限定継続のための TDPE に向けて

対馬 かなえ
お茶の水女子大学
tsushima.kanae@is.ocha.ac.jp

浅井 健一
お茶の水女子大学
asai@is.ocha.ac.jp

Abstract

TDPE (Type-directed partial evaluation) は部分評価の一種であり、プログラムとその型を用いることによって、プログラム内部の構造に触れずに部分評価を進めることが出来る。そのため、他の部分評価器と比較して高速である。

本研究は TDPE の対象言語に限定継続の一種である `shift/reset` を導入し、その正当性を証明することを目的にしている。

まず、`shift/reset` と CPS 変換の関係性に着目して、CPS 変換後の型を持つプログラムが TDPE でどのように部分評価されるかを観察し、それを直接形式に戻すことで `shift/reset` を含む TDPE を導いた。また、`shift/reset` の影響を受けない部分については従来の TDPE と同じように部分評価されるようにした。

次に call-by-name の TDPE の正当性を示し、今後の他の TDPE の証明への展望を示す。

1 導入

Type-directed partial evaluation (TDPE) は Danvy によって提案された Normalization by evaluation の一例である [4]。TDPE に入力として項を与えると、その項に η-expanding を施し、結果として正規形へ部分評価された項を出力する。

他の部分評価と比べた特徴として、TDPE は入力された項を直接実行するので、項の構造に触れることなく部分評価を行うことが出来る。そのため、他の部分評価に比べて高速であり、速さという点で魅力的なプログラム最適化の一つである。

他方で、コントロールオペレータは CPS 変換なしに継続を扱うことが出来る便利なツールである。限定継続のための命令としては `control/prompt`、`shift/reset` など様々なコントロールオペレータがあるが、本研究では `shift/reset` [6] を用いた。その理由として、`shift/reset` は CPS と密接な関係があり、明確な意味論を持っているため、非決定性プログラミング [6] 、部分評価における let-insertion [12] 、typed printf の記述 [2] など多くの研究に使われているからである。

以前の論文 [13] では、従来の λ 計算のための TDPE を限定継続の命令 `shift/reset` を使えるように拡張し、TDPE の表現力を広げることを目的とした。概要としては、CPS の型を持つ項が従来の call-by-value の TDPE でどのように扱われるのかを観察し、それを直接形式に戻すことによって、限定継続のための TDPE を求めた。

本論文では、その論文で求めた限定継続のための TDPE を改良し、`shift/reset` の影響を受けない部分については従来の TDPE と同じように部分評価されるようにした。また、この限定継続のための TDPE の正当性への足掛かりとなる、call-by-name の TDPE の正当性を証明し、その後の展望を示す。

次の節では `shift/reset` について簡単に説明する。3 節で Danvy の TDPE を、4 節で call-by-value の TDPE を振り返った後、5 節で以前の論文からの `shift/reset` のための TDPE を改良したものを説明する。6 節で call-by-name の TDPE の正当性を証明してその先の証明についての展望を示し、7 節で結論を述べる。

2 shift/reset

`shift/reset` は限定継続を扱うための命令である。`shift` は Scheme の `call/cc` のように現在の継続を切り取るが、`reset` に囲まれた一部分の継続

のみを切り取るため、範囲が限定されない Scheme の `call/cc` とは異なる。

具体的には、`(shift k M)` は現在の継続を切り取って、それを `k` に束縛し、`M` を空の継続で実行する。`(reset M)` は `M` を空の継続で実行する。
例えば、次の式を考えてみる。

```
(+ 1 (reset (+ 2 (shift k (k (k 3))))))
⇝ (+ 1 (reset (+ 2 (+ 2 3))))
⇝ (+ 1 (reset 7))
⇝ (+ 1 7)
⇝ 8
```

一行目で切り取られる継続は、`shift` 式が `reset` で囲まれているため `(+ 2 [])` である。この式では切り取られた継続 `k` は `(k (k 3))` のように `shift` 式の中で二度使われている。`reset` の本体の式が実行されて value になったとき、`reset` 式は本体の式の value へ評価される。よって、`(reset 7)` は `7` になる。

正確な `shift/reset` の意味論については CPS 変換を使うことで定義されている。[6]

3 λ 計算の TDPE

部分評価 [10] とはプログラムの実行時間短縮のために、与えられたプログラムの static な部分をあらかじめ出来るだけ実行しておくプログラム最適化の一種である。

Type-directed partial evaluation (TDPE) は部分評価の一種であるが、通常の部分評価とは二つの点で異なる。まず、TDPE は式の型の構造に着目した部分評価である。なので、TDPE は実行する時に式だけでなく、その式の型も必要とする。次に、TDPE は式の構造を操作することなく、代わりに受け取ったプログラムを実際に実行することによって結果を求める。よって、TDPE が部分評価を実行するためにかかる時間は普通のインタプリタが基になっている部分評価に比べて短い。これはコンパイルされたプログラムの方がインタプリタで実行するより速いのと似たような理由である。

