall v, \texttt{interp\_nv}\ b\ \mathcal{V}\ \mathsf{var}(v)\ v$$

補題 23 (interp_nc_LetResetApp, $\Lambda_{cbv}^{S/R}$).

$$\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall x, \forall x', \texttt{interp_nv}\ b\ \mathcal{V}\ \mathsf{var}(x)\ x'\ \Rightarrow \\
\forall e, \forall e', \texttt{interp_nv}\ b\ \mathcal{V}\ e\ e'\ \Rightarrow\ \forall f, \forall f', \\
(\forall y, \forall y', \forall r, \texttt{interp_nv}\ b\ \mathcal{V}\ \mathsf{var}(y)\ y'\ \Rightarrow\ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (f\ y)\ (\lambda k.\lambda k'.\ f'\ y'\ (\lambda v.k\ v\ k'))) \Rightarrow \\
\forall r, \texttt{interp_nc}\ b\ \mathcal{V}\ r\ \mathsf{let_reset_app}(x, e, f)\ (\lambda k.\lambda k'.\ x'\ e'\ \mathrm{id}\ (\lambda v.\ f'\ v\ (\lambda v'.\ k\ v'\ k')))
\end{array}$$

補題 24 (interp_nc_LetResetLetApp, $\Lambda_{cbv}^{S/R}$).

$$\begin{array}{l}
\forall b, \forall \mathcal{V}, \forall x, \forall x', \texttt{interp_nv}\ b\ \mathcal{V}\ \mathsf{var}(x)\ x'\ \Rightarrow \\
\forall e, \forall e', \texttt{interp_nv}\ b\ \mathcal{V}\ e\ e'\ \Rightarrow \\
\forall f, \forall f', \\
(\forall y, \forall y', \forall r, \texttt{interp_nv}\ b\ \mathcal{V}\ \mathsf{var}(y)\ y'\ \Rightarrow\ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (f\ y)\ (\lambda k.\lambda k'.\ f'\ y'\ (\lambda v.k\ v\ k'))) \Rightarrow \\
\forall g, \forall g', \\
(\forall z, \forall z', \forall r, \texttt{interp_nv}\ b\ \mathcal{V}\ \mathsf{var}(z)\ z'\ \Rightarrow\ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (g\ z)\ (\lambda k.\lambda k'.\ g'\ z'\ (\lambda v.k\ v\ k'))) \Rightarrow \\
\forall r, \texttt{interp_nc}\ b\ \mathcal{V}\ r\ \mathsf{let_reset_let_app}(x, e, f, g)\ (\lambda k.\lambda k'.\ x'\ e'\ f'\ (\lambda x.\ g'\ x\ (\lambda v'.\ k\ v'\ k')))
\end{array}$$

そして補題 25 を使うことによって，補題 26 が証明出来る．

補題 25 (main'_vl/main'_tm, $\Lambda_{cbv}^{S/R}$).

$$\begin{array}{ll}
(\text{main'_vl}) & \forall b, \forall \mathcal{V}, \forall A, \forall v_1, \forall v_2, \\
& \mathsf{related_vl}\ b\ \mathcal{V}\ v_1\ v_2\ \Rightarrow \\
& \mathsf{R}\ b\ \mathcal{V}\ A\ (\mathsf{soundness_r_vl}\ (V_s\ b\ \mathcal{V})\ v_1)\ (\mathsf{soundness_s_vl}\ \mathcal{V}\ v_2) \\
(\text{main'_tm}) & \forall b, \forall \mathcal{V}, \forall B, \forall c, \forall d, \forall t_1, \forall t_2, \\
& \mathsf{related_tm}\ b\ \mathcal{V}\ t_1\ t_2\ \Rightarrow \\
& \forall k_1, \forall k_1', \\
& (\forall e_2, \forall a_2, \mathsf{R}\ b\ \mathcal{V}\ B\ e_2\ a_2\ \Rightarrow\ \forall D, \forall k_3, \forall k_3', \\
& \ (\forall e_3, \forall a_3, \forall r, \mathsf{R}\ b\ \mathcal{V}\ c\ e_3\ a_3\ \Rightarrow \\
& \ \ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (k_3\ e_3)\ (\lambda k.\lambda k'.k_3'\ a_3\ (\lambda v.k\ v\ k'))) \Rightarrow\ \forall r, \\
& \ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (k_1\ e_2\ D\ k_3) \\
& \qquad\qquad (\lambda k.\lambda k'.\ k_1'\ a_2\ \mathrm{id}\ (\lambda v.k_3'\ v\ (\lambda v'.k\ v'k')))) \Rightarrow \\
& \forall D', \forall k_2, \forall k_2', \\
& (\forall e_4, \forall a_4, \forall r, \mathsf{R}\ b\ \mathcal{V}\ d\ e_4\ a_4\ \Rightarrow \\
& \ \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (k_2\ e_4)\ (\lambda k.\lambda k'.\ k_2'\ a_4\ (\lambda v.k\ v\ k'))) \Rightarrow\ \forall r, \\
& \texttt{interp_nc}\ b\ \mathcal{V}\ r\ (\mathsf{soundness_r_tm}\ (V_s\ b\ \mathcal{V})\ t_1\ k_1\ D'\ k_2) \\
& \qquad\qquad (\lambda k.\lambda k'.\mathsf{soundness_s_tm}\ \mathcal{V}\ t_2\ (\lambda x.\lambda f.k_1'\ x\ \mathrm{id}\ f) \\
& \qquad\qquad\qquad\qquad (\lambda x.k_2'\ x\ (\lambda v.k\ v\ k')))
\end{array}$$

