限定継続命令をもつ依存型付き言語の設計

叢悠悠 [1], 浅井健一 [1]

[1] お茶の水女子大学

{so.yuyu, asai}@is.ocha.ac.jp

概 要　Coq や Agda などの定理証明支援系では、依存型を使ってプログラムの性質を正確に記述することができる。一方、依存型をもつ言語の多くは副作用を許さない。実際、依存型付きの言語で副作用を無制限に許すと、その言語が論理体系として矛盾することが知られている。そこで、適切な制約を設けながら、依存型と副作用を組み合わせる試みがなされてきた。本研究では、限定継続命令 `shift`/`reset` をもつ依存型付き言語を考える。`shift`/`reset` はさまざまな effect の模倣を可能にするが、その応用例や型システムは単純型や多相型付きの言語で考えられてきた。本稿では、`shift`/`reset` に対する依存型システムを設計し、その健全性を示す。また、`shift`/`reset` を含むプログラムから、これらを含まないものへの CPS 変換を定義する。さらに、`shift`/`reset` を使ったプログラムが論理的にどのような意味を持つかを考察する。

1　はじめに

依存型はプログラマにとって欠かせない要素の 1 つとして認識されはじめている。型はプログラムの信頼性を高めるが、特に依存型を使うと、型の中に項レベルの計算を含ませることによって、プログラムの性質をより正確に記述することができる。このことから、Coq [30] や Agda [24] といった依存型付き言語は、プログラム検証や定理証明などの目的で広く使われている。

依存型付き言語の多くは pure な項、すなわち、副作用を起こさない項のみを許す。副作用の欠如は、こうした言語を実用的な目的で使用する妨げとなるが、一方で、依存型付き言語を 1 つの論理体系として見たとき、この制約は妥当であるようにも思える。たとえば、$\mathsf{let}\ x = \mathsf{get}\ ()\ \mathsf{in}\ e$（$\mathsf{get}\ ()$ は状態の値を返す計算）という項の中で、e の型が x に依存するとしよう。すると、この項の型は、実行時の状態の値に依存する。本来、型はプログラムの性質を静的な情報としてユーザに提供するものであるため、実行時の環境という動的な情報に左右されるべきではないが、依存型付き言語の中で副作用を無制限に許すと、型が静的に定まらなくなり、その言語を定理証明のツールとして使うことが難しくなってしまう。このような背景から、依存型と副作用をともにもちつつ、論理として意味をなす体系を築くための研究が行われてきた。その結果、依存型とプリミティブな effect をもつ言語 F* [28] や、より一般的な effect を許す依存型付きの計算体系 [1, 32] などが生まれた。これらはいずれも、型に現れる項を pure なものに制限するという方針をとっている。

本研究では、限定継続命令 `shift`/`reset` [13] をもつ依存型付き言語を考える。`shift`/`reset` はさまざまな effect の模倣を可能にするほか、部分評価アルゴリズムや Web アプリケーションの実装にも有用であることが知られている [12, 21]。したがって、依存型と `shift`/`reset` をともにもつ言語を設計すれば、前者を使って項の性質を正確に記述しながら、後者を使って複雑な動作を実現することが可能になる。継続を明示的に扱うことは一種の副作用であるため、`shift`/`reset` を依存型付きの言語に導入する際には、何らかの制約が必要となることが容易に想像されるが、本研究では、effect と依存型に関する先行研究と同様に、型に現れる項を pure なものに制限する。このような制限を与えると、健全な型システムを設計することができる。

本稿ではまず、単純型付き λ 計算における `shift`/`reset` の振る舞いを説明する（2 節）。次に、計算体系を依存型で拡張し、例を用いながら、型の依存性を pure な項に制限する理由を明らかに

する (3 節)。この議論に従って、`shift/reset` をもつ依存型付き言語 $\lambda_{\Pi}^{s/r}$ を定義し、preservation や progress などの性質を示す (4 節)。なお、`shift/reset` を含むプログラムは、CPS 変換によってこれらを含まないものに変換することができる。本研究では、Bowman ら [9] のアイディアに基づいて $\lambda_{\Pi}^{s/r}$ に対する CPS 変換を定義し、その変換が型を保存することを証明する (5 節)。その後、$\lambda_{\Pi}^{s/r}$ で実装可能なプログラムの例を紹介する (6 節)。ここまでの結果は、`shift/reset` を定理証明支援系に導入することの可能性を示唆するが、これにつながる議論として、`shift/reset` の論理的な意味について考察する (7 節)。最後に関連研究と今後の課題について述べる (8 - 9 節)。

本稿では $\lambda_{\Pi}^{s/r}$ の表現に青いサンセリフ体 (abc) を、CPS 変換後の表現に赤い太字のセリフ体 (**abc**) を使用する。両者は白黒でも見分けられるが、印刷する際にはカラーで出力することを推奨する。また、言語の完全な定義と証明は `https://sites.google.com/site/youyoucong212/ppl2018` から入手可能である。

2 shift/reset と answer type の変更

Danvy and Filinski の `shift/reset` は、限定継続を扱うためのオペレータであり、前者が後者によって限定された継続を捕捉する。`shift` を $\mathcal{S}$、`reset` を $\langle\rangle$ と表す場合、$1 + \langle 2 + \mathcal{S}k : int \to int. k\ (k\ 3)\rangle$ というプログラムの中で、`shift` は `reset` に囲まれた部分の継続 $\lambda x : int. \langle 2 + x\rangle$ を捕捉し、変数 k をこの継続に束縛したうえで、本体 $k\ (k\ 3)$ を計算する。したがって、$k\ (k\ 3)$ は 7 になり、最後に `reset` の外側の 1 が足されて、最終的に 8 が得られる。

`shift` を使ったプログラムを実行すると、answer type（`reset` が返す型）が変化することがある。例として、$1 :: \langle 2 + \mathcal{S}k : int \to int. [k\ 3]\rangle$ というプログラムを考えよう。このプログラムの実行結果は [1; 5] である。ここで、`reset` に囲まれている部分に注目すると、項全体が足し算の形をしているため、この部分項は int 型をもつように見える。しかし、この項を `reset` の中で実行すると、$[k\ 3] = [2 + 3] = [5]$ というリストが得られる。これを、この部分項の answer type が int から int list に変化したと表現する。このような現象をサポートするためには、実行前後における answer type の情報を明示的にもつ型判断が必要となる。本研究では、Asai and Kameyama [5] に従って、2 種類の型判断を使用する。まず、pure でない項の型付けには、$\Gamma; \alpha \vdash e : A; \beta$ という形の型判断を用いる。これは、「項 e は型環境 Γ のもとで型 A をもち、その実行は initial answer type α を final answer type β に変更する」ということを意味する。e が pure である場合は、$\Gamma \vdash_p e : A$ という形の型判断を用いる。これは、任意の型 α に対して $\Gamma; \alpha \vdash e : A; \alpha$ が導出できることに対応している。pure な項は任意の文脈の中で実行でき、answer type を変化させない、という意味である。

3 依存型付き言語における shift/reset の使用

依存型付きの言語で `shift/reset` を使うと何が起きるだろうか。本節では、自然数と、長さでインデックスされたリストをもつ言語を用いて、`shift/reset` の使用に関してどのような制限が必要かを見ていく。以下、$\mathbb{N}$ を自然数の型、$\mathrm{L}(n)$ を n 個の自然数からなるリストの型とする。長さが m のリスト l に自然数 n を加える操作は $::\ m\ n\ l$ と表し、2 つのリストの連結は @ 関数を用いる。この関数は $\Pi m\ n : \mathbb{N}. \mathrm{L}(m) \to \mathrm{L}(n) \to \mathrm{L}(m + n)$ という型をもち、2 つのリストの前に、それらの長さを表す自然数を受け取る。では、以下の 2 つのプログラムを考えよう。

(1) $@\ 1\ 2\ (\mathcal{S}k : \mathrm{L}(1) \to \mathrm{L}(3).\ @\ 3\ 1\ (k\ [1])\ [4])\ [2; 3]$

(2) $@\ (\mathcal{S}k : \mathbb{N} \to \mathrm{L}(3).\ @\ 3\ 1\ (k\ 1)\ [4])\ 2\ [1]\ [2; 3]$

(1) を `reset` の中で実行した場合、`shift` が切り取る継続 k は、「受け取ったリストを [2; 3] と結合させる」という計算となる。したがって、k [1] は [1; 2; 3] に簡約され、これを [4] と結合させることで、最終的にリスト [1; 2; 3; 4] が得られる。実行結果が分かったところで、今度は (1) に型を付けてみよう。項全体が @ 1 2 ... という形であることから、(1) は $\mathrm{L}(1+2) = \mathrm{L}(3)$ という型をもつ。次に、`shift` が切り取る継続 k のアノテーションを見ると、返す値の型が $\mathrm{L}(3)$ となっているため、initial answer type は $\mathrm{L}(3)$ となる。最後に、結果として得られるリストは長さが 4 であるため、final answer type は $\mathrm{L}(4)$ となる。したがって、(1) は以下のような型判断をもつ：

$$\Gamma;\, \mathrm{L}(3) \vdash @\ 1\ 2\ (\mathcal{S}k : \mathrm{L}(1) \to \mathrm{L}(3).\, @\ 3\ 1\ (k\ [1])\ [4]) : \mathrm{L}(3);\, \mathrm{L}(4)$$

続いて、(2) について考えよう。(1) と同様に、`reset` の中で (2) を実行すると、同じリストが得られそうである。しかし、(2) はどのような型をもつだろうか。項の形から、(2) は $\mathrm{L}(x+2)$（ただし、x は `shift` 節全体）という型になりそうだが、$x+2$ は `reset` に囲まれていないため、`shift` の簡約規則を適用できず、値にすることができない。したがって、(2) に $\mathrm{L}(x+2)$ という型を付けたとしても、その型から (2) の長さの情報は得られない。このような型は有益ではないため、本研究では (2) を型付け不可能な項として捉える。

上の例で得られた観察を一般化してみよう。(1) と (2) の違いは、`shift` 節が関数適用の結果の型に現れるかどうかにある。一般に、pure でない項は、それ自身を見ただけではどのような値に簡約されるかが分からない。つまり、pure でない項に依存する型をもつ項は、実行時の環境によって異なる型をもち得る。冒頭でも述べた通り、これは望ましくない性質である。この理由から、本研究では、型に現れる項を pure なものに制限するという方針をとる。