3.1 構文

図 1 は、Danvy の Type-directed partial evaluation であり、これは λ 計算を定数とペアで拡張したものを対象としている。また、この TDPE は Scheme のような datatype constructors を持つ call-by-value の言語で表現されていると仮定している。

$$
\begin{array}{rcl}
t \in \text{Type} & ::= & b \mid t_1 \to t_2 \mid t_1 \times t_2 \\
v \in \text{Value} & ::= & c \mid x \mid \overline{\lambda} x.v \mid v_0 \mathbin{\overline{@}} v_1 \mid \\
 & & \overline{pair}(v_1, v_2) \mid \overline{fst}\ v \mid \overline{snd}\ v \\
e \in \text{Expr} & ::= & c \mid x \mid \underline{\lambda} x.e \mid e_0 \mathbin{\underline{@}} e_1 \\
 & & \underline{pair}(e_1, e_2) \mid \underline{fst}\ e \mid \underline{snd}\ e \\
\text{reify} & = & \lambda t.\lambda v : t.{\downarrow^t} v \\
 & : & \text{Type} \to \text{Value} \to \text{TLT} \\
{\downarrow^b}\ v & = & v \\
{\downarrow^{t_1 \to t_2}}\ v & = & \underline{\lambda} x_1^{\diamond}.\ {\downarrow^{t_2}}\ (v \mathbin{\overline{@}} {\uparrow_{t_1}}\ x_1^{\diamond}) \\
{\downarrow^{t_1 \times t_2}}\ v & = & \underline{pair}({\downarrow^{t_1}}\ \overline{fst}\ v,\ {\downarrow^{t_2}}\ \overline{snd}\ v) \\
 & & \\
\text{reflect} & = & \lambda t.\lambda e : t.{\uparrow_t} e \\
 & : & \text{Type} \to \text{Expr} \to \text{TLT} \\
{\uparrow_b}\ e & = & e \\
{\uparrow_{t_1 \to t_2}}\ e & = & \overline{\lambda} v_1.\ {\uparrow_{t_2}}\ (e \mathbin{\underline{@}} {\downarrow^{t_1}}\ v_1) \\
{\uparrow_{t_1 \times t_2}}\ e & = & \overline{pair}({\uparrow_{t_1}}\ \underline{fst}\ e,\ {\uparrow_{t_2}}\ \underline{snd}\ e) \\
 & & \\
\text{residualize} & = & \text{statically-reduce} \circ \text{reify} \\
 & : & \text{Type} \to \text{Value} \to \text{Expr}
\end{array}
$$

図 1. Danvy の λ 計算のための Type-Directed Partial Evaluation[4]

型は基本型 (b)、関数型 ($t_1 \to t_2$)、積の型 ($t_1 \times t_2$) の三種である。オーバーラインは static であることを意味し、TDPE のコンテキストでは中身を調べていない compiled representation を意味する。逆にアンダーラインは dynamic であることを意味し、datatype constructors で表された式を意味する。TLT は Expr と Value の 2 レベルの項を持つ。上付き文字 $^{\diamond}$ がついた変数は、新しく作られた変数であることを意味している。

3.2 Reify と Reflect

TDPE は reify と reflect の二つの関数によって行われる。下向き矢印 ($\downarrow^t$) は reify と呼ばれ、型 t を使うことで、compiled representation を datatype に変換する。例えば、compiled representation f が関数型 $b \to b$ を持つとする。型が $b \to b$ なので、 f はある M に対して $\underline{\lambda} x_1.M$ という形を持っていることが分かる。その本体 M を求める