補題 26 ($\texttt{main}$, $\Lambda_{cbv}^{S/R}$)**.**

$$\forall b, \forall \mathcal{V}, \forall A, \forall (Vl : \mathsf{VL}(A)),$$
$$\mathsf{R}\ b\ \mathcal{V}\ A\ (\mathsf{soundness\_r\_VL}\ (V_s\ b\ \mathcal{V})\ Vl)\ (\mathsf{soundness\_s\_VL}\ \mathcal{V}\ Vl)$$

補題 25 で使用している related_vl と related_tm は，前節の定義に shift(_) と reset(_) のケースを追加したのみで，後は全く同じ定義となっている：

$$\begin{array}{lcl}
\mathsf{related_vl} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \\
 & & \mathsf{vl}\ (V_r\ (V_s\ b\ \mathcal{V}))\ A \to \mathsf{vl}\ (V_s\ b\ \mathcal{V})\ A \to * \\
\mathsf{related_vl}\ b\ \mathcal{V}\ \mathsf{var}(v)\ \mathsf{var}(v') & \Longleftrightarrow & \mathsf{R}\ b\ \mathcal{V}\ A\ v\ v' \\
\mathsf{related_vl}\ b\ \mathcal{V}\ \mathsf{lam}(t)\ \mathsf{lam}(t') & \Longleftrightarrow & \forall v, \forall v', \mathsf{R}\ b\ \mathcal{V}\ A\ v\ v' \Rightarrow \mathsf{related_tm}\ b\ \mathcal{V}\ (t\ v)\ (t'\ v') \\
 & & \\
\mathsf{related_tm} & : & \forall b, \forall \mathcal{V}, \forall \{A\}, \\
 & & \mathsf{tm}\ (V_r\ (V_s\ b\ \mathcal{V}))\ A \to \mathsf{tm}\ (V_s\ b\ \mathcal{V})\ A \to * \\
\mathsf{related_tm}\ b\ \mathcal{V}\ \mathsf{val}(v)\ \mathsf{val}(v') & \Longleftrightarrow & \mathsf{related_vl}\ b\ \mathcal{V}\ v\ v' \\
\mathsf{related_tm}\ b\ \mathcal{V}\ \mathsf{app}(t_1, t_2)\ \mathsf{app}(t_1', t_2') & \Longleftrightarrow & \mathsf{related_tm}\ b\ \mathcal{V}\ t_1\ t_1' \wedge \mathsf{related_tm}\ b\ \mathcal{V}\ t_2\ t_2' \\
\mathsf{related_tm}\ b\ \mathcal{V}\ \mathsf{shift}(t)\ \mathsf{shift}(t') & \Longleftrightarrow & \forall v, \forall v', \mathsf{R}\ b\ \mathcal{V}\ (A/c \to c/c)\ v\ v' \Rightarrow \\
 & & \mathsf{related_tm}\ b\ \mathcal{V}\ (t\ v)\ (t'\ v') \\
\mathsf{related_tm}\ b\ \mathcal{V}\ \mathsf{reset}(t)\ \mathsf{reset}(t') & \Longleftrightarrow & \mathsf{related_tm}\ b\ \mathcal{V}\ t\ t'
\end{array}$$

前節までと同様，述語 related_vl と related_tm に関する以下の性質は Coq で証明出来ないので，公理として定義する：

公理 4 ($\texttt{Vr\_related/T\_related}$, $\Lambda_{cbv}^{S/R}$)**.**

$$\begin{array}{ll}
(\mathrm{Vr_related}) & \forall b, \forall \mathcal{V}, \forall Vl, \mathsf{related_vl}\ b\ \mathcal{V}\ (Vl\ (V_r\ (V_s\ b\ \mathcal{V})))\ (Vl\ (V_s\ b\ \mathcal{V})) \\
(\mathrm{T_related}) & \forall b, \forall \mathcal{V}, \forall Tm, \mathsf{related_tm}\ b\ \mathcal{V}\ (Tm\ (V_r\ (V_s\ b\ \mathcal{V})))\ (Tm\ (V_s\ b\ \mathcal{V}))
\end{array}$$

5.4　関連研究

本章で示した（Λ_{cbn} と Λ_{cbv} における）correctness 定理の証明は，Filinski[15, 16] にほぼ対応したものとなっている．Filinski は変数名の生成を明示的に行う為に，変数名を追えるよう Kripke 意味論を使用しているが，本章で行った定式化においては，変数名の生成はメタ言語である Coq が全て自動的に行ってくれる．但し，Filinski は TDPE 実行時に無限ループが発生するのを許しているが，本章の定式化では定義可能な term は全て型が付いているため，無限ループに関しては本研究はサポートしていない．