4 依存型・shift/reset 付き λ 計算

本節では、依存型と `shift`/`reset` を持つ値呼びの言語 $\lambda_{\Pi}^{s/r}$ を定義する。図 1 に構文を示した。まず、型文脈はベースとなる空の文脈 $\bullet$ と、3 種類の拡張によって定義される。拡張は変数によるもの、変数とその定義によるもの、同値性によるものがあり、それぞれ λ 抽象、let 文、match 文の型規則で使われる。次に定義されている種 $*$ は、型の型を表す。

型は、3 種類の帰納的な型と、2 種類の関数型からなる。このうち、前者は unit 型 Unit、自然数を表す型 $\mathbb{N}$、n 個の自然数からなるリストの型 $\mathsf{L(n)}$ からなる。関数型は、関数本体の実行前後における answer type を明示的にもつ。本体が pure である場合は、$\Pi\,\mathsf{x}:\mathsf{A}.\,\Pi\,\alpha:*.\,\alpha \,\|\, \mathsf{B} \,\|\, \alpha$ という形となる。これは、answer type が任意であり、関数本体の実行前後において変化しないことを表す。本体が pure でない関数の型は $\Pi\,\mathsf{x}:\mathsf{A}.\,\alpha \,\|\, \mathsf{B} \,\|\, \beta$ という形であり、関数適用が行われた際に、answer type が α から β に変化することを意味する。引数の answer type に言及していないのは、この言語が値呼びの意味論をもち、関数適用で渡される引数が pure な値であることが保証されているからである。なお、関数型の中に現れる B, α, β は引数 x を自由変数として含み得る。

項については、値というカテゴリを別途定義している。値は変数、λ 抽象、再帰関数、コンストラクタに値を渡したものからなる。項は値か、関数適用、let 文、コンストラクタに項を渡したもの、パターンマッチ、`shift` 節あるいは `reset` 節である。

図 2 は評価文脈と簡約規則を定義している。評価文脈のうち、E は一般の文脈、F は hole を `reset` で囲んでいない（pure な）文脈であり、いずれも値呼びで、左から右への実行を規定している。たとえば、関数適用 $\mathsf{e_1}\ \mathsf{e_2}$ を実行する際は、文脈 $\mathsf{E}\ \mathsf{e_2}$ で関数 $\mathsf{e_1}$ を値 $\mathsf{v_1}$ に評価し、$\mathsf{v_1}\ \mathsf{E}$ で引数 $\mathsf{e_2}$ を値 $\mathsf{v_2}$ に評価したうえで、β 簡約を行う。簡約規則は $\Gamma \vdash \mathsf{e} \triangleright \mathsf{e'}$ という形の関係によって定義される（以降、型環境を省略する場合がある）。一番上の規則は δ 簡約とよばれ、型環境中の情報を使った変数の置き換えを行う。次に、`shift` の簡約規則を見ると、hole と `reset` の間が pure

$$
\begin{array}{rcl}
\Gamma & ::= & \bullet \mid \Gamma, \mathsf{x} : \mathsf{A} \mid \Gamma, \mathsf{x} = \mathsf{A} : \mathsf{e} \mid \Gamma, \mathsf{e} \equiv \mathsf{e} \\
\kappa & ::= & * \\
\mathsf{A}, \alpha & ::= & \mathsf{Unit} \mid \mathbb{N} \mid \mathsf{L}(\mathsf{e}) \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \Pi\, \alpha : *.\, \alpha \parallel \mathsf{B} \parallel \alpha \mid \Pi\, \mathsf{x} : \mathsf{A}.\, \alpha \parallel \mathsf{B} \parallel \beta \\
\mathsf{v} & ::= & \mathsf{x} \mid \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e} \mid \mathsf{fix}\ \mathsf{f} : \mathsf{A}.\, \mathsf{x} : \mathsf{B}.\, \mathsf{e} \mid () \mid \mathsf{z} \mid \mathsf{suc}\ \mathsf{v} \mid [\,] \mid :: \mathsf{v}\ \mathsf{v}\ \mathsf{v} \\
\mathsf{e} & ::= & \mathsf{v} \mid \mathsf{e}\ \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{suc}\ \mathsf{e} \mid :: \mathsf{e}\ \mathsf{e}\ \mathsf{e} \\
 & & \mid \mathsf{match}\ \mathsf{e}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \parallel \mathsf{suc}\ \mathsf{m} \to \mathsf{e} \mid \mathsf{match}\ \mathsf{e}\ \mathsf{with}\ [\,] \to \mathsf{e} \parallel :: \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e} \\
 & & \mid \mathcal{S}\mathsf{k} : \mathsf{A}\ .\, \mathsf{e} \mid \langle \mathsf{e} \rangle
\end{array}
$$

図 1. $\lambda_{\Pi}^{s/r}$ の構文

な文脈になっており、この `reset` が最も内側の `reset` であることが分かる。ここで定義した簡約規則の反射的・推移的かつ compatible な閉包を $\triangleright^\star$ と表す。

図 3 は型検査時に使う同値性を定義している。2 つの項（型）が等しいのは、それらが $\triangleright^\star$ によって同じ値に簡約されるときである。なお、本研究で用いる同値性は型なしの関係であり、両辺が同じ型をもつことを要求しない (5.2 節参照)。また、以降では型環境を省略することにする。

この言語を設計する際に、型に現れる項として pure なもののみを許すという方針にしたが、$\lambda_{\Pi}^{s/r}$ では、図 4 - 5 の型規則を用いて型の依存性を制限する。まず、変数定義による型環境の拡張 (Extend2) に注目すると、e が pure であることが要求されている。型環境中の変数定義は、δ 簡約によって使われるが、ここで e を pure な項に制限することで、型の中に現れる変数が pure でない項に置き換えられることを防いでいる。同じような制約が同値性による拡張 (Extend3) でみられる。リスト型の導出 (List) も、インデックス e として pure な項を要求することで、リストの長さが静的に定まることを保証している。

項の規則については、部分項が pure であるか否かによって、異なる導出規則が用意されている。各規則名の最後の数字は、pure $<$ impure という順序に従って、部分項の可能な組み合わせを辞書式に並べたときの順番を表す。たとえば、関数適用は関数の本体、関数部分、引数部分のそれぞれが pure か impure である可能性があるため、(App1) から (App8) までの 8 種類の規則がある。このうち、結果の型 B が自由変数 x を含むのは、引数が pure である奇数番目の規則のみである。パターンマッチも同様に、結果の型が依存性をもつのは、マッチされる項が pure である場合に限られる。なお、$\lambda_{\Pi}^{s/r}$ はリストに依存する型をもたないため、リスト e をマッチした結果の型は e の長さのみに依存する。

ここで、いくつかの型付け規則を詳しく見ていく。まず、自然数のパターンマッチの中で、マッチされる表現 e が pure である場合は、各ブランチを型検査するときに、「e が現在のパターンと同値である」という情報が必要となる。これは、ブランチ $\mathsf{e}_1, \mathsf{e}_2$ の型がパターンに依存しなければならないのに対し、e_1 と e_2 が e に依存した型をもち得るからである。なお、ここで導入される同値性は矛盾する場合がある。たとえば、(MatchN1) の中で e が $\mathsf{suc}\ \mathsf{z}$ であった場合、e_1 の型検査時に導入される同値性は $\mathsf{suc}\ \mathsf{z} \equiv \mathsf{z}$ であるが、このような場合は e_2 が必ず実行される [19]。

次に、`shift` に関する規則を見ると、継続の型が Π 型ではなく、依存性をもたない関数型 $\to$ で表されている。これは、型の依存性が pure な項に限られていることによる。継続の型が Π 型を必要とするのは、その `shift` を限定する `reset` の本体の型が定まらない場合である。たとえば、$\langle \mathsf{F}[\mathcal{S}\mathsf{k} : (\Pi\, \mathsf{x} : \mathsf{A}.\, \Pi\, \alpha : *.\, \alpha \parallel \mathsf{B}(\mathsf{x}) \parallel \alpha)\ .\, \mathsf{e}] \rangle$ という項があった場合、文脈 F が返す値の型は、k が返す型 $\mathsf{B}(\mathsf{x})$ の x を hole に入っている項で置き換えたものとなるため、`reset` の本体は $\mathsf{B}(\mathcal{S}\mathsf{k} : \_\ .\, \mathsf{e})$ という型となる。$\lambda_{\Pi}^{s/r}$ はこのような型を許さないため、継続の型は必ず $\to$ を使って表すことができるものとなる。さらに、この関数型は継続の本体が pure であることを表している。これは、`shift`

$$
\begin{aligned}
\mathsf{E} \quad ::= \quad & [\,] \mid \mathsf{E}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{E} \mid \mathsf{let}\ \mathsf{x} = \mathsf{E}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{suc}\ \mathsf{E} \mid :: \mathsf{E}\ \mathsf{e}\ \mathsf{e} \mid :: \mathsf{v}\ \mathsf{E}\ \mathsf{e} \mid :: \mathsf{v}\ \mathsf{v}\ \mathsf{E} \\
& \mid \mathsf{match}\ \mathsf{E}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \parallel \mathsf{suc}\ \mathsf{m} \to \mathsf{e} \mid \mathsf{match}\ \mathsf{E}\ \mathsf{with}\ [\,] \to \mathsf{e} \parallel :: \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e} \mid \langle \mathsf{E} \rangle \\
\mathsf{F} \quad ::= \quad & [\,] \mid \mathsf{F}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{F} \mid \mathsf{let}\ \mathsf{x} = \mathsf{F}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{suc}\ \mathsf{F} \mid :: \mathsf{F}\ \mathsf{e}\ \mathsf{e} \mid :: \mathsf{v}\ \mathsf{F}\ \mathsf{e} \mid :: \mathsf{v}\ \mathsf{v}\ \mathsf{F} \\
& \mid \mathsf{match}\ \mathsf{F}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e} \parallel \mathsf{suc}\ \mathsf{m} \to \mathsf{e} \mid \mathsf{match}\ \mathsf{F}\ \mathsf{with}\ [\,] \to \mathsf{e} \parallel :: \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}
\end{aligned}
$$