$$
\begin{aligned}
& \downarrow^{t_1\to((t_1\to t_2)\to t_2)} (\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x))) \\
= \;& \underline{\lambda}x_1.\downarrow^{(t_1\to t_2)\to t_2} ((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}(\uparrow_{t_1} x_1)) \\
= \;& \underline{\lambda}x_1.\downarrow^{(t_1\to t_2)\to t_2} ((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1) \\
= \;& \underline{\lambda}x_1.\underline{\lambda}x_2.\downarrow^{t_2} (((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1)\overline{@}(\uparrow_{t_1\to t_2} x_2)) \\
= \;& \underline{\lambda}x_1.\underline{\lambda}x_2.\downarrow^{t_2} ((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1)\overline{@}(\overline{\lambda}v_1.\uparrow_{t_2} (x_2\underline{@}(\downarrow^{t_1} v_1))) \\
= \;& \underline{\lambda}x_1.\underline{\lambda}x_2.\downarrow^{t_2} ((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1)\overline{@}(\overline{\lambda}v_1.\uparrow_{t_2} (x_2\underline{@}v_1)) \\
= \;& \underline{\lambda}x_1.\underline{\lambda}x_2.\downarrow^{t_2} ((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1)\overline{@}(\overline{\lambda}v_1.(x_2\underline{@}v_1)) \\
= \;& \underline{\lambda}x_1.\underline{\lambda}x_2.((\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))\overline{@}x_1)\overline{@}(\overline{\lambda}v_1.(x_2\underline{@}v_1)) \\
\rightsquigarrow \;& \underline{\lambda}x_1.\underline{\lambda}x_2.(\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x_1)))\overline{@}(\overline{\lambda}v_1.(x_2\underline{@}v_1)) \\
\rightsquigarrow \;& \underline{\lambda}x_1.\underline{\lambda}x_2.(\overline{\lambda}v_1.(x_2\underline{@}v_1))\overline{@}((\overline{\lambda}a.a)\overline{@}x_1) \\
\rightsquigarrow \;& \underline{\lambda}x_1.\underline{\lambda}x_2.(\overline{\lambda}v_1.(x_2\underline{@}v_1))\overline{@}x_1 \\
\rightsquigarrow \;& \underline{\lambda}x_1.\underline{\lambda}x_2.x_2\underline{@}x_1
\end{aligned}
$$

図 **2. 例:** $(\overline{\lambda}x.\overline{\lambda}f.(f\overline{@}((\overline{\lambda}a.a)\overline{@}x)))$ **of type** $t_1 \to ((t_1 \to t_2) \to t_2)$ **の TDPE**

には、f に新しく作った dynamic な変数 x_1 を適用してみれば良い。もし、f が完全に polymorphic な型を持つならば、f は x_1 の値を使うことはなく、安全に f に適用することが出来る。すると、f の本体を得ることが出来、この場合は x_1 となり、$\underline{\lambda}x_1.x_1$ を得ることが出来る。

上向き矢印 ($\uparrow_t$) は reflect と呼ばれる。これは reify の逆であり、datatype representation をコンパイルされた実行可能な値に変換する。

reify を使うことで、TDPE は完全に static な入力の項を、dynamic な項に変換する。入力の式や TDPE における全ての static な部分は reify (またその中での reflect) によって全て簡約され、結果として全て dynamic な項が得られる。

3.3 例

例として $(\overline{\lambda}x.\overline{\lambda}f.f\overline{@}((\overline{\lambda}a.a)\overline{@}x))$ という式を TDPE で実行することを考えてみる。この式の型は $t_1 \to ((t_1 \to t_2) \to t_2)$ である。

この項に対する TDPE の実行結果は図 2 に示した。入力の項は全て static な項として考える。$\downarrow^{t_1\to((t_1\to t_2)\to t_2)}$ を $(\overline{\lambda}x.\overline{\lambda}f.f\overline{@}((\overline{\lambda}a.a)\overline{@}x))$ に適用することで、2 レベルの項:

$$\underline{\lambda}x_1.\underline{\lambda}x_2.(\overline{\lambda}x.\overline{\lambda}f.f\overline{@}((\overline{\lambda}a.a)\overline{@}x))\overline{@}x_1\overline{@}\,\overline{\lambda}v_1.x_2\underline{@}v_1$$

を得ることが出来る。

この 2 レベルの項の意味について考えてみると、入力として与えた項の型が $t_1 \to ((t_1 \to t_2) \to t_2)$ という形なので、部分評価後にはある M に対して $\underline{\lambda}x_1.\underline{\lambda}x_2.M$ という形を持つことが分かる。

M を求めるには、入力の項に x_1 と $\overline{\lambda}v_1.x_2\underline{@}v_1$ という 2 つの引数を与える必要がある。なぜなら、x_2 は $t_1 \to t_2$ という型を持ち、関数として中で使われる必要があるため x_2 自身を適用することは出来ない。しかし、reflect を使ってそれを static な項に変換したら、つまり 2 レベル η 拡張した結果の項 $\overline{\lambda}v_1.x_2\underline{@}v_1$ なら適用することが出来るからである。

結果として得られた項の全ての static な部分を実行すると、残余の項として $\underline{\lambda}x_1.\underline{\lambda}x_2.x_2\underline{@}x_1$ を得ることが出来る。これは入力の項を部分評価した項になっていて、入力の項に存在していた $(\overline{\lambda}a.a)\overline{@}x$ という redex が評価されている。

4 call-by-value の TDPE

3 節での TDPE は call-by-name であった。例えば、3 節の TDPE で $(\text{int} \to \alpha) \to \text{int}$ という型を持つ式 $\overline{\lambda}x.((\overline{\lambda}y.1)\overline{@}(x\overline{@}2))$ を実行すると、結果は $(x\overline{@}2)$ という関数適用を捨てて、$(\underline{\lambda}x.1)$ となる。しかし、call-by-value では、関数適用を捨てることはできない。代わりに let 文として残す必要がある。