Coquand [10] は証明エディター ALF を用いて soundness と completeness の証明を定式化し，そして completeness が TDPE と Curry Howard 同型対応にあることを指摘した．その後に，Ilik

[18, 19, 20] は CBN，CBV，限定継続命令付きの CBV の TDPE を定式化した．4 章でも述べたように，Coquand と Ilik は共に α 同値問題を避ける為に Kripke 意味論を使用している．本章における定式化の前半部分は 4 章と同じ strategy を用いているが，Kripke モデルでなく PHOAS を使用しているおかげで，よりシンプルな定式化が実現出来ている．証明から抽出されたプログラムは非常にシンプルであり，Danvy の TDPE と全く対応したものとなっている．この proof term のシンプルさは，proof term が複雑で解釈がしばしば困難であった Ilik [18, 19, 20] の先行研究や，本論文 2 章で抽出した評価器プログラムとは対称的である．

Chlipala[9] は PHOAS を用いた定式化において，本研究の related_term に関する性質 1 のように，term の構造に関する性質については Coq において証明不可能であることを指摘し，仕方なくその部分に関しては公理として Coq に載せている．本研究もそれに倣い，related_term についての性質に関しては手動で証明を行い，Coq で定式化する際には公理として扱った．

5.5　まとめと今後の課題

本章では，CBN と CBV の TDPE における Filinski [15] の正当性定理の証明を Coq で定式化し，それを拡張する形で Tsushima ら [26] の shift/reset 付き TDPE の正当性定理を Coq で証明した．これにより，無限ループが絶対に起こらず，かつ入力プログラムと意味の等価な正規形を計算することが保証された TDPE を抽出出来たことになる．又，TDPE 抽出に用いた standard semantics は，4 章で定義した Kripke 意味論における論理関係と対応関係にある定義となっている．故に本章の completeness 定理と soundness 定理はそれぞれ，4 章の completeness 定理，soundness 定理と非常に似た定義となっている．

又，本章では Chlipala [8, 9] の定式化を参考に，PHOAS を用いて型システムを定式化することによって α 同値問題を回避した．この手法は 2 章の Locally Nameless 手法や 4 章の Kripke モデルを用いるよりも簡潔な定式化が実現出来ている．例えば Locally Nameless 手法を使う場合は，それを実現するために Aydemir ら [3] のライブラリを使う必要があったが，PHAOS を用いる場合にはその様なライブラリを一切使用する必要がない．そして Kripke モデルを用いる場合には常に型情報として型環境を持ち歩く必要があったが，PHOAS ではその必要もない．又，Chlipala の先行研究によって Coq での PHOAS の定式化が行われているものの，PHOAS を使った研究は未だ少なく，本章は PHOAS を使った定式化方法の一つのケーススタディともなっていると考えら

れる．

但し，本章においては term（或は value）に関する論理関係 related_tm（related_vl）の性質の証明は Coq で定式化することが出来ていない．これは higher-order を用いて term や value を定義していることに起因している．つまり term らにおける λ 抽象を定義する際にはメタ言語 Coq の無名関数を使って定義している．故に Coq で証明する際，term らの構造を調べたくとも，Coq の関数の中について場合分けすることが出来ない．それがこの性質の証明を Coq で定式化することが出来ない理由である．

Chlipala [8] は先行研究にて，本研究が使用した related_tm と類似の性質をほぼ自明に成立するものとして扱っている．我々が用いた性質は，Chlipala が用いた性質の部分集合となっているため，Chlipala の性質が成り立つならば，本章における性質も成立してしかるべきである．しかし定義を見る限り，自明に成立する様には思われず，我々は未だ彼の意図を理解出来ていない．

5.1.5 節で，我々は性質について手動の証明を行っているが，この証明で行っている term T の構造に関する場合分けの仕方が（そうなるだろうとは考えられるものの,）正しいか否かに関しては未だ示せていない．故に厳密に言えばこの証明が本当に正しいのかは示せていないのである．この問題を解決するのが今後の課題となっている．

謝辞

本論文の作成過程において，指導教員の浅井健一先生には研究テーマ及び研究過程の細部に至まで懇切丁寧な指導を頂きました．ここに深く感謝申し上げます．又，私がこの研究段階まで進むことが出来ましたのは，本学科で受けた教育とそれをサポートする環境によるものと思っております．ご指導下さいました副指導教員の戸次大介先生及び本学科の諸先生方に深く感謝申し上げます.

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 Y. Kameyama “Polymorphic Delimited Continuations,” *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 239–254 (November 2007).

[3] Aydemir, B., A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich: “Engineering Formal Metatheory,” *POPL'08*, pp. 3–15 (January 2008).

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

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

[6] Biernacka, M., and D. Biernacki “Context-based proofs of termination for typed delimited-control operators,” *11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (September 2009).

[7] Biernacka, M., O. Danvy, and K. Støvring “Program extraction from proofs of weak head normalization,” *Electronic Notes in Theoretical Computer Science 155*, pp. 169–189 (May 2006).