$$
\begin{aligned}
\Gamma \vdash \mathsf{x} &\ \triangleright\ \mathsf{e} \quad \text{if } \mathsf{x} = \mathsf{e} : \mathsf{A} \in \Gamma \\
\Gamma \vdash (\lambda \mathsf{x} : \mathsf{A}.\, \mathsf{e})\ \mathsf{v} &\ \triangleright\ \mathsf{e}[\mathsf{v}/\mathsf{x}] \\
\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{v}\ \mathsf{x} &\ \triangleright\ \mathsf{v} \\
\Gamma \vdash (\mathsf{fix}\ \mathsf{f} : \mathsf{A}.\, \mathsf{x} : \mathsf{B}.\, \mathsf{e})\ \mathsf{v} &\ \triangleright\ \mathsf{e}[\mathsf{fix}\ \mathsf{f} : \mathsf{A}.\, \mathsf{x} : \mathsf{B}.\, \mathsf{e}/\mathsf{f}, \mathsf{v}/\mathsf{x}] \\
\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{v}\ \mathsf{in}\ \mathsf{e} &\ \triangleright\ \mathsf{e}[\mathsf{v}/\mathsf{x}] \\
\Gamma \vdash \mathsf{match}\ \mathsf{z}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \parallel \mathsf{suc}\ \mathsf{m} \to \mathsf{e}_2 &\ \triangleright\ \mathsf{e}_1 \\
\Gamma \vdash \mathsf{match}\ \mathsf{suc}\ \mathsf{v}\ \mathsf{with}\ \mathsf{z} \to \mathsf{e}_1 \parallel \mathsf{suc}\ \mathsf{m} \to \mathsf{e}_2 &\ \triangleright\ \mathsf{e}_2[\mathsf{v}/\mathsf{m}] \\
\Gamma \vdash \mathsf{match}\ [\,]\ \mathsf{with}\ [\,] \to \mathsf{e}_1 \parallel :: \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 &\ \triangleright\ \mathsf{e}_1 \\
\Gamma \vdash \mathsf{match}\ ::\ \mathsf{v}\ \mathsf{v}_1\ \mathsf{v}_2\ \mathsf{with}\ [\,] \to \mathsf{e}_1 \parallel :: \mathsf{m}\ \mathsf{h}\ \mathsf{t} \to \mathsf{e}_2 &\ \triangleright\ \mathsf{e}_2[\mathsf{v}/\mathsf{m}, \mathsf{v}_1/\mathsf{h}, \mathsf{v}_2/\mathsf{t}] \\
\Gamma \vdash \langle \mathsf{F}[\mathcal{S}\mathsf{k} : \_\,.\, \mathsf{e}] \rangle &\ \triangleright\ \langle \mathsf{e}[\lambda \mathsf{v} : \_.\, \langle \mathsf{F}[\mathsf{v}] \rangle / \mathsf{k}] \rangle \\
\Gamma \vdash \langle \mathsf{v} \rangle &\ \triangleright\ \mathsf{v}
\end{aligned}
$$

図 2. 評価文脈と簡約規則

$$
\frac{\Gamma \vdash \mathsf{e}_1 \rhd^\star \mathsf{v} \quad \Gamma \vdash \mathsf{e}_2 \rhd^\star \mathsf{v}}{\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2}\ (\equiv\text{-RED})
$$

図 3. 同値性

によって捕捉される継続の本体が `reset` に囲まれるからである。

最後に、型の付け換えを行う規則 (CONV1), (CONV2) に注目しよう。前者は通常の依存型付き言語でみられる規則と同じであるが、後者は answer type の付け換えも可能にしている。

$\lambda_{\Pi}^{s/r}$ のもつ性質として、以下の 3 つの定理を証明することができる。

定理 1 (Preservation).

1. $\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ かつ $\mathsf{e} \rhd^\star \mathsf{e}'$ ならば、$\Gamma \vdash_p \mathsf{e}' : \mathsf{A}$。
2. $\Gamma; \alpha \vdash \mathsf{e} : \mathsf{A}; \beta$ かつ $\mathsf{e} \rhd^\star \mathsf{e}'$ ならば、$\Gamma; \alpha \vdash \mathsf{e}' : \mathsf{A}; \beta$。

定理 2 (Progress と分解の一意性). $\bullet \vdash_p \mathsf{e} : \mathsf{A}$ が導出可能であれば、e は値であるか、$\mathsf{E}[\mathsf{r}]$ という形に一意に分解される（ただし、E は評価文脈、r は簡約基）。

定理 3 ($\lambda_{\Pi}^{s/r}$ – fix の強正規化定理). $\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ または $\Gamma; \alpha \vdash \mathsf{e} : \mathsf{A}; \beta$ が導出可能であり、かつ e が $\mathsf{fix}\ \mathsf{f} : \mathsf{A}'.\, \mathsf{x} : \mathsf{B}.\, \mathsf{e}'$ という形の項を含まないならば、e は強正規化可能である。

定理 2 において pure な項のみを考えているのは、pure でない項が一般に実行不可能であることによる [5]。また、定理 3 を示す際には、Pierce [25] に従って、型の依存性を除去する変換を定義し、その変換が型と簡約を保存することを証明する。すると、示したい定理は、Asai and Kameyama [5] の体系を自然数とリストで拡張した、単純型付き言語の強正規化定理に帰着する。これについては、`shift`/`reset` のための論理関係 [4] を使うことで示すことができる。

$$\frac{}{\vdash \bullet}\ (\textsc{Empty}) \qquad \frac{\vdash \Gamma \quad \Gamma \vdash A : *}{\vdash \Gamma, x : A}\ (\textsc{Extend1})$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash_p e : A}{\vdash \Gamma, x = e : A}\ (\textsc{Extend2}) \qquad \frac{\vdash \Gamma \quad \Gamma \vdash_p e_1 : A \quad \Gamma \vdash_p e_2 : A}{\vdash \Gamma, e_1 \equiv e_2}\ (\textsc{Extend3})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash *}\ (\textsc{Star}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Unit} : *}\ (\textsc{Unit}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : *}\ (\textsc{Nat}) \qquad \frac{\Gamma \vdash_p e : \mathbb{N}}{\Gamma \vdash \mathsf{L}(e) : *}\ (\textsc{List})$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x{:}A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha : *}\ (\textsc{Pi1}) \qquad \frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \quad \Gamma, x : A \vdash \alpha : * \quad \Gamma, x : A \vdash \beta : *}{\Gamma \vdash \Pi x{:}A.\, \alpha \,\|\, B \,\|\, \beta : *}\ (\textsc{Pi2})$$

図 4. 型環境と型の導出規則

5 CPS 変換

単純型・多相型付き λ 計算では、`shift`/`reset` を含むプログラムを CPS 変換 [26] によって `shift`/`reset` なしのプログラムに変換できる [13, 5]。本節では、Bowman ら [9] のアイディアを用いて $\lambda^{s/r}_{\Pi}$ に対する値呼びの CPS 変換を定義し、変換が型を保存することを証明する。

5.1 CPS 変換後の言語

図 6 に CPS 変換後の言語 λ^{k}_{Π} の構文を示す。型 $\mathbf{\Pi}\,\alpha : *.\,\mathbf{B}$ や項 $\mathbf{e}\ \mathbf{A}$ から分かるように、λ^{k}_{Π} は多相型をもつ言語である。また、λ^{k}_{Π} は `shift`/`reset` が除去されたあとの pure な言語であることから、関数型は answer type に言及しない Π 型となる。通常、CPS 変換は実行順を規定するが、`shift`/`reset` の変換結果は末尾呼び出しでない関数適用を含み、その実行は変換後の言語の評価戦略に依存する。ここでは λ^{k}_{Π} の評価戦略も値呼びとし、一般の項と値を区別する。なお、項の定義に $\mathbf{e} @ \mathbf{A}\ \mathbf{e}$ という構文が加わっているが、これは CPS 変換された項とその継続の関数適用を表しており、意味上は $\mathbf{e}\ \mathbf{A}\ \mathbf{e}$ と同じである（5.2 節参照）。

型規則は基本的に、$\lambda^{s/r}_{\Pi}$ の型規則のうち、すべての部分項が pure であるもの（規則名が (xxx1) であるもの）と、多相型に関する規則からなる。同値性も $\lambda^{s/r}_{\Pi}$ と同様に、簡約によって定義されるが、CPS 変換の型保存を証明するために、新しい型規則 [T-Cont] と同値性 [≡-Cont] を導入する必要がある [9]。なお、[T-Cont] と [≡-Cont] の追加は、矛盾を導かないことが証明されている [9]。詳細は原論文を参照されたいが、おおまかな流れとしては、新たな規則を追加した体系から、extensional Calculus of Constructions (ECC) [10] への変換を定義し、変換の型の保存性と、ECC の無矛盾性から、変換前の体系が矛盾しないことを導出する。次節では、新たに導入した構文と規則の背後にあるアイディアを説明する。

5.2 変換と証明のアイディア

依存型をもつ言語に対する CPS 変換は、単純型付きの言語に比べて、型の保存性の議論が難しいことが知られている [7, 8]。本節では、型保存に関係する問題点と、その解決方法について述べる。以下、項 e と型 A の CPS 変換（厳密には、これらの導出の変換）をそれぞれ $e^{\div}$, A^{+} と表す。

5.2.1 証明の相互依存の回避

型保存の証明が難しい理由の 1 つは、自身の証明と、それに必要な補題の証明が相互依存してしまうからである。依存型付きの言語では、型検査を容易にするために、λ 抽象で束縛される変数に型のアノテーションを付けることが多い。CPS 変換を行うと、継続や部分項の計算結果を表す変数

$$\frac{\vdash \Gamma \quad x : A \in \Gamma \text{ or } x = e : A \in \Gamma}{\Gamma \vdash_p x : A}\ (\textsc{Var}) \qquad \frac{\vdash \Gamma \quad c \text{ は } A \text{ 型のコンストラクタ}}{\Gamma \vdash_p c : A}\ (\textsc{Base-Const})$$

$$\frac{\Gamma, x : A \vdash_p e : B}{\Gamma \vdash_p \lambda x : A.\, e : \Pi x : A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha}\ (\textsc{Abs1})$$

$$\frac{\Gamma, f : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta, x : A; \alpha \vdash e : B; \beta}{\Gamma \vdash_p \text{fix } f : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta.\, x : A.\, e : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta}\ (\textsc{Fix2})$$

$$\frac{\Gamma \vdash_p e_1 : \Pi x : A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha \quad \Gamma \vdash_p e_2 : A}{\Gamma \vdash_p e_1\ e_2 : B\,[e_2/x]}\ (\textsc{App1})$$

$$\frac{\Gamma \vdash_p e_1 : \Pi x : A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha \quad \Gamma; \beta \vdash e_2 : A; \gamma \quad x \notin FV(B)}{\Gamma; \beta \vdash e_1\ e_2 : B; \gamma}\ (\textsc{App2})$$