以前の論文 [13] で我々は、call-by-value の TDPE を導入するために state-based let-insertion を行うものを示した。ここでは、それを紹介するとともに shift/reset-based let-insertion を行う方法

$$
\begin{array}{rcl}
\text{reify} & = & \lambda t.\lambda v : t.{\downarrow^t} v \\
 & : & \text{Type} \to \text{Value} \to \text{TLT} \\
{\downarrow^b}\ v & = & v \\
{\downarrow^{t_1 \to t_2}}\ v & = & \underline{\lambda} x_1^{\diamond}.init; wrap({\downarrow^{t_2}}\ (v\ \overline{@}\ {\uparrow_{t_1}}\ x_1^{\diamond})) \\
{\downarrow^{t_1 \times t_2}}\ v & = & \underline{pair}({\downarrow^{t_1}}\ \overline{fst}\ v,\ {\downarrow^{t_2}}\ \overline{snd}\ v)
\end{array}
$$

$$
\begin{array}{rcl}
\text{reflect} & = & \lambda t.\lambda e : t.{\uparrow_t} e \\
 & : & \text{Type} \to \text{Expr} \to \text{TLT} \\
{\uparrow_b}\ e & = & e \\
{\uparrow_{t_1 \to t_2}}\ e & = & \overline{\lambda} v_1.\ {\uparrow_{t_2}}\ [e\ \underline{@}\ {\downarrow^{t_1}}\ v_1] \\
{\uparrow_{t_1 \times t_2}}\ e & = & \overline{pair}({\uparrow_{t_1}}\ \underline{fst}\ e,\ {\uparrow_{t_2}}\ \underline{snd}\ e)
\end{array}
$$

$$
\begin{array}{rcl}
\text{residualize} & = & \text{statically-reduce} \circ \text{reify} \\
 & : & \text{Type} \to \text{Value} \to \text{Expr}
\end{array}
$$

図 3. λ 計算のための call-by-value の TDPE

$$
\begin{array}{rcl}
init & = & wrapstack \leftarrow () :: wrapstack \\
\\
wrap(e) & = & let\ f :: rest\ = wrapstack\ in \\
 & & wrapstack \leftarrow rest; \\
 & & wrap\text{-}gen(f,\ e) \\
\\
[e] & = & let\ f :: rest\ = wrapstack\ in \\
 & & wrapstack \leftarrow ((g^{\diamond}, e) :: f) :: rest; \\
 & & g^{\diamond}
\end{array}
$$

$$
\begin{array}{l}
wrap\text{-}gen\ ([],\ body)\ =\ body \\
wrap\text{-}gen\ (lst,\ body) \\
\qquad = \ let\ (l, r) :: rest = lst\ in \\
\qquad\quad wrap\text{-}gen(rest,\ \underline{\langle}\ \underline{let}\ l = r\ \underline{in}\ body\ \underline{\rangle})
\end{array}
$$

図 4. state-based let-insertion の wrap, [...] and init の定義

$$
\begin{array}{rcl}
init & \equiv & ;\text{何もしない} \\
\\
wrap(e) & \equiv & \overline{\langle} e \overline{\rangle} \\
\\
[e] & \equiv & \overline{\mathcal{S}} k.\underline{\langle} \underline{let}\ x^{\diamond} = e\ \underline{in}\ k \overline{@} x^{\diamond} \underline{\rangle}
\end{array}
$$

図 5. shift/reset-based let-insertion の wrap, [...] and init の定義

についても触れる。なお、これらの実装はいずれも Filinski の実装 [9] に基づくものである。Filinski の実装では、例外や gensym、state-based let-insertion [11] などの全てのモナドは Scheme の native effects を使うことで実現されている。

4.1 Residualization

call-by-value の TDPE の構文は call-by-name の TDPE と同じなので図 1 に、call-by-value のための reify と reflect は図 3 に示した。

call-by-value の TDPE を実現するためには、先ほどの例では捨てられてしまったような dynamic な関数適用を dynamic な `let` 文として残す (let-insertion) 必要がある。let-insertion をするには state を使って行う方法と、shift/reset を使って行う方法がある。

let-insertion を行うには、図 3 で使用している [..]、*wrap*、*init* というオペレータを使えばよい。それぞれの役割として、[..] は `let` 文に残さなくてはならない dynamic な関数適用を登録するもの、*init* は (state-based let-insertion において) 状態を初期化するもの、*wrap* は登録された dynamic な関数を順次、`let` 文にして展開するものである。

state-based と shift/reset-based の let-insertion の違いはこれら 3 つのオペレータを state を使って実現するか、`shift/reset` を使って実現するかであって、どちらの方法でも可能である。図 4 に state-based の let-insertion の定義を、図 5 に shift/reset-based の let-insertion の定義を示す。shift/reset-based let-insertion では、e を特定のコンテキストの中で実行しなくてはならないので、wrap や [...] は関数ではなくマクロとして定義されている。