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

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

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

[11] Danvy, O., and A. Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

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

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

[14] Danvy, O. "Type-Directed Partial Evaluation," In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann editors, *Partial Evaluation, Practice and Theory (LNCS 1706)*, pp. 367–411 (1999).

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

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

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

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

[19] 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).

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

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

[22] Kameyama, Y., and M. Hasegawa “A Sound and Complete Axiomatization of Delimited Continuations,” *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 177–188 (August 2003).

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

[24] Masuko, M., and K. Asai “Caml Light + shift/reset = Caml Shift”, *Theory and Practice of Delimited Continuations (TPDC 2011)*, pp. 33–46 (May 2011).

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

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

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

[28] Yang, Z. “Encoding Types in ML-like Languages,” *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pp. 289–300 (September 1998).

付 録A　Locally Nameless手法を用いたshift/reset付き評価器の証明の詳細

定理 2. $e \rightsquigarrow^* e'$ かつ $R_T(e)$ ならば，$R_T(e')$.

Proof. T に関する帰納法を用いる．

($T = tb$ のとき) R の定義により，$R_{tb}(e) = N(e)$，$R_{tb}(e') = N(e')$ と変形出来る．示したいのは $N(e')$ である．$N(e)$ の成立により，$e \rightsquigarrow^* v$ となる value v が存在する．故に仮定 $e \rightsquigarrow^* e'$ と補題 2 により，$e' \rightsquigarrow^* v$ が成立．よって $N(e')$ が示せた．

($T = tf$ のとき) $T = tb$ の場合と同様.

($T = (\sigma/\alpha \to \tau/\beta)$ のとき) 仮定 $R_{(\sigma/\alpha \to \tau/\beta)}(e)$ により，以下の 2 式が成立.

(i) $N(e)$.

(ii) $R_\sigma(v)$ を満たす任意の value v と，$\lambda.K \models \tau \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e\,v)\rangle)$ が成立.

$R_{(\sigma/\alpha \to \tau/\beta)}(e')$ を示すには，以下の 2 式の成立を示せば良い.

(1) $N(e')$.

(2) $R_\sigma(v)$ を満たす任意の value v と，$\lambda.K \models \tau \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e'\,v)\rangle)$ が成立.

(1) は $T = tb$ の場合と同様にして成立．又，仮定 $e \rightsquigarrow^* e'$ と簡約規則により，$\langle(\lambda.K)(e\,v)\rangle \rightsquigarrow^* \langle(\lambda.K)(e'\,v)\rangle$．よって (ii) と帰納法の仮定により (2) が成立.

□

定理 3. $e \rightsquigarrow^* e'$ かつ $R_T(e')$ ならば，$R_T(e)$.

Proof. T に関する帰納法を用いる．

($T = tb$ のとき) $R_{tb}(e) = N(e)$ と変形出来る．仮定により，$e' \rightsquigarrow^* v$ となる value v が存在する．故に仮定 $e \rightsquigarrow^* e'$ と補題 3 により，$e \rightsquigarrow^* v$ が成立．よって $N(e)$ が示せた．

($T = tf$ のとき) $T = tb$ のときと同様にして示せる．

($T = (\sigma/\alpha \to \tau/\beta)$ のとき) 仮定により以下の 2 式が成立．

(i) $N(e')$.

(ii) $R_\sigma(v)$ を満たす任意の value v と，$\lambda.K \models \tau \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e'\,v)\rangle)$ が成立.

$R_{(\sigma/\alpha\to\tau/\beta)}(e)$ を示すには，以下の 2 式の成立を示せば良い.

(1) $N(e)$.

(2) $R_\sigma(v)$ を満たす任意の value v と，$\lambda.K \models \tau \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e\,v)\rangle)$ が成立.

(1) は $T = tb$ のときと同様にして示せる．又，仮定 $e \rightsquigarrow^* e'$ と簡約規則により，$\langle(\lambda.K)(e\,v)\rangle \rightsquigarrow^* \langle(\lambda.K)(e'\,v)\rangle$．よって (ii) と帰納法の仮定により (2) が成立する．

□

定理 4. $\Gamma = x_1 : T_1, x_2 : T_2, ..., x_n : T_n$ とおく．又，以下に登場する $v_1 ..., v_n$ は全て，$T'_1 \preceq T_1, ..., T'_n \preceq T_n$ を満たす任意の型 $T'_1, ..., T'_n$ についてそれぞれ $R_{T'_1}(v_1), ..., R_{T'_n}(v_n)$ を満たす *value* であるとする.

- $\Gamma;\, \alpha \vdash e\{0 \mapsto y_0\}...\{m \mapsto y_m\} : T;\, \beta$ が成立するとき，上の条件を満たすような任意の $v_1 ..., v_n$ と，$\lambda.K \models T \rightharpoonup \alpha$ を満たす任意の $\lambda.K$ について，$R_\beta(\langle(\lambda.K)(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$ が成立する.