$$\frac{\Gamma; \beta \vdash e_1 : \Pi x : A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha; \gamma \quad \Gamma \vdash_p e_2 : A}{\Gamma; \beta \vdash e_1\ e_2 : B\,[e_2/x]; \gamma}\ (\textsc{App3})$$

$$\frac{\Gamma; \gamma \vdash e_1 : \Pi x : A.\, \Pi \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha; \delta \quad \Gamma; \beta \vdash e_2 : A; \gamma \quad x \notin FV(B)}{\Gamma; \beta \vdash e_1\ e_2 : B; \delta}\ (\textsc{App4})$$

$$\frac{\Gamma \vdash_p e_1 : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta \quad \Gamma \vdash_p e_2 : A}{\Gamma; \alpha\,[e_2/x] \vdash e_1\ e_2 : B\,[e_2/x]; \beta\,[e_2/x]}\ (\textsc{App5})$$

$$\frac{\Gamma \vdash_p e_1 : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta \quad \Gamma; \beta \vdash e_2 : A; \gamma \quad x \notin FV(B, \alpha, \beta)}{\Gamma; \alpha \vdash e_1\ e_2 : B; \gamma}\ (\textsc{App6})$$

$$\frac{\Gamma; \beta\,[e_2/x] \vdash e_1 : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta; \gamma \quad \Gamma \vdash_p e_2 : A}{\Gamma; \alpha\,[e_2/x] \vdash e_1\ e_2 : B\,[e_2/x]; \gamma}\ (\textsc{App7})$$

$$\frac{\Gamma; \gamma \vdash e_1 : \Pi x : A.\, \alpha \,\|\, B \,\|\, \beta; \delta \quad \Gamma; \beta \vdash e_2 : A; \gamma \quad x \notin FV(B, \alpha, \beta)}{\Gamma; \alpha \vdash e_1\ e_2 : B; \delta}\ (\textsc{App8})$$

$$\frac{\Gamma \vdash_p e_1 : A \quad \Gamma, x = e_1 : A \vdash_p e_2 : B}{\Gamma \vdash_p \text{let } x = e_1 \text{ in } e_2 : B\,[e_1/x]}\ (\textsc{Let1}) \qquad \frac{\Gamma \vdash_p e : \mathbb{N} \quad \Gamma \vdash_p e_1 : \mathbb{N} \quad \Gamma \vdash_p e_2 : L(e)}{\Gamma \vdash_p :: e\ e_1\ e_2 : L(\text{suc } e)}\ (\textsc{Cons1})$$

$$\frac{\begin{array}{c}\Gamma \vdash_p e : \mathbb{N} \quad \Gamma, x : \mathbb{N} \vdash A : * \quad \Gamma, e \equiv z \vdash_p e_1 : A\,[z/x] \\ \Gamma, m : \mathbb{N}, e \equiv \text{suc } m \vdash_p e_2 : A\,[\text{suc } m/x]\end{array}}{\Gamma \vdash_p \text{match } e \text{ with } z \to e_1 \,\|\, \text{suc } m \to e_2 : A\,[e/x]}\ (\textsc{MatchN1})$$

$$\frac{\begin{array}{c}\Gamma \vdash_p e : L(n) \quad \Gamma, x : \mathbb{N} \vdash A : * \quad \Gamma, x \equiv z \vdash_p e_1 : A\,[z/x] \\ \Gamma, m : \mathbb{N}, h : \mathbb{N}, t : L(m), x \equiv \text{suc } m \vdash_p e_2 : A\,[\text{suc } m/x]\end{array}}{\Gamma \vdash_p \text{match } e \text{ with } [\,] \to e_1 \,\|\, :: m\ h\ t \to e_2 : A\,[n/x]}\ (\textsc{MatchL1})$$

$$\frac{\Gamma, c : A \to \Pi \alpha' : *.\, \alpha' \,\|\, \alpha \,\|\, \alpha'; B \vdash e : B; \beta}{\Gamma; \alpha \vdash \mathcal{S}c : (A \to \Pi \alpha' : *.\, \alpha' \,\|\, \alpha \,\|\, \alpha')\ .\ e : A; \beta}\ (\textsc{Shift2}) \qquad \frac{\Gamma; B \vdash e : B; A}{\Gamma \vdash_p \langle e \rangle : A}\ (\textsc{Reset2})$$

$$\frac{\Gamma \vdash_p e : A \quad \Gamma \vdash B : * \quad A \equiv B}{\Gamma \vdash_p e : B}\ (\textsc{Conv1}) \qquad \frac{\begin{array}{c}\Gamma; \alpha \vdash e : A; \beta \quad \Gamma \vdash B : * \quad \Gamma \vdash \alpha' : * \\ \Gamma \vdash \beta' : * \quad A \equiv B \quad \alpha \equiv \alpha' \quad \beta \equiv \beta'\end{array}}{\Gamma; \alpha' \vdash e : B; \beta'}\ (\textsc{Conv2})$$

図 5. 型付け規則（抜粋）

$$
\begin{array}{rcl}
\kappa & ::= & * \\
\mathbf{A} & ::= & \alpha \mid \mathbf{Unit} \mid \mathbb{N} \mid \mathbf{L(e)} \mid \mathbf{\Pi\, x : A.\, B} \mid \mathbf{\Pi\, \alpha : *.\, B} \\
\mathbf{v} & ::= & \mathbf{x} \mid \mathbf{\lambda\, x : A.\, e} \mid \mathbf{\lambda\, \alpha : *.\, e} \mid \mathbf{fix\ f : A.\, x : B.\, e} \mid () \mid \mathbf{z} \mid \mathbf{suc\ v} \mid [\,] \mid \mathbf{::\ v\ v\ v} \\
\mathbf{e} & ::= & \mathbf{v} \mid \mathbf{e\ e} \mid \mathbf{e\ A} \mid \mathbf{e @ A\ e} \mid \mathbf{suc\ e} \mid \mathbf{::\ e\ e\ e} \\
 & & \mid \mathbf{match\ e\ with\ z \to e \parallel suc\ m \to e} \mid \mathbf{match\ e\ with\ [\,] \to e \parallel ::\ m\ h\ t \to e}
\end{array}
$$

$$
\frac{\Gamma \vdash \mathbf{e_1} : \mathbf{\Pi\, \alpha : *.\, (B \to \alpha) \to \alpha} \quad \Gamma, \mathbf{x = e_1\ B\ id : B} \vdash \mathbf{e_2 : A}}{\Gamma \vdash \mathbf{e_1 @ A\ (\lambda\, x : B.\, e_2)} : \mathbf{A}}\ [\text{T-Cont}]
$$

$$
\frac{}{\Gamma \vdash \mathbf{e_1 @ A\ (\lambda\, x : B.\, e_2)} \equiv \mathbf{(\lambda\, x : B.\, e_2)\ (e_1\ B\ id)}}\ [\equiv\text{-Cont}]
$$

図 6. λ_{Π}^{k} の構文と追加の型規則・同値性

が導入されるが、これらに型のアノテーションを付ける際には、変換前の項の型情報が必要となるため、CPS 変換を項の導出に関して帰納的に定義しなければならない。その場合、型の付け替え規則（図 5 の (Conv1) および (Conv2)）に対する変換を定義することになるが、このケースにおける型の保存性を示すためには、CPS 変換が同値性を保存すること、すなわち「$\mathsf{A} \equiv \mathsf{B}$ ならば $\mathsf{A}^{+} \equiv \mathsf{B}^{+}$」ということを示さなければならない。しかし、Coq や Agda のように、型付きの同値性をもつ言語では、同値性の保存を示すために型の保存が必要となる（A, B が依存型である場合、これらの変換の中で項の変換 $\div$ が使われることに注意されたい）。つまり、型付きの同値性を用いた場合、型の保存と同値性の保存を独立して示すことができなくなってしまう。この理由から、本研究では [9] に従い、変換前後の言語において型なしの同値性を用いる。

5.2.2 項に依存した型の同値性の判断

型保存の議論が難しいもう 1 つのケースは、変換する項の型が自身の部分項に依存する場合である。(App7) によって導出された関数適用 $\mathsf{e_1\ e_2}$ を考えよう。導出規則より、引数 $\mathsf{e_2}$ は pure な項である。この関数適用は $\mathsf{\lambda\, k : (B\,[e_2/x])^{+} \to \alpha^{+}.\, e_1^{\div}\ (\lambda\, v_1 : (\Pi\, x : A.\, \alpha \parallel B \parallel \beta)^{+}.\, e_2^{\div}\ (\lambda\, v_2 : A^{+}.\, v_1\ v_2\ k))}$ と変換されるが [26]、ここで部分項 $\mathbf{v_1\ v_2\ k}$ に注目すると、$\mathbf{v_1\ v_2}$ が $\mathsf{B^{+}\,[v_2/x]}$ をドメインとする継続を要求するのに対し、$\mathbf{k}$ のドメインは $\mathsf{(B\,[e_2/x])^{+}}$ となっている。これらの型が同値であることを示したい。

$\lambda_{\Pi}^{s/r}$ は値呼びの言語であるため、B 中の x に代入されるのは $\mathsf{e_2}$ を実行した値である。CPS 変換後の言語において、この値に対応するのは $\mathsf{e_2^{\div}}$ に空の継続（恒等関数）を渡した結果、すなわち、関数適用 $\mathsf{e_2^{\div}\ (\lambda\, x : A^{+}.\, x)}$ の結果である（以降、恒等関数を $\mathbf{id}$ と表す）。今、継続を関数として表しているため、$\mathsf{e_2^{\div}}$ の継続が受け取る引数 $\mathbf{v_2}$ は任意の値になりそうだが、$\mathsf{e_2}$ が pure である場合、$\mathbf{v_2}$ に代入され得るのは、必ず $\mathsf{e_2^{\div}}\ \mathbf{id}$ を計算した結果の値であることが知られている [2]。すると、以下が成立すれば、$\mathbf{v_1\ v_2\ k}$ に正しく型を付けることができる。

1. e が pure であれば、$\mathsf{e^{\div}}$ に恒等関数を渡すことができる。
2. $\mathsf{e^{\div}}$ の継続の引数と $\mathsf{e^{\div}}\ \mathbf{id}$ の同値性が必要であるときに、その情報が型環境に入っている。
3. $\mathsf{(B\,[e/x])^{+}}$ は $\mathsf{B^{+}\,[e^{\div}\ id/x]}$ と同値である。