図 3 の *init*、*wrap*、[...] を無視すると、reify と reflect は Danvy の TDPE のように完全な対称性を持っている。

4.2 例

call-by-value の TDPE を使うと、この節のはじめの例 $\overline{\lambda} x.((\overline{\lambda} y.1)\overline{@}(x\overline{@}2))$ (型は $(\text{int} \to \alpha) \to \text{int}$) は $\underline{\lambda} x.\ \underline{let}\ y = x\underline{@}2\ \underline{in}\ 1$ へと部分評価される。dynamic な関数適用 $x\underline{@}2$ は捨てられずに、`let` 文

$$
\begin{array}{rcl}
t \in \text{Type} & ::= & b \mid t_1 \to t_2 \mid t_1/\alpha \to t_2/\beta \\
& & t_1 \times t_2 \\
v \in \text{Value} & ::= & c \mid x \mid \overline{\lambda} x.v \mid v_0 \overline{@} v_1 \mid \overline{\langle}\ v\ \overline{\rangle} \mid \overline{\mathcal{S}} k.v \\
& & \overline{pair}(v_1, v_2) \mid \overline{fst}\ v \mid \overline{snd}\ v \\
e \in \text{Expr} & ::= & c \mid x \mid \underline{\lambda} x.e \mid e_0 \underline{@} e_1 \mid \underline{\langle}\ e\ \underline{\rangle} \mid \underline{\mathcal{S}} k.e \\
& & \underline{pair}(e_1, e_2) \mid \underline{fst}\ e \mid \underline{snd}\ e
\end{array}
$$

図 **6. 構文: λ 計算と shift/reset**

として残され、正しく call-by-value で評価されていることが分かる。

5 限定継続のための TDPE

この節では `shift/reset` で拡張した TDPE について説明する。

5.1 構文

対象とする言語とその型は図 6 に示した。`shift/reset` が存在すると、関数型は普通の関数型 $t_1 \to t_2$ に加えて、4つ組の関数型 $t_1/\alpha \to t_2/\beta$ を持つ。それぞれの t_1 は普通の関数型と同じ引数の型であり、t_2 はその関数が返す式の型である。加えて、α と β はそれぞれ関数が呼ばれる前と後のコンテキストの answer type である。

4つ組の関数型が pure な場合、つまり α と β が polymorphic で同じであるような場合に普通の関数型で表すことにする。`shift/reset` を含む式の型は Danvy と Filinski [5] によって定義されており、今回使用する型システムはそれに pure な関数型 [3] を加えたものとする。

対象とする言語の項は、ラムダ計算に加えて `shift/reset` の式である。$\mathcal{S}k.\ v$ という項は `(shift k v)` を意味し、$\langle v \rangle$ という項は `(reset v)` を意味する。また、入力の項は完全に static であり、前述の型システムに基づいて型が付いていると仮定している。

5.2 Residualization

この節では、CPS の関数型を持つ項が call-by-value の TDPE でどのように評価されるかを観察し、それを直接形式に戻すことで4つ組の関数型 $t_1/\alpha \to t_2/\beta$ のための TDPE を求める方法について説明する。

まず、$t_1/\alpha \to t_2/\beta$ は CPS 変換で $(t_1 \times (t_2 \to \alpha)) \to \beta$ という型に対応している。そのため、CPS の関数型の項がどのように変換されるかは従来の call-by-value の TDPE で $\downarrow^{(t_1 \times (t_2 \to \alpha)) \to \beta}\ v$ と $\uparrow_{(t_1 \times (t_2 \to \alpha)) \to \beta}\ e$ を実行してみて、結果を観察すれば良い。これらの式を実際に実行してみると、下のようになる。

$$
\begin{array}{l}
\downarrow^{t_1 \times (t_2 \to \alpha) \to \beta} v = \\
\quad \underline{\lambda} p_1^\diamond.init; wrap(\downarrow^\beta (v\ \overline{@}\ \overline{pair}(\uparrow_{t_1} \underline{fst}\ p_1^\diamond, \\
\qquad\qquad \overline{\lambda} v_1.\ \uparrow_\alpha [(\underline{snd}\ p_1^\diamond\ \underline{@}\ \downarrow^{t_2} v_1)])))) \\
\uparrow_{t_1 \times (t_2 \to \alpha) \to \beta} e = \\
\quad \overline{\lambda} p_1.\ \uparrow_\beta [(e\ \underline{@}\ \underline{pair}(\downarrow^{t_1} \overline{fst}\ p_1, \\
\qquad\qquad \underline{\lambda} x_1^\diamond.init; wrap(\downarrow^\alpha (\overline{snd}\ p_1\ \overline{@}\ \uparrow_{t_2} x_1^\diamond))))]
\end{array}
$$