- $\Gamma \vdash_p e\{0 \mapsto y_0\}...\{m \mapsto y_m\} : T$ が成立するとき，上の条件を満たすような任意の $v_1 ..., v_n$ について，$R_T(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])$ が成立する．

Proof. e に関する帰納法を用いる．

($e = i$**（束縛変数）のとき**) $i < 0, m < i$ の場合，$i\{0 \mapsto y_0\}...\{m \mapsto y_m\} = i$ となり，仮定の型規則が成立せず矛盾する．故に $0 \leq i \leq m$ であり，$i\{0 \mapsto y_0\}...\{m \mapsto y_m\} = y_i$ となる．以下，$\Gamma;\ \alpha \vdash y_i : T;\ \beta$ が成立する場合と $\Gamma \vdash_p y_i : T$ が成立する場合のそれぞれに関して証明を行う．

(i) $\Gamma;\ \alpha \vdash y_i : T;\ \beta$ のとき：型付け規則の定義により，以下の推論木が成り立つはずである．（故に $\alpha = \beta$，$T = M^{U_s}$ となる．）

$$\frac{ok\ \Gamma \quad (y_i : M) \in \Gamma}{\Gamma \vdash_p y_i : M^{U_s}}\ (var)$$
$$\frac{}{\Gamma;\ \alpha \vdash y_i : M^{U_s};\ \alpha}\ (exp)$$

示したいのは $R_\beta(\langle(\lambda.K)(y_i[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$ である．上の推論木の成立により，$(y_i : M) \in \Gamma$ であるから，$y_i = x_j$ となる $x_j : M$（$1 \leq j \leq n$）が存在するはずであり，$y_i[x_1 \mapsto v_1]...[x_n \mapsto v_n] = v_j$ となる．故に $R_\beta(\langle(\lambda.K)(y_i[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle) = R_\alpha(\langle(\lambda.K)(v_j)\rangle)$.

仮定から $\forall M' \preceq M$ について $R_{M'}(v_j)$ が成立する．推論木により $M^{U_s} \preceq T_j$ であるから，$R^{U_s}_M(v_j)$ が成立．故に $\lambda.K$ の仮定により $R_\alpha(\langle(\lambda.K)v_j\rangle)$ が成立．

(ii) $\Gamma \vdash_p y_i : T$ のとき：以下の推論木が成立する．

$$\frac{ok\ \Gamma \quad (y_i : M) \in \Gamma}{\Gamma \vdash_p y_i : M^{U_s}}\ (var)$$

$T = M^{U_s}$ とおく．上の推論木により $y_i : M \in \Gamma$ であるから，$\forall M' \preceq M$ について $R_{M'}(v_j)$ を満たすような $y_i[x_1 \mapsto v_1]...[x_n \mapsto v_n] = v_j$（$0 \leq j \leq n$）が存在するはずである．$M^{U_s} \preceq M$ であるから v_j の仮定により $R_{M^{U_s}}(v_j)$ が成立する．

($e = x$**（自由変数）のとき**) x は自由変数なので，$x\{0 \mapsto y_0\}...\{m \mapsto y_m\} = x$ となる．

(i) $\Gamma;\ \alpha \vdash x : T;\ \beta$ のとき：型付け規則により，以下の推論木が成立．故に $\alpha = \beta$，$T = M^{U_s}$.

$$\frac{ok\ \Gamma \quad (x : M) \in \Gamma}{\Gamma \vdash_p x : M^{U_s}}\ (var)$$
$$\frac{}{\Gamma;\ \alpha \vdash x : M^{U_s};\ \alpha}\ (exp)$$

$x : M \in \Gamma$ であるから，$x[x_1 \mapsto v_1]...[x_n \mapsto v_n] = v_i$ となる value v_i が存在し，仮定により $\forall M' \preceq M$ について $R_{M'}(v_i)$ が成立する．$\lambda.K$ の仮定により，$R_\alpha(\langle(\lambda.K)\, v\rangle)$ が成立.

(ii) $\Gamma \vdash_p x : T$ のとき：型付け規則の定義により，$T = M^{U_s}$ とおける．又，$ok\ \Gamma$ と $(x : M) \in \Gamma$ が成立．$x : M \in \Gamma$ であるから，$\forall M' \preceq M$ について $R_{M'}(v_i)$ を満たすような $x[x_1 \mapsto v_1]...[x_n \mapsto v_n] = v_i$（$0 \leq i \leq n$）が存在するはずである．$M^{U_s} \preceq M$ であるから v_i の仮定により $R_{M^{U_s}}(v_i)$ が成立する.

($e = \lambda.e$ のとき) $(\lambda.e)\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]$

$= \lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])$.