1 を保証するために、本研究では A 型の pure な項を $\mathsf{\Pi\, \alpha : *.\, (A^{+} \to \alpha) \to \alpha}$ 型の項に変換する。$\mathsf{\Gamma \vdash_p e : A}$ は任意の α に対して $\mathsf{\Gamma; \alpha \vdash e : A; \alpha}$ が導出できることを意味するため、e の変換結果 $\mathsf{e^{\div}}$ には多相の answer type をもたせることができる。すると、$\mathsf{e^{\div}}$ に $\mathsf{A^{+}}$ を渡し、answer type を $\mathsf{A^{+}}$ に具体化することで、$\mathsf{A^{+} \to A^{+}}$ 型の恒等関数を継続として渡せるようになる。

2 は新しい構文 @ を使うことで保証する。pure な項 e の変換結果が $\mathbf{A}$ 型を返す継続 $\lambda \mathbf{x} : \mathsf{B}^+. \mathbf{e_2}$ を受け取り、$\mathbf{e_2}$ を型付けする際に $\mathbf{x} = \mathsf{e}^\div\ \mathsf{B}^+\ \mathbf{id}$ という同値性が必要である場合、e の変換とその継続の関数適用を $\mathsf{e}^\div\ @\ \mathbf{A}\ (\lambda \mathbf{x} : \mathsf{B}^+. \mathbf{e_2})$ と表す。この項の型付けには図 6 の [T-Cont] を使うが、これにより、必要な同値性を使って継続の本体を型付けすることが可能になる。

3 を示すためには、図 6 の同値性 [≡-Cont] を用いる。この同値性は、「pure な項の変換結果 $\mathbf{e_1}$ に継続を渡すのは、$\mathbf{e_1}$ を恒等関数で実行し、その値を継続に渡すのと等しい」ということを意味し、「pure な項を `reset` で囲んだときに意味が変化しない」と言い換えることもできる。これは、answer type を多相にしたことによって得られる free theorem [33] である。なお、この同値性は $\mathbf{e_1}$ が $\mathbf{\Pi}\,\alpha : *. (\mathbf{B} \to \alpha) \to \alpha$ という多相の型をもつときにのみ意味をなすが、ここでは型なしの同値性を扱っているため、$\mathbf{e_1}$ の型を前提にもたせることができない。しかし、λ^k_Π で同値性を使って型の付け替えを行うときには、[Conv] 規則によって、付け替え前後の型が導出可能であることがチェックされるため、導出不可能な項や型の同値性が用いられることはない。[≡-Cont] の場合、@ の導入規則 [T-Cont] が $\mathbf{e_1}$ の型として多相のものを要求するため、$\mathbf{e_1}$ は必ず正しい型をもつ。さらに、$\mathbf{e_1}$ が多相の型をもつことによって、[≡-Cont] の同値性が矛盾しない、ということが保証される。

5.3 CPS 変換の定義

上の議論に従い、$\lambda^{s/r}_\Pi$ から λ^k_Π への値呼びの CPS 変換を定義する（図 7 - 8）。変換は図 4 - 5 の導出一つひとつに対して定義される。たとえば、(Empty) $\overset{\perp}{\rightsquigarrow} \bullet$ は導出 (Empty) の変換が $\bullet$ であることを意味する。また、項については、pure であるかどうかによって、異なる形の項に変換する。

- $\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ であるとき、$\lambda\, \alpha : *. \lambda \mathbf{k} : \mathsf{A}^+ \to \alpha. \ldots$ という形に変換する。
- $\Gamma;\, \alpha \vdash \mathsf{e} : \mathsf{A};\, \beta$ であるとき、$\lambda \mathbf{k} : \mathsf{A}^+ \to \alpha^+. \ldots$ という形に変換する。

図 7 の中で、(Extend2), (Extend3), (List) の変換を見ると、もとの規則に現れる項 e が $\mathsf{e}^\div\ \mathsf{A}^+\ \mathbf{id}$ という形に変換されていることが分かる。5.2.2 節で説明したように、これは e を実行した結果の値に対応する項である。$\lambda^{s/r}_\Pi$ を定義した際に、これらの規則に現れる項を pure なものに制限したため、$\mathsf{e}^\div$ は多相の answer type をもち、$\mathsf{e}^\div\ \mathsf{A}^+\ \mathbf{id}$ は必ず正しく型が付く。また、関数型 $\Pi\, \mathsf{x} : \mathsf{A}.\, \alpha \,\|\, \mathsf{B} \,\|\, \beta$ は、本体の answer type に言及していた部分が関数型 $\to$ に展開され、「A^+ 型の引数と $\mathsf{B}^+ \to \alpha^+$ 型の継続を受け取ったら、β^+ 型の値を返す」という形になる。これは、$\lambda^{s/r}_\Pi$ において、変換前の型をもつ関数が「A 型の引数を受け取り、hole が B 型で α 型を返すような文脈の中で本体が実行されると、β 型の値を返す」という挙動を示すのに対応している。なお、変換後の型に依存性を伴わない関数型が使われている理由は、5.4 節で述べる。

項の変換は、単純型付き言語に対する値呼びの CPS 変換とほぼ同じだが、型が部分項に依存するケースにおいて、関数適用 @ を使っている。こうすることで、型保存の証明を行うときに、(T-Cont) が使えるようになる。この型規則は、@ の左に置かれる部分項 $\mathbf{e_1}$ の値を $\mathbf{e_1}\ \mathbf{B}\ \mathbf{id}$ という計算によって取り出すが、ここでも $\lambda^{s/r}_\Pi$ において型に現れる部分項が pure であることが保証されているため、$\mathbf{e_1}\ \mathbf{B}\ \mathbf{id}$ は正しく型が付く。

5.4 型保存の証明

本節では、CPS 変換が型を保存することを証明する。最終的に示したいのは以下の定理である。

1. $\Gamma \vdash_p \mathsf{e} : \mathsf{A} \Rightarrow \Gamma^+ \vdash \mathsf{e}^\div : \mathbf{\Pi}\, \alpha : *. (\mathsf{A}^+ \to \alpha) \to \alpha$
2. $\Gamma;\, \alpha \vdash \mathsf{e} : \mathsf{A};\, \beta \Rightarrow \Gamma^+ \vdash \mathsf{e}^\div : (\mathsf{A}^+ \to \alpha^+) \to \beta^+$

$$(\textsc{Empty})\ \overset{+}{\rightsquigarrow}\ \bullet \qquad (\textsc{Extend1})\ \overset{+}{\rightsquigarrow}\ \Gamma^+, \mathbf{x} : A^+ \qquad (\textsc{Extend2})\ \overset{+}{\rightsquigarrow}\ \Gamma^+, \mathbf{x} = e^{\div}\ A^+\ \mathbf{id} : A^+$$

$$(\textsc{Extend3})\ \overset{+}{\rightsquigarrow}\ \Gamma^+, e_1{}^{\div}\ A^+\ \mathbf{id} \equiv e_2{}^{\div}\ A^+\ \mathbf{id}$$

$$(\textsc{Star})\ \overset{+}{\rightsquigarrow}\ * \qquad (\textsc{Unit})\ \overset{+}{\rightsquigarrow}\ \mathbf{Unit} \qquad (\textsc{Nat})\ \overset{+}{\rightsquigarrow}\ \mathbb{N} \qquad (\textsc{List})\ \overset{+}{\rightsquigarrow}\ \mathbf{L}(e^{\div}\ \mathbb{N}\ \mathbf{id})$$

$$(\textsc{Pi1})\ \overset{+}{\rightsquigarrow}\ \mathbf{\Pi}\, \mathbf{x} : A^+. \mathbf{\Pi}\, \alpha : *. (B^+ \to \alpha) \to \alpha \qquad (\textsc{Pi2})\ \overset{+}{\rightsquigarrow}\ \mathbf{\Pi}\, \mathbf{x} : A^+. (B^+ \to \alpha^+) \to \beta^+$$

図 7. 型環境と型の変換（ただし、$e^{\div}$, A^+, Γ^+ はそれぞれ e, A, Γ の導出の変換結果）

変換結果の型に依存性をもたない関数型 $\to$ が 2 つ使われているが、1 つ目は `shift` の継続の型が依存性をもたないのと同じ理由、2 つ目は、$\lambda_{\Pi}^{s/r}$ において、型が関数型をもつ項に依存し得ないという理由による。上の定理を示すために、変換と代入が可換であること、変換が簡約関係を保存すること、変換が同値性を保存することを、この順で証明する。

補題 1 (変換と代入の可換性). $\Gamma \vdash_p e' : A'$ であるとき、以下が成り立つ。

1. $(A\,[e'/x])^+ \equiv A^+\,[e'^{\div}\ A'^+\ \mathbf{id}/\mathbf{x}]$
2. $(e\,[e'/x])^{\div} \equiv e^{\div}\,[e'^{\div}\ A'^+\ \mathbf{id}/\mathbf{x}]$

補題 2 (簡約の保存).

1. $\Gamma \vdash A : *$ かつ $A \triangleright^\star A'$ ならば、$A^+ \triangleright^\star \mathbf{A}'$ かつ $\mathbf{A}' \equiv A'^+$ を満たす $\mathbf{A}'$ が存在する。
2. $\Gamma \vdash_p e : A$ かつ $e \triangleright^\star e'$ ならば、$e^{\div} \triangleright^\star \mathbf{e}'$ かつ $\mathbf{e}' \equiv e'^{\div}$ を満たす $\mathbf{e}'$ が存在する。

補題 3 (同値性の保存).

1. $A \equiv A'$ ならば、$A^+ \equiv A'^+$。
2. $e \equiv e'$ ならば、$e^{\div} \equiv e'^{\div}$。

変換と代入の可換性では、代入される項 e' が pure である場合のみを考慮する。$\lambda_{\Pi}^{s/r}$ において、型への代入が許されるのは pure な項のみであり、項への代入が起きるのは、値呼びの β 簡約によって変数を pure な値で置き換えるときのみであるため、e' が pure な場合を考えれば十分である。

上の補題を用いて、型環境、型、および項の導出が保存されることを証明する。以下の 1 から 4 は、導出に関する mutual な帰納法によって示される。

定理 4 (導出と型の保存).