上の2つの式において、最も外側の抽象 $(\underline{\lambda} p_1^\diamond ...)$ と $(\overline{\lambda} p_1 ...)$ は積の型を持つ引数を受け取る。その積にパターンマッチを適用すると、上の式を次のように書き直すことが出来る。

$$
\begin{array}{l}
\downarrow^{t_1 \times (t_2 \to \alpha) \to \beta} v = \\
\quad \underline{\lambda}(x_1^\diamond,\ k_1^\diamond).init; wrap(\downarrow^\beta (v\ \overline{@}\ \overline{pair}(\uparrow_{t_1} x_1^\diamond, \\
\qquad\qquad \overline{\lambda} v_1.\ \uparrow_\alpha [(k_1^\diamond\ \underline{@}\ \downarrow^{t_2} v_1)]))) \\
\uparrow_{t_1 \times (t_2 \to \alpha) \to \beta} e = \\
\quad \overline{\lambda}(v_1,\ k_1).\ \uparrow_\beta [(e \underline{@}\ \underline{pair}\ (\downarrow^{t_1} v_1, \\
\qquad\qquad \underline{\lambda} x_1^\diamond.init; wrap(\ \downarrow^\alpha (k_1\ \overline{@}\ \uparrow_{t_2} x_1^\diamond)))))]
\end{array}
$$

こうして CPS の関数型を持つ項の TDPE について考えたところで、まず reify ($\downarrow$) を直接形式に戻すことについて考えてみる。CPS 形式では $\underline{\lambda}(x_1^\diamond,\ k_1^\diamond)$ は引数と継続を受け取る。直接形式では $x_1^\diamond$ が引数を受け取ることに対応し、$k_1^\diamond$ が継続を切り取ることに対応する。よって、直接形式に書き直すと、ある M に対して $\underline{\lambda} x_1^\diamond.\underline{\mathcal{S}} k_1^\diamond.M$ となる。

M を求めようとしたところで、v は $\uparrow_{t_1} x_1^\diamond$ に $\overline{\lambda} v_1.\ \uparrow_\alpha [(k_1^\diamond\ \underline{@}\ \downarrow^{t_2} v_1)]$ という継続のもとで適用されていることが分かる。直接形式では、これは $\overline{\lambda} v_1.\ \uparrow_\alpha [(k_1^\diamond\ \underline{@}\ \downarrow^{t_2} v_1)]$ というコンテキストのもとで、$v \overline{@}\ \uparrow_{t_1} x_1^\diamond$ を実行することと一致する。
よって $\downarrow^{(t_1 \times (t_2 \to \alpha)) \to \beta} v$ は

$$
\begin{array}{l}
\underline{\lambda} x_1^\diamond.\underline{\mathcal{S}} k_1^\diamond.init; \\
wrap(\downarrow^\beta (\overline{\lambda} v_1.\ \uparrow_\alpha [(k_1^\diamond\ \underline{@}\ \downarrow^{t_2} v_1)]) \overline{@} (v\ \overline{@}\ \uparrow_{t_1} x_1^\diamond))
\end{array}
$$