(i) $\Gamma;\, \alpha \vdash \lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}) : T;\, \beta$ のとき：以下の推論木が成立し，$\alpha = \beta$ となる．又，$T = (\sigma/\alpha' \to \tau/\beta')$ とおける.

$$\dfrac{\dfrac{\forall x \notin L.\ (\Gamma, x : \sigma;\, \alpha' \vdash (e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\})^x : \tau;\, \beta')}{\Gamma \vdash_p \lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}) : (\sigma/\alpha' \to \tau/\beta')}\,(fun)}{\Gamma;\, \alpha \vdash \lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}) : (\sigma/\alpha' \to \tau/\beta');\, \alpha}\,(exp)$$

示したいのは，(A) $R_\alpha(\langle(\lambda.K)(\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$ である．仮定により $\lambda.K \models (\sigma/\alpha' \to \tau/\beta') \rightharpoonup \alpha$ であるから，(A) を示すには，
$R_{(\sigma/\alpha' \to \tau/\beta')}(\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))$ の成立，即ち以下の 2 式が成り立つことを示せば良い.

(1) $N(\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))$.

(2) $\forall v.\, R_\sigma(v)$ と $\forall \lambda.K' \models \tau \rightharpoonup \alpha'$ について，
$R_{\beta'}(\langle(\lambda.K')((\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))\, v)\rangle)$.

(1) は明らか．以下，(2) を示す.

$$\begin{array}{rl} & \langle(\lambda.K')((\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))\, v)\rangle \\ \rightsquigarrow & \langle(\lambda.K')(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]\{0 \mapsto v\})\rangle \\ = & \langle(\lambda.K')(e\{0 \mapsto x'\}\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n][x' \mapsto v])\rangle \end{array}$$

と変形することが出来る.（ただし，$(x' : \sigma \notin \Gamma)$.）さて，型規則の成立により，$\sigma$ は polymorphic type ではないことが分かっている．故に $\sigma \preceq \sigma$ が成立．故に，上の推論木における L を Γ とおけば，帰納法の仮定により，以下の式が成立する.

$R_{\beta'}(\langle(\lambda.K')(e\{0\mapsto x'\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x'\mapsto v]\rangle\rangle)$

故に系 1 により (2) が成立.

(ii) $\Gamma\vdash_p \lambda.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}) : T$ のとき：型付け規則の定義により，$T=(\sigma/\alpha'\to\tau/\beta')$ とおけ，$\forall x\notin L.\ (\Gamma, x:\sigma;\ \alpha'\vdash(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\})^x:\tau;\ \beta')$ が成立する．又，型規則の成立により，σ は単相型である.（$\sigma\preceq\sigma$.）よって帰納法の仮定により，以下の式が成立.

(A) $\forall v.\ R_\sigma(v)$ と $\forall\lambda.K\models\tau\rightharpoonup\alpha'$ に対して，

$R_{\beta'}(\langle(\lambda.K)(e\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v]\rangle\rangle)$.

今示したいのは $R_{(\sigma/\alpha'\to\tau/\beta')}(\lambda.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))$ である．この式を示すには以下の 2 式を示せば良い.

(1) $N(\lambda.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))$.

(2) $\forall v.\ R_\sigma(v)$ と $\forall\lambda.K\models\tau\rightharpoonup\alpha'$ について，

$R_{\beta'}(\langle(\lambda.K)((e\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v])\,v)\rangle\rangle)$.

(1) の成立は明らかである．以下で (2) の成立を示す.

$$\begin{array}{rl} & \langle(\lambda.K)((\lambda.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))\,v)\rangle \\ \rightsquigarrow & \langle(\lambda.K)(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\{0\mapsto v\})\rangle \\ = & \langle(\lambda.K)(e\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v])\rangle \end{array}$$

と変形出来（ただし $(x:\sigma\notin\Gamma)$），故に (A) に登場する x の制限に使われる L を Γ とおくことにより，(A) と系 1 を適用出来，結果 (2) が成立する.

($e=e_1\,e_2$ のとき)

$$\begin{array}{rl} & (e_1\,e_2)\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n] \\ = & (e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]) \\ & (e_2\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]) \end{array}$$

故に仮定の型規則より，$\Gamma;\gamma\vdash e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}:(\sigma/\alpha\to T/\delta);\beta$ と
$\Gamma;\delta\vdash e_2\{0\mapsto y_0\}...\{m\mapsto y_m\}:\sigma;\gamma$ が成立する．よって帰納法の仮定により，以下の 2 式が成立.

(A) $\forall\lambda.K_1 \models (\sigma/\alpha \to T/\delta) \rightharpoonup \gamma$ について，

$R_\beta(\langle(\lambda.K_1)(e_1\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$.

(B) $\forall\lambda.K_2 \models \sigma \rightharpoonup \delta$ について，

$R_\gamma(\langle(\lambda.K_2)(e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$.