1. $\vdash \Gamma \Rightarrow \vdash \Gamma^+$
2. $\Gamma \vdash A : * \Rightarrow \Gamma^+ \vdash A^+ : *$
3. $\Gamma \vdash_p e : A \Rightarrow \Gamma^+ \vdash e^{\div} : \mathbf{\Pi}\, \alpha : *. (A^+ \to \alpha) \to \alpha$
4. $\Gamma; \alpha \vdash e : A; \beta \Rightarrow \Gamma^+ \vdash e^{\div} : (A^+ \to \alpha^+) \to \beta^+$

6 応用例

$\lambda_{\Pi}^{s/r}$ で型付けできる項の例として、書き換え可能な状態を扱うプログラムを紹介する [6]。まず、状態にアクセスするための関数 get と、状態の値を変更する関数 tick を以下のように定義する（以降、`shift` が捕捉する継続の型を省略する）。

$$\begin{aligned} \mathsf{get} &\overset{\mathrm{def}}{\equiv} \lambda\,() : \mathsf{Unit}.\, \mathcal{S}k.\, \lambda\,s : \mathbb{N}.\, k\ s\ s \\ \mathsf{tick} &\overset{\mathrm{def}}{\equiv} \lambda\,() : \mathsf{Unit}.\, \mathcal{S}k.\, \lambda\,s : \mathbb{N}.\, k\ ()\ (\mathsf{suc}\ s) \end{aligned}$$

$$(\textsc{Var})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\,\lambda\,\mathbf{k} : \mathsf{A}^{+} \to \alpha.\,\mathbf{k}\ \mathbf{x} \qquad (\textsc{Base-Const})\ \overset{\div}{\rightsquigarrow}\ \lambda\,\alpha : *.\,\lambda\,\mathbf{k} : \mathsf{A}^{+} \to \alpha.\,\mathbf{k}\ \mathbf{c}$$

$$(\textsc{Abs1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha' : *.\,\lambda\,\mathbf{k} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \parallel \mathsf{B} \parallel \alpha)^{+} \to \alpha'.\,\mathbf{k}\ (\lambda\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{e}^{\div})$$

$$(\textsc{Fix2})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha' : *.\,\lambda\,\mathbf{k} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+} \to \alpha'.\,\mathbf{k}\ (\mathbf{fix}\ \mathbf{f} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+}.\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{e}^{\div})$$

$$(\textsc{App1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha' : *.\ \lambda\,\mathbf{k} : (\mathsf{B}\,[\mathsf{e}_2/\mathsf{x}])^{+} \to \alpha'.\ \mathsf{e}_1{}^{\div}\ \alpha'\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \parallel \mathsf{B} \parallel \alpha)^{+}.\,\mathsf{e}_2{}^{\div}\ @\ \alpha'\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \alpha'\ \mathbf{k}))$$

$$(\textsc{App2})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{B}^{+} \to \beta^{+}.\,\mathsf{e}_1{}^{\div}\ \gamma^{+}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \parallel \mathsf{B} \parallel \alpha)^{+}.\,\mathsf{e}_2{}^{\div}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \beta^{+}\ \mathbf{k}))$$

$$(\textsc{App3})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : (\mathsf{B}\,[\mathsf{e}_2/\mathsf{x}])^{+} \to \beta^{+}.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \parallel \mathsf{B} \parallel \alpha)^{+}.\,\mathsf{e}_2{}^{\div}\ @\ \beta^{+}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \beta^{+}\ \mathbf{k}))$$

$$(\textsc{App4})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{B}^{+} \to \beta^{+}.\,\mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \parallel \mathsf{B} \parallel \alpha)^{+}.\,\mathsf{e}_2{}^{\div}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \beta^{+}\ \mathbf{k}))$$

$$(\textsc{App5})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : (\mathsf{B}\,[\mathsf{e}_2/\mathsf{x}])^{+} \to (\alpha\,[\mathsf{e}_2/\mathsf{x}])^{+}.\ \mathsf{e}_1{}^{\div}\ (\beta\,[\mathsf{e}_2/\mathsf{x}])^{+}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+}.\,\mathsf{e}_2{}^{\div}\ @\ (\beta\,[\mathsf{e}_2/\mathsf{x}])^{+}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{k}))$$

$$(\textsc{App6})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{B}^{+} \to \alpha^{+}.\,\mathsf{e}_1{}^{\div}\ \gamma^{+}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+}.\,\mathsf{e}_2{}^{\div}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{k}))$$

$$(\textsc{App7})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : (\mathsf{B}\,[\mathsf{e}_2/\mathsf{x}])^{+} \to (\alpha\,[\mathsf{e}_2/\mathsf{x}])^{+}.\ \mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+}.\,\mathsf{e}_2{}^{\div}\ @\ (\beta\,[\mathsf{e}_2/\mathsf{x}])^{+}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{k}))$$

$$(\textsc{App8})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{B}^{+} \to \alpha^{+}.\,\mathsf{e}_1{}^{\div}\ (\lambda\,\mathbf{v_1} : (\Pi\,\mathsf{x} : \mathsf{A}.\,\alpha \parallel \mathsf{B} \parallel \beta)^{+}.\,\mathsf{e}_2{}^{\div}\ (\lambda\,\mathbf{v_2} : \mathsf{A}^{+}.\,\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{k}))$$

$$(\textsc{Let1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\,\lambda\,\mathbf{k} : (\mathsf{B}\,[\mathsf{e}_1/\mathsf{x}])^{+} \to \alpha.\,\mathsf{e}_1{}^{\div}\ @\,\alpha\ (\lambda\,\mathbf{x} : \mathsf{A}^{+}.\,\mathsf{e}_2{}^{\div}\ \alpha\ \mathbf{k})$$

$$(\textsc{Cons1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\ \lambda\,\mathbf{k} : (\mathsf{L}(\mathsf{suc}\ \mathsf{e}))^{+} \to \alpha.\ \mathsf{e}^{\div}\ @\,\mathbb{N}\ (\lambda\,\mathbf{v} : \mathbb{N}.\,\mathsf{e}_1{}^{\div}\ \alpha\ (\lambda\,\mathbf{v_1} : \mathbb{N}.\,\mathsf{e}_2{}^{\div}\ \alpha\ (\lambda\,\mathbf{v_2} : (\mathsf{L}(\mathsf{e}))^{+}.\,\mathbf{k}\ (::\ \mathbf{v}\ \mathbf{v_1}\ \mathbf{v_2}))))$$

$$(\textsc{MatchN1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\ \lambda\,\mathbf{k} : (\mathsf{A}\,[\mathsf{e}/\mathsf{x}])^{+} \to \alpha.\ \mathsf{e}^{\div}\ @\,\alpha\ (\lambda\,\mathbf{v} : \mathbb{N}.\,\mathbf{match}\ \mathbf{v}\ \mathbf{with}\ \mathbf{z} \to (\mathsf{e}_1{}^{\div}\ \alpha\ \mathbf{k}) \parallel \mathbf{suc}\ \mathbf{m} \to (\mathsf{e}_2{}^{\div}\ \alpha\ \mathbf{k}))$$

$$(\textsc{MatchL1})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\,\lambda\,\mathbf{k} : (\mathsf{A}\,[\mathsf{n}/\mathsf{x}])^{+} \to \alpha.\ \mathsf{e}^{\div}\ @\,\alpha\ (\lambda\,\mathbf{v} : (\mathsf{L}(\mathsf{n}))^{+}.\,\mathbf{match}\ \mathbf{v}\ \mathbf{with}\ [\,] \to (\mathsf{e}_1{}^{\div}\ \alpha\ \mathbf{k}) \parallel ::\ \mathbf{m}\ \mathbf{h}\ \mathbf{t} \to (\mathsf{e}_2{}^{\div}\ \alpha\ \mathbf{k}))$$

$$(\textsc{Shift2})\ \overset{\div}{\rightsquigarrow} \lambda\,\mathbf{k} : \mathsf{A}^{+} \to \alpha^{+}.\,(\mathsf{e}^{\div}\,[\lambda\,\mathbf{v} : \mathsf{A}^{+}.\,\lambda\,\alpha' : *.\,\lambda\,\mathbf{k}' : (\alpha^{+} \to \alpha').\,\mathbf{k}'\ (\mathbf{k}\ \mathbf{v})/\mathbf{c}])\ (\lambda\,\mathbf{x} : \mathsf{B}^{+}.\,\mathbf{x})$$

$$(\textsc{Reset2})\ \overset{\div}{\rightsquigarrow} \lambda\,\alpha : *.\,\lambda\,\mathbf{k} : \mathsf{A}^{+} \to \alpha.\,\mathbf{k}\ (\mathsf{e}^{\div}\ (\lambda\,\mathbf{x} : \mathsf{B}^{+}.\,\mathbf{x}))$$

$$(\textsc{Conv1})\ \overset{\div}{\rightsquigarrow}\ \mathsf{e}^{\div} \qquad (\textsc{Conv2})\ \overset{\div}{\rightsquigarrow}\ \mathsf{e}^{\div}$$

図 8. 項の変換（抜粋）

これらの関数は、「状態を受け取ったら、答えを返す」という形の文脈の中で呼ばれることを前提とする。get は文脈に現在の状態の値（1 つ目の s）を返し、その後の計算を現在の状態（2 つ目の s）で実行する。一方、tick は文脈に () を返し、その後の計算を新しい状態 suc s で実行する。これらの関数を使った (3) は [0; 1] というリストを返す（ただし、$e_1; e_2$ は let $\_ = e_1$ in e_2 を表す）。

(3) $\langle \mathsf{let\ l} = :: 1\ (\mathsf{get}\ ())\ (:: 0\ (\mathsf{tick}\ (); \mathsf{get}\ ())\ [\,])\ \mathsf{in}\ \lambda \mathsf{s} : \mathbb{N}.\, \mathsf{l} \rangle\ 0$

上のプログラムは、reset の外側で渡されている 0 を初期状態として、let 文を計算する。その中で状態に 2 度アクセスしており、get はその度ごとに異なる値を返している。このような動作は Coq や Agda ではみられない。$\lambda_{\Pi}^{s/r}$ で (3) が型付け可能であるのは、状態の値がリストの要素として使われており、リスト型のインデックスとして型に現れないからである。