となりそうである。

しかし、これはコンテキストの範囲が定まっていないため、うまく動かない。v は static な

$$
\begin{array}{rcl}
\mathrm{reify} & = & \lambda t.\lambda v : t.{\downarrow^t} v \\
 & : & \mathrm{Type} \to \mathrm{Expr} \to \mathrm{TLT} \\
{\downarrow^b}\ v & = & v \\
{\downarrow^{t_1 \to t_2}}\ v & = & \underline{\lambda} x_1^{\diamond}.\ {\downarrow^{t_2}}\ (v\ \overline{@}\ {\uparrow_{t_1}}\ x_1^{\diamond}) \\
{\downarrow^{t_1/\alpha \to t_2/\beta}}\ v & = & \underline{\lambda} x_1^{\diamond}.\underline{\mathcal{S}} k_1^{\diamond}.\ \mathit{init};\ \mathit{wrap}({\downarrow^{\beta}}\ \overline{\langle}\ \overline{\mathrm{let}}\ v_1 = v\ \overline{@}\ {\uparrow_{t_1}}\ x_1^{\diamond}\ \ \overline{\mathrm{in}}\ ({\uparrow_{\alpha}}\ [k_1^{\diamond}\ \underline{@}\ {\downarrow^{t_2}}\ v_1])\overline{\rangle}) \\
{\downarrow^{t_1 \times t_2}}\ v & = & \underline{pair}({\downarrow^{t_1}}\ \overline{fst}\ v,\ {\downarrow^{t_2}}\ \overline{snd}\ v)
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{reflect} & = & \lambda t.\lambda e : t.{\uparrow_t} e \\
 & : & \mathrm{Type} \to \mathrm{Value} \to \mathrm{TLT} \\
{\uparrow_b}\ e & = & e \\
{\uparrow_{t_1 \to t_2}}\ e & = & \overline{\lambda} v_1.\ {\uparrow_{t_2}}\ (e\ \underline{@}\ {\downarrow^{t_1}}\ v_1) \\
{\uparrow_{t_1/\alpha \to t_2/\beta}}\ e & = & \overline{\lambda} v_1.\overline{\mathcal{S}} k_1.\ {\uparrow^{\beta}}\ [\underline{\langle}\ \underline{\mathrm{let}}\ x_1^{\diamond} = (e\ \underline{@}\ {\downarrow_{t_1}}\ v_1)\ \underline{\mathrm{in}}\ (\mathit{init};\ \mathit{wrap}({\downarrow_{\alpha}}\ (k_1\ \overline{@}\ {\uparrow^{t_2}}\ x_1^{\diamond})))] \\
{\uparrow_{t_1 \times t_2}}\ e & = & \overline{pair}({\uparrow_{t_1}}\ \underline{fst}\ e,\ {\uparrow_{t_2}}\ \underline{snd}\ e)
\end{array}
$$

図 7. 限定継続のための TDPE

`shift` 式を含むかもしれない static な項である。よって、もし ${\downarrow^{(t_1 \times (t_2 \to \alpha)) \to \beta}}\ v$ が上のように定義されていたとすると、v の中の static な `shift` が全ての計算の残りを切り取ってしまう。代わりに、$(\overline{\lambda} v_1.\ {\uparrow_{\alpha}}\ [(k_1^{\diamond}\ \underline{@}\ {\downarrow^{t_2}}\ v_1)])$ というコンテキストのみを切り取る必要がある。v が切り取るコンテキストの範囲を定めるために、${\downarrow^{(t_1 \times (t_2 \to \alpha)) \to \beta}}\ v$ を次のように定義する。

$\underline{\lambda} x_1^{\diamond}.\underline{\mathcal{S}} k_1^{\diamond}.\mathit{init};$
$\mathit{wrap}({\downarrow^{\beta}}\ \overline{\langle}(\overline{\lambda} v_1.\ {\uparrow_{\alpha}}\ [k_1^{\diamond}\ \underline{@}\ {\downarrow^{t_2}}\ v_1])\overline{@}(v\ \overline{@}\ {\uparrow_{t_1}}\ x_1^{\diamond})\overline{\rangle})$
同様にして reflect ${\uparrow_{t_1/\alpha \to t_2/\beta}}\ e$ は
$\overline{\lambda} v_1.\overline{\mathcal{S}} k_1.$
$\quad {\uparrow^{\beta}}\ [\underline{\langle}(\underline{\lambda} x_1^{\diamond}.\mathit{init}; \mathit{wrap}({\downarrow_{\alpha}}\ (k_1\ \overline{@}\ {\uparrow^{t_2}}\ x_1^{\diamond})))$
$\qquad\qquad\qquad\qquad \underline{@}(e\ \underline{@}\ {\downarrow_{t_1}}\ v_1)\underline{\rangle}]$
のように定義が出来る。

これらの式の static や dynamic な β-redex を、static や dynamic な `let` 式で書き換える事で、図 7 の限定継続のための TDPE を得る事ができる。(static な) `shift/reset` を使っているため、`shift` や `reset` の命令 [7] をもつ言語で実行することを仮定している。

5.3 Scheme での実装

図 7 の TDPE を単純に Scheme で実装した例を図 8 に示す。メインの関数 `residualize` は式とその型を受け取り、TDPE の結果を返すような関数である。

4 節での let-insertion については state-based のものを使用した。その理由として、実際に計算を行う対象言語の `shift/reset` が let-insertion を行う `shift/reset` に影響を与えてしまうためである。例えば、${\downarrow^{t_1/\alpha \to t_2/\beta}}\ v$ において $k_1^{\diamond}\underline{@}\ {\downarrow^{t_2}}\ v_1$ という dynamic な関数適用を外側の wrap の位置に残す必要がある。しかし、$k_1^{\diamond}\underline{@}\ {\downarrow^{t_2}}\ v_1$ はもうすでに他の static な `reset` によって囲まれているので、その wrap の位置に残すことは出来ない。