以下で，$\forall v_1'.\ R_{(\sigma/\alpha\to T/\delta)}(v_1')$ において，$\lambda.(\lambda.K)(v_1'\ 0) \models \sigma \rightharpoonup \delta$ が成立することを示す．One-step の簡約規則により，$R_\sigma(v_2')$ を満たす任意の value v_2' について，以下の簡約が可能である．

$$\langle(\lambda.(\lambda.K)(v_1'\ 0))\, v_2'\rangle \rightsquigarrow \langle(\lambda.K)(v_1'\ v_2')\rangle$$

仮定により，$\lambda.K \models T \rightharpoonup \alpha$，$R_{(\sigma/\alpha\to T/\delta)}(v_1')$，$R_\sigma(v_2')$ が成立している．よって $R_{(\sigma/\alpha\to T/\delta)}(v_1')$ の定義により，$R_\delta(\langle(\lambda.K)(v_1'\ v_2')\rangle)$ が成立．故に系 1 により $\langle(\lambda.(\lambda.K)(v_1'\ 0))\, v_2'\rangle$ が導ける．よって $\lambda.(\lambda.K)(v_1'\ 0) \models \sigma \rightharpoonup \delta$ が示せた．この式を (B) に適用することにより，

$R_\gamma((\lambda.(\lambda.K)(v_1'\ 0))(e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))$

が得られる．よって公理 1 により，

$R_\gamma((\lambda.K)(v_1'\,(e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])))$

が成立．又，

$$\begin{array}{rl} & \langle(\lambda.(\lambda.K)(0\ (e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])))\, v_1'\rangle \\ \rightsquigarrow & \langle(\lambda.K)(v_1'\,(e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))\rangle \end{array}$$

と簡約出来るので，系 1 により，

$R_\gamma(\langle(\lambda.(\lambda.K)(0\ (e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])))\, v_1'\rangle)$

が成立する．故に，

$\lambda.(\lambda.K)(0\ (e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])) \models (\sigma/\alpha \to T/\delta) \rightharpoonup \gamma$.

この式を (A) に適用することにより，

$R_\beta(\langle(\lambda.(\lambda.K)(0\ (e_2\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))))$

$(e_1\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])\rangle)$

が成立し，故に公理 1 により，

$$R_\beta(\langle(\lambda.K)((e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])$$
$$(e_2\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))\rangle)$$

が得られる．

($e = let\ e_1\ e_2$ のとき) $(let\ e_1\ e_2)\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]$

$$= let\ (e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])$$
$$(e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])$$

と変形出来，故に仮定の型規則により，以下の 2 式が成立．

$$\forall U_x.\,(\Gamma\vdash_p e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n] : M^{U_s})$$
$$\forall x\notin L.(\Gamma, x:M;\,\alpha\vdash (e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])^x : T;\,\beta)$$

よって帰納法の仮定により，以下の 2 式が成立．

(A) $\forall U_s.\ R_{M^{U_s}}(e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])$.

(B) $\forall M'\preceq M$ について $R_{M'}(v')$ を満たす任意の value v' と $\forall\lambda.K\models T\rightharpoonup\alpha$ に対して，
$R_\beta(\langle(\lambda.K)(e_2\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v'])\rangle)$.

(A) と定理 1 により，$e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rightsquigarrow^* v$ となる value v が存在し，定理 2 により $R_{M^{U_s}}(v)$ がいえる．$\forall U_s$ について $M^{U_s}\preceq M$ であるから，(B) により，

$$R_\beta(\langle(\lambda.K)(e_2\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v])\rangle)$$

が成立する．又，

$$\begin{array}{rl} & let\ (e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]) \\ & \quad (e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]) \\ \rightsquigarrow^* & let\ v\ (e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]) \\ \rightsquigarrow & e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\{0\mapsto v\} \\ = & e_2\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto v] \end{array}$$

と変形することが出来る．（ただし $x\notin\Gamma$.）故に定理 3 により，

$$R_\beta(\langle(\lambda.K)(let\ (e_1\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])$$
$$(e_2\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))\rangle)$$

が得られる．

($e = S.e$ のとき) $(S.e)\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]$

$= S.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])$.

仮定により，以下の型規則が成立.

$\forall x \notin L.(\Gamma, x : \forall.(T/0 \to \alpha/0);\, \sigma \vdash (e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\})^x : \sigma;\, \beta)$.

故に帰納法の仮定により，以下の式が成立.

(A) $\forall M \preceq \forall.(T/0 \to \alpha/0)$ について $R_M(v)$ を満たすような任意の value v と，$\forall \lambda.K' \models \sigma \rightharpoonup \sigma$ に対して，
$R_\beta(\langle(\lambda.K')(e\{0 \mapsto x\}\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n][x \mapsto v])\rangle)$.

$R_\beta(\langle(\lambda.(e\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))(\lambda.\langle(\lambda.K)\,0\rangle)\rangle)$ であることを示す.

まず，$\lambda.0 \models \sigma \rightharpoonup \sigma$ を示す．$R_\sigma(v')$ を満たす任意の value v' について，

$\langle(\lambda.0)\,v'\rangle \rightsquigarrow \langle v'\rangle \rightsquigarrow v'$

が成り立つ．故に系 1 により，$R_\sigma(\langle(\lambda.0)\,v'\rangle)$ である．よって $\lambda.0 \models \sigma \rightharpoonup \sigma$ が示せた．故に (A) と公理 1 により，

(B) $\forall M \preceq \forall.(T/0 \to \alpha/0)$ について $R_M(v)$ を満たすような任意の value v に対して，
$R_\beta(\langle e\{0 \mapsto x\}\{1 \mapsto y_0\}...\{m+1 \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n][x \mapsto v]\rangle)$.