次に、answer type の変更を伴う例を見てみよう。以下に定義する prefix-prod は、自然数のリストを受け取り、そのプリフィックスの積からなるリストを返す [5]。

$$\begin{aligned} \mathsf{visit} &\stackrel{\mathrm{def}}{\equiv} \mathsf{fix\ f} : (\Pi\, \mathsf{n} : \mathbb{N}.\, \Pi\, \alpha : *.\, \alpha \parallel (\mathsf{L(n)} \to \mathbb{N} \parallel \mathbb{N} \parallel \mathsf{L(n)}) \parallel \alpha).\, \mathsf{n} : \mathbb{N}.\, \lambda \mathsf{l} : \mathsf{L(n)}. \\ &\qquad \mathsf{match\ l\ with}\ [\,] \to \mathcal{S}\mathsf{k}.\, [\,] \parallel :: \mathsf{m\ h\ t} \to \mathsf{m} * \mathcal{S}\mathsf{k}.\, :: \mathsf{m}\ (\mathsf{k}\ 1)\ \langle \mathsf{k}\ (\mathsf{f\ m\ t}) \rangle \\ \mathsf{prefix\text{-}prod} &\stackrel{\mathrm{def}}{\equiv} \lambda \mathsf{n} : \mathbb{N}.\, \lambda \mathsf{l} : \mathsf{L(n)}.\, \langle \mathsf{visit\ n\ l} \rangle \end{aligned}$$

prefix-prod の引数が $[e_1; e_2; \ldots; e_n]$ という形のリストであった場合、i 回目に呼ばれる visit の shift が切り取る継続は、「受け取った自然数に e_1 から e_i までをかけ合わせる」という計算となる。この継続が複数回使われている（つまり、文脈に異なる値が返されている）ため、上の関数は非決定的な振る舞いをもつといえる。引数のリストが空になったら、かけ算を行う計算を破棄し、空のリストを返す。このような動作から、visit は自然数 n と長さが n のリストを受け取ると、answer type を $\mathbb{N}$ から L(n) に変化させることが分かる。単純型付きの言語で実装した場合と比べて、上の関数は実行前後においてリストの長さが変化しないというインバリアントをもつ。例として、prefix-prod に 3 と [1; 2; 3] を渡すと、$[1; 1*2; 1*2*3] = [1; 2; 6]$ が返される。

構文を一般的な帰納的データ型で拡張した場合、より実用的なプログラムを実装できるようになる。たとえば、依存性をもつ帰納的な型を使うと、正しく型付けされた項のみを生成する対象言語を定義できるが [3]、shift/reset があれば、そのような言語に対して、例外処理をサポートするインタプリタを直接形式で書くことができる。具体的には、例外が発生するケースで shift を使って継続を破棄し、直接エラーを返す。shift/reset を用いずに、例外の伝搬が起きないインタプリタを実装するためには、プログラム全体を CPS あるいはモナド形式で書く必要がある。

7 shift/reset の論理的な解釈

ここまでの結果から、型に現れる項を適切に制限すれば、定理証明支援系の中で shift/reset を使うことは可能であると考えられる。その場合、たとえばバックトラックを使った証明を、（CPS ではなく）直接形式で書くことができるようになる。しかし、shift/reset をもつ定理証明支援系は、どのような論理体系となるだろうか。本節では、shift/reset を使ったプログラムにおいて、どのような項を証明とみなすべきか、また、これらの演算子の導入によって、論理体系の強さが変化するかどうかについて議論する。

7.1 証明項と proof transformer

pure な λ 計算において、$\Gamma \vdash e : A$ という型判断は、「仮定 Γ のもとで、e は A の証明項である」と解釈される。$\lambda_{\Pi}^{s/r}$ は、$\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ と $\Gamma; \alpha \vdash \mathsf{e} : \mathsf{A}; \beta$ の 2 種類の型判断をもつが、後者は 2 つの answer type α, β に言及している。このような型判断をもつ項は、証明においてどのような役割をもつだろうか。2 つの answer type のうち、α は e の実行が可能である文脈の返す型、β はその

ような文脈の中で e を実行した結果の型である。具体例をみてみよう。e を `shift` 節 $\mathcal{S}\mathsf{k}.\,\mathsf{e}'$ とし、その周りを pure な文脈 F が取り囲んでいるとする。この場合、$\langle \mathsf{F}[\mathcal{S}\mathsf{k}.\,\mathsf{e}'] \rangle$ の導出は以下のような形となる：

$$\dfrac{\begin{array}{c}\Gamma;\,\alpha \vdash \mathcal{S}\mathsf{k}.\,\mathsf{e}' : \mathsf{A};\,\beta \\ \vdots \\ \Gamma;\,\alpha \vdash \mathsf{F}[\mathcal{S}\mathsf{k}.\,\mathsf{e}'] : \alpha;\,\beta\end{array}}{\Gamma \vdash_p \langle \mathsf{F}[\mathcal{S}\mathsf{k}.\,\mathsf{e}'] \rangle : \beta}\ (\textsc{Reset2})$$

(Reset2) の前提は、answer type を無視すれば、$\mathsf{F}[\mathcal{S}\mathsf{k}.\,\mathsf{e}']$ が α の証明であることを表している。一方、結論部分は、$\langle \mathsf{F}[\mathcal{S}\mathsf{k}.\,\mathsf{e}'] \rangle$ が β の証明であることを意味する。すると、上の導出において、$\mathcal{S}\mathsf{k}.\,\mathsf{e}'$ は「α の証明を作る文脈 F に囲まれたら、それを β の証明に変換する」という役割をもつ項として理解することができる。もう少し一般的にいうと、導出 $\Gamma;\,\alpha \vdash \mathsf{e} : \mathsf{A};\,\beta$ が得られたとき、e は証明ではなく、「α の証明を β の証明に変換する proof transformer」として捉えることができる。

pure でない項のはたらきは、CPS 変換を通して理解することもできる。5.4 節の結果より、$\Gamma;\,\alpha \vdash \mathsf{e} : \mathsf{A};\,\beta$ を CPS 変換した結果 $\mathsf{e}^{\div}$ は $(\mathsf{A}^+ \to \alpha^+) \to \beta^+$ という型をもつが、これは $\mathsf{e}^{\div}$ が「α^+ の証明を作る継続を受け取ったら、β^+ の証明を返す」という役割をもつことを意味する。CPS 変換前の言語では、どのような文脈に囲まれるかを考えるが、変換後の言語では、どのような継続を受け取るかを考える。また、CPS 変換後の言語では、すべての項が $\mathbf{\Gamma \vdash e : A}$ という通常の導出をもつ。このような導出をもつ項は、証明として理解される。つまり、CPS 変換は、$\lambda_{\Pi}^{s/r}$ の項を直観主義論理の証明項へと変換している。ただし、変換後の型が表す命題は、ある種の二重否定が付いたものであり、もとの命題そのものを表しているわけではない。

2 つの answer type に言及する導出に加えて、$\lambda_{\Pi}^{s/r}$ はもう 1 つ、通常の λ 計算にはない要素をもつ。$\lambda_{\Pi}^{s/r}$ の関数は、本体が pure であるかどうかによって、2 種類の異なる型が付くことを思い出そう。このうち、本体が pure である場合の型 $\Pi \mathsf{x} : \mathsf{A}.\,\Pi\,\alpha : *.\,\alpha \,\|\, \mathsf{B} \,\|\, \alpha$ は、通常の関数型 $A \to B$ に対応しており、このような型をもつ項は、「A ならば B」という含意の証明として理解することができる。では、本体が pure でない関数型 $\Pi \mathsf{x} : \mathsf{A}.\,\alpha \,\|\, \mathsf{B} \,\|\, \beta$ はどのように理解したら良いだろうか。この型は特定の answer type に言及しており、関数の本体を実行するときに、α 型を返す文脈に囲まれることを要求する。このような型は含意として理解されないだけでなく、そもそも対応する論理的な解釈をもたない。実際、この型をもつ関数は、A の証明を受け取ったら、α の証明を β の証明に変換する proof transformer を返す。返されるものが証明ではないため、このような関数は証明として理解されるべきではない。

これらの議論から、$\lambda_{\Pi}^{s/r}$ の項 e が証明とみなされるのは、$\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ が導出可能であり、かつ A が $\Pi \mathsf{x} : \mathsf{A}'.\,\alpha \,\|\, \mathsf{B} \,\|\, \beta$ という形の型を含まない場合となる。なお、以降ではこの特殊な関数型を含まないような型を「pure な型」とよぶことにする。

7.2 論理体系の強さ

継続演算子の導入は、言語の論理的な表現力を変化させる場合がある。たとえば、Scheme の `call/cc` は、パース則 $((A \to B) \to A) \to A$ に対応する型をもつ [14]。パース則は、二重否定除去則 $((A \to \bot) \to \bot) \to A$ や排中律 $A \vee \neg A$ を含意するため [15]、`call/cc` をもつ言語は古典論理の体系としてみなされる。実際、`call/cc` を使うと、バックトラックを伴うプログラムを実装することができるが、このようなプログラムは背理法を使った古典論理の証明に対応している。

では、`shift`/`reset` の導入は論理体系にどのような影響を与えるだろうか。Ilik [18] は、answer type の変更を許さない体系における `shift`/`reset` の論理的な強さについて議論している。Ilik の体系において、関数型は $\Pi\, x : A.\, B$ という形であり、answer type に言及しない。`shift` は二重否定除去に相当する型規則をもち、`reset` は古典論理の推論が許される範囲を定めるものとされる。こ

のような体系で証明できる定理の例として、Ilik は $\forall x.\, \neg\neg A(x) \to \neg\neg(\forall x.\, A(x))$ (double-negation shift; DNS) の証明項を示している：

(4) $\lambda\, a : \forall x.\, \neg\neg A(x).\, \lambda\, b : \neg(\forall x.\, A(x)).\, \langle b\ (\lambda\, x.\, \mathcal{S}k.\, a\ x\ k)\rangle$

名前の通り、DNS は、全称量化の中にある二重否定を外に移動させる。この定理は、直観主義論理で証明できないことが知られているが、上の項では、$k : \neg A(x)$ という仮定のもとで $a\ x\ k : \bot$ を導出し、`shift` によって $A(x)$ を導くことで、この定理を証明している。

DNS を $\lambda_{\Pi}^{s/r}$ で導出することはできるだろうか。7.1 節の議論により、「$\lambda_{\Pi}^{s/r}$ で命題 P が証明できる」といえるのは、P に含まれる含意がすべて $\Pi\, x : A.\, \Pi\, \alpha : *.\, \alpha \,\|\, B \,\|\, \alpha$ という形であり、かつ型 P をもつ項が pure な項として導出されるときのみである。(4) と同じ形の項を考えてみよう。(4) は項全体が 1 つの λ 抽象であるため、pure である。したがって、これに pure な型を与えられれば、(4) を DNS の証明としてみなすことができる。ここで、2 つ目の引数 b の型を考える。b は $\lambda\, x.\, \mathcal{S}k.\, a\ x\ k$ という関数に対して適用されているため、b のドメインはこの関数の型と一致しなければならない。この関数の本体は `shift` 節であり、...; $\bot \vdash \mathcal{S}\mathsf{k}.\, \mathsf{a}\ \mathsf{x}\ \mathsf{k} : \mathsf{A(x)}$; $\bot$ という導出をもつ。λ 抽象の型規則 (Abs2) により、b に渡される関数の型は $\Pi\, \mathsf{x} : \_.\, \bot \,\|\, \mathsf{A(x)} \,\|\, \bot$ となる。これは論理的な解釈をもたない型である。この型が (4) 全体の型に現れるため、(4) は証明として捉えられるべきではない。したがって、(4) と同じ形をもつ項は、$\lambda_{\Pi}^{s/r}$ において DNS の証明とみなされない。

上の議論は、$\lambda_{\Pi}^{s/r}$ で DNS が証明できる可能性を排除しないが、型が完全に pure であるという条件は、証明項を作る際に強い制約となる。特に、この条件は否定の型をもつ関数の本体が pure であること、つまり、本体が `reset` で囲まれていない `shift` を含まないことを要求するが、これによって、証明の中で背理法を用いることが難しくなる。そこで、本研究は以下の予想を立てる：

予想 (`shift`/`reset` の論理的な強さ).