よって、shift/reset-based let insertion を使うには、$\mathtt{shift}_2$ [6] が必要となる。

5.4 例

いくつかの例を実行した結果を図 9 に示した。`parse` という関数は、型の parser である。
例えば、`'((A * B) / C -> D / E)` という型を
`(make-func-sr (make-prod (make-base 'A) (make-base 'B)) (make-base 'C) (make-base 'D) (make-base 'E))`
という Scheme 上で使える構造体に変換するような関数である。

一番最初の例としては pure な関数型を持つ式 $(\overline{\lambda} x.(\overline{\lambda} f.(f\overline{@}x)))$ を考える。これは pure な関数型なので、call-by-value の TDPE と同じように部分評価され、$(\underline{\lambda} x.x)$ となる。

二つ目の例は pure でない関数型を持つ式 $(\overline{\lambda} f.(f\overline{@}(\overline{\mathcal{S}} k_1.(k_1\overline{@}(\overline{\mathcal{S}} k_2.(k_2\overline{@}(+\ \ 1\ \ 2)))))))$ である。この式中の $\overline{\mathcal{S}} k_1$ は、$f@\Box$ という継続と、外側の継続 h を受け取る。つまり、$\overline{\mathcal{S}} k_1$ が切り取

る継続をまとめると、$\underline{\langle} h@(f@\Box)\underline{\rangle}$ となる。また、$\overline{\mathcal{S}}k_2$ は $k@\Box$ という継続を受け取る。ここで実行結果を簡略化してみると、

```
(lambda (f) (shift h (reset h (f 3))))
```

となり、$\overline{\mathcal{S}}k_1$ と $\overline{\mathcal{S}}k_2$ の二つの static な `shift` 式が実行され、必要な dynamic な `shift` 式が作られたことが分かる。

三つ目の例も pure でない関数型を持つ例で、受け取った継続を 2 回使うような例 $\overline{\lambda}f.f\overline{@}(\overline{\mathcal{S}}k.k\overline{@}(k\overline{@}(1+2)))$ である。前の例と同じく $\overline{\mathcal{S}}k$ は $\underline{\langle} h@(f@\Box)\underline{\rangle}$ という継続を受け取り、それを $k\overline{@}(k\overline{@}(1+2))$ と 2 回使っている。ここで実行結果を見てみると、継続は実際に 2 回に展開されていて、正しく実行されていることが分かる。

最後の二つの例は `shift/reset` で書かれた (typed) `printf` 関数 [2] に関する実行例である。まず、この関数 `(lambda (f) (reset (f (cons (% str) (f (cons " is " (% str)))))))` は、関数 f をペアとして与えられた二つの string を結合させて一つの string を返すような関数だとすると、C 言語での `sprintf "%s is %s"` と同じ働きをする。`(% str)` は 2 回使われるので、この式は引数を 2 つ受け取って string を返すような関数である。よってこの式の型は `string -> string -> string` となる。この型は実際には pure な型であるが、四つ目の例では pure でない型を持つとして、五つ目の例では pure な型としてそれぞれ実行した例となっている。

pure でない型として実行した四つ目の例は複雑な結果となっているが、`let` 式を η 簡約したり、tail な位置の継続を全て実行してしまうと、

```
(lambda (f) (lambda (x) (lambda (y)
  (reset (let ((g3 (f (cons " is " y))))
    (reset (f (cons x g3)))))))))
```

のようになる。この結果の式は、`sprintf "%s is %s"` で期待されたように、指定された数の引数を受け取って string を返す関数になっている。

最後に五つ目の例は、 前の例と同じように、指定された数の引数を受け取って string を返す関数になっている。上の例で示した簡略化した式とほとんど同じになっているが、`reset` の有無が異なるのは、上の例では pure でない型として実行しているので、pure でない場合に備えて継続を区切っているためである。

6 正当性の証明

この節では、call-by-name の TDPE の証明を紹介した後、今後、他の TDPE の証明がどのような形になるかについての展望を述べる。

6.1 call-by-name の TDPE の正当性

call-by-name の TDPE については、Filinski が意味論に基づく Kripke の論理関係を使った証明を行っている [8]。ここではそれを平易に噛み砕いて紹介する。

証明する上で必要となる call-by-name の TDPE の定義はすでに図 1 に示している。また、TDPE 前は static な項であるが、TDPE 後には dynamic な項となるため、それら 2 つを比較するために dynamic な項を static に変換する関数が必要となる。

・dynamic な項を static に変換する関数

$$\begin{aligned} I[\![x]\!]\rho &= \rho(x) \\ I[\![\underline{\lambda}x.M]\!]\rho &= \lambda v.I[\![M]\!]\rho[x := v] \\ I[\![M\underline{@}N]\!]\rho &= I[\![M]\!]\rho@I[\![N]\!]\rho \end{aligned}$$

これは、dynamic な term に対するインタプリタととらえても良い。

次に、型に関する帰納法で、次のような論理関係を定義する。

(論理関係の定義)

$$\begin{