次に，$\forall M \preceq \forall.(T/0 \to \alpha/0)$ について，$R_M(\lambda.\langle(\lambda.K)\,0\rangle)$ が成立することを示す．任意の単相型 τ を一つ用意し，$B = (T/\tau \to \alpha/\tau)$ とおく．$R_B(\lambda.\langle(\lambda.K)\,0\rangle)$ が成り立つことを示すには，以下の 2 式を示せば良い.

(1) $N(\lambda.\langle(\lambda.K)\,0\rangle)$.

(2) $\forall v.\ R_T(v)$ と $\lambda.K\text{"} \models \alpha \rightharpoonup \tau$ について，$R_\tau(\langle(\lambda.K\text{"})((\lambda.\langle(\lambda.K)\,0\rangle)\,v)\rangle)$.

(1) が成立するのは明らか．又，$(\lambda.\langle(\lambda.K)\,0\rangle)\,v \rightsquigarrow \langle(\lambda.K)\,v\rangle$ かつ $\lambda.K \models T \rightharpoonup \alpha$ であるので，$R_\alpha(\langle(\lambda.K)\,v\rangle)$ が成立．故に，$\lambda.K\text{"}$ の定義と定理 1，定理 3 により，$R_\tau(\langle(\lambda.K\text{"})\langle(\lambda.K)\,v\rangle\rangle)$ が成立．系 1 により (2) が得られる．任意の単相型 τ について (1) と (2) が成立．よって $R_B(\lambda.\langle(\lambda.K)\,0\rangle)$ が示せた．ここで，

$$\begin{array}{rl} & \langle(\lambda.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))(\lambda.\langle(\lambda.K)\,0\rangle)\rangle \\ \rightsquigarrow & e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\{0\mapsto\lambda.\langle(\lambda.K)\,0\rangle\} \\ = & e\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto\lambda.\langle(\lambda.K)\,0\rangle] \end{array}$$

と変形出来る.（ただし $x\notin\Gamma$，$L=\Gamma$.）$R_B(\lambda.\langle(\lambda.K)\,0\rangle)$ であるから，(B) により，

$R_\beta(e\{0\mapsto x\}\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n][x\mapsto\lambda.\langle(\lambda.K)\,0\rangle])$

が成立．よって公理 2 により，

$R_\beta(\langle(\lambda.K)(S.(e\{1\mapsto y_0\}...\{m+1\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]))\rangle)$ が得られる.

($e=\langle e\rangle$ のとき) $\langle e\rangle\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]$

$=\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle$.

(i) $\Gamma;\,\alpha\vdash\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle : T;\,\beta$ のとき：以下の推論木が成立する.（故に $\beta=\alpha$ となる.）

$$\dfrac{\dfrac{\Gamma;\,\sigma\vdash e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]:\sigma;\,T}{\Gamma\vdash_p\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle:T}\,(reset)}{\Gamma;\,\alpha\vdash\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle:T;\,\beta}\,(exp)$$

帰納法の仮定により，$R_T(\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle)$ が成立．よって定理 1 と定理 2 により，$\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle\rightsquigarrow^* v$ かつ $R_T(v)$ を満たす value v が存在するはずである．$\lambda.K\models T\rightharpoonup\alpha$ により，$R_\alpha(\langle(\lambda.K)\,v\rangle)$ が成立する．よって定理 3 により，

$R_\alpha(\langle(\lambda.K)\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle\rangle)$ が得られる.

(ii) $\Gamma\vdash_p\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle:T$ のとき：以下の推論木が成立.

$$\dfrac{\Gamma;\,\sigma\vdash e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]:\sigma;\,T}{\Gamma\vdash_p\langle e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n]\rangle:T}\,(reset)$$

よって帰納法の仮定により，以下の式が成立する.

(A) $\forall\lambda\models\sigma\rightharpoonup\sigma$ について，

$R_T(\langle(\lambda.K)(e\{0\mapsto y_0\}...\{m\mapsto y_m\}[x_1\mapsto v_1]...[x_n\mapsto v_n])\rangle)$.

$e = S.e$ のケースで示したように，$\lambda.0 \models \sigma \rightharpoonup \sigma$ であり，この式を (A) に適用出来，
$R_T((\lambda.0)(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n]))$ がいえる．よって公理 1 により，
$R_T(e\{0 \mapsto y_0\}...\{m \mapsto y_m\}[x_1 \mapsto v_1]...[x_n \mapsto v_n])$ が得られる．

□

付 録B　一般のde Bruijn index termへの変換プログラム

4章のde Bruijn index termを一般のde Bruijn index termへと変換する（Coq上の）normalize関数は以下の如くに定義出来る．

```
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.
```

又，4章における normal form, neutral term についても，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 $\mathsf{wkn}(\_)$（複数の $\mathsf{wkn}(\_)$ が入れ子になっていても良い）が $\mathsf{app}(\_,\_)$ にかかっていた場合にも，

$\mathsf{wkn}(\_)$ を中に潜らせれば最終的に $\mathsf{app}(\_, \_)$ となるため，neutral term になっていることが確認できる．