1. `shift`/`reset` を使って証明できるのは、直観主義論理の定理のみである。
2. $\Gamma \vdash_p \mathsf{e} : \mathsf{A}$ かつ A が pure であるとき、e の正規形は（存在すれば）`shift` を含まない。

予想 1 の裏付けとして、answer type の変更を伴わない `shift`/`reset` 付きの体系において、「`shift` の型規則を使えるのは、以降の導出でそれを限定する `reset` 規則が使われる場合のみ」とすると、`shift` と `reset` のペアを使った導出は、直観主義論理の規則のみで導出できることが知られている [20]。予想 2 を示すためには、`shift`/`reset` に対応する正規化アルゴリズムが必要となる。これに関連して、対馬ら [34] は `shift`/`reset` をもつ単純型付き λ 計算の TDPE (Type-Directed Partial Evaluation) を与えている。TDPE は、型の情報を用いて正規形を求める手法であり [12]、$\lambda_{\Pi}^{s/r}$ に対する（正当性が保証された）TDPE が定義されれば、予想を示すことができると考えられる。

8 関連研究

依存型をもつ体系への継続演算子の導入は、Herbelin [16, 17] によって考えられている。Herbelin は、Σ 型と等号型をもつ体系の中で `call/cc` を無制限に使用すると矛盾が導かれることを示し、型が依存し得る項を、「簡約時に `call/cc` の実行を伴わないもの」に制限した。この制約は、型に現れる項が pure であることを保証する。このように型の依存性を制限することで、Herbelin は一種の選択公理が成り立つ古典論理の体系を築いた。同じ制約を使って、Miquey [23] は $\lambda\mu\tilde{\mu}$-計算 [11] を依存型で拡張し、その無矛盾性を示している。

継続演算子のかわりに、effect を導入することによって、依存型付き言語に動的な振る舞いをもたせるという研究も行われている。Swamy ら [28] の F* は、実用的な定理証明支援系として開発されている言語であり、例外や状態などの effect をプリミティブな要素としてもつ。F* も本研究

と同様に、型の依存性を pure な項に制限しているが、高速なプログラム検証を可能にするヒープの実装や、安全な分散プログラミングなど、さまざまな応用が実現されている [27, 28]。

より理論的な研究として、Ahman ら [1] と Vákár [32] は一般的な effect をもつ依存型付き言語を設計している。両者とも Levy の Call-By-Push-Value (CBPV) [22] に従い、項を「計算」と「値」に分けて定義しており、型が依存できる項を値のみに制限している。CBPV における値は、簡約不可能な値に加えて、簡約した際に effect が発生しない項も含むため、pure な項に対応している。

継続演算子の論理的な解釈は、Thielecke [31] によって議論されている。Thielecke は、ジャンプを表す演算子と、ジャンプ先をラベル付けする演算子をもつ λ 計算を定義した。この計算体系に対し、Thielecke は 3 種類の異なる CPS 変換を通して意味を与え、変換の定義によってこの体系が古典論理となるか、直観主義論理にとどまるかが決まることを示した。Thielecke の体系において、型判断は $\Gamma \vdash e : A, C$ という形をもつ。A は e が正常終了した場合の型、C はジャンプが実行された場合の型を表しており、後者は $\mathsf{\Gamma}; \alpha \vdash \mathsf{e} : \mathsf{A}; \beta$ の β に対応している。Thielecke は、上の型判断を「A または C」と解釈しているが、$\lambda_{\Pi}^{s/r}$ における pure でない項の型判断は、特定の型をもつ文脈を要求するため、「A または β」という選言として解釈することはできない。

9 まとめと今後の課題

本研究では、`shift`/`reset` をもつ依存型付き言語を設計した。`shift` を含む計算を無制限に許すと、プログラムの型が静的に定まらなくなるため、型に現れる項を pure なものに制限した。これにより、健全な型システムと、型を保存する CPS 変換を得ることができた。また、`shift`/`reset` を使ったプログラムの論理的な解釈について考察した。証明として理解されるのは、pure な項のうち、型も pure であるようなもののみであり、証明が存在する命題は、直観主義論理の定理であるという予想を立てた。

今後の課題として、まず 7.2 節で立てた予想の証明が挙げられる。また、言語の要素として、ユーザによって定義された帰納的データ型を追加し、高度なパターンマッチを実現させたいと考えている。$\lambda_{\Pi}^{s/r}$ ではパターンマッチの構文を固定しているが、依存型付きの言語では、マッチされる項の型によって可能なパターンの候補が絞られることがある。たとえば、$\mathsf{L(suc\ z)}$ 型のリストは必ず $\mathsf{::\ m\ h\ t}$ という形にマッチされるため、Coq や Agda では $\mathsf{[\,]}$ のケースが省略される。型の依存性の制限を守れば、$\lambda_{\Pi}^{s/r}$ でこのような絞り込みをサポートするのは可能であると考えられるが、CPS 変換を定義する際に、変換が可能なパターンを保存するという性質が必要になる。さらに、$\lambda_{\Pi}^{s/r}$ は任意の再帰関数を許すが、今後は停止性を判断できるように拡張し、型検査における無限ループを回避したいと考えている。この拡張を行う際にも、パターンの絞り込みと同様に、CPS 変換が停止性を保存することの証明が必要となる。

もう 1 つの方向性は、名前呼びの意味論に基づいた言語を設計する、というものである。副作用を議論する際には、値呼びの意味論を用いることが多いが、名前呼びの `shift`/`reset` に対する実行規則や型システムも考えられている [29]。このような演算子を扱う場合にも、型の依存性に関して、$\lambda_{\Pi}^{s/r}$ と同じ制約、すなわち、「型に現れる項は pure なもののみ」という制約が必要であると考えられる。一方、名前呼びの言語に対する CPS 変換と型保存の証明は、値呼びの場合に比べて単純になる。本稿で定義した CPS 変換において、$@$ を用いた関数適用や、[T-Cont] の提供する非自明な同値性が必要であるのは、ある $\lambda_{\Pi}^{s/r}$ の項 e の部分項 $\mathsf{e'}$ が e の型に現れ、なおかつ e を評価するときに $\mathsf{e'}$ を評価しなければならない場合である（たとえば、$\mathsf{e} = \mathsf{e}_1\ \mathsf{e'}$）。しかし、名前呼びの言語では、部分項の実行が値呼びの言語ほど頻繁に起きないため、これらを必要とする場面が少なくなる。実際、Bowman ら [9] の CPS 変換も、名前呼びの方が単純になっている。

謝辞

テクニカルな議論やプレゼンテーション面のアドバイスなど、多くの建設的なコメントをくださった査読者の皆さまに感謝いたします。本研究は JSPS 科研費 15K00090 の助成を受けたものです。

参考文献

[1] Danel Ahman, Neil Ghani, and Gordon D. Plotkin. *Dependent Types and Fibred Computational Effects*, pages 36–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[2] Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444, September 2011.

[3] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, CSL '99, pages 453–468, London, UK, 1999. Springer-Verlag.

[4] Kenichi Asai. Logical relations for call-by-value delimited continuations. *Trends in Functional Programming*, 6:63–78, 2005.

[5] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS '07, pages 239–254, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Kenichi Asai and Oleg Kiselyov. Introduction to programming with shift and reset. In *ACM SIGPLAN Continuation Workshop 2011*, 2011.

[7] Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, September 1999.

[8] Gilles Barthe and Tarmo Uustalu. Cps translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 131–142, New York, NY, USA, 2002. ACM.

[9] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving CPS translation of Σ and Π types is not not possible. In *Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '18, January 2018.

[10] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[11] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ACM sigplan notices*, volume 35, pages 233–243. ACM, 2000.

[12] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 242–257, New York, NY, USA, 1996. ACM.

[13] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

[14] Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[15] Mauricio Guillermo and Alexandre Miquel. Specifying peirce's law in classical realizability. *Mathematical Structures in Computer Science*, 26(7):1269–1303, 2016.

[16] Hugo Herbelin. On the degeneracy of Σ-types in presence of computational classical logic. In *International Conference on Typed Lambda Calculi and Applications*, TLCA '05, pages 209–220. Springer, 2005.

[17] Hugo Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 365–374. IEEE Computer Society, 2012.

[18] Danko Ilik. Delimited control operators prove double-negation shift. *Annals of Pure and Applied logic*, 163(11):1549–1559, 2012.

[19] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 275–286, New York, NY, USA, 2010. ACM.

[20] Yukiyoshi Kameyama. Towards logical understanding of delimited continuations. In *Continuations Workshop*, pages 27–33, 2001.

[21] Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher Order Symbolic Computation*, 20(4):431–460, December 2007.

[22] Paul Blain Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer Science & Business Media, 2012.

[23] Étienne Miquey. A classical sequent calculus with dependent types. In *European Symposium on Programming*, pages 777–803. Springer, 2017.

[24] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[25] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[26] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[27] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[28] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, New York, NY, USA, 2016. ACM.

[29] Asami Tanaka and Yukiyoshi Kameyama. A call-by-name CPS hierarchy. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, pages 260–274, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[30] The Coq Development Team. The Coq proof assistant reference manual, November 2016.

[31] Hayo Thielecke. Comparing control constructs by typing double-barrelled cps transforms. *Higher-Order and Symbolic Computation*, 15(2):141–160, 2002.

[32] Matthijs Vákár. *In Search of Effectful Dependent Types*. PhD thesis, Oxford University, 2017.

[33] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM, ACM.

[34] 対馬かなえ, 浅井健一. 限定継続のための TDPE に向けて. 第 12 回プログラミングおよびプログラミング言語ワークショップ予稿集, March 2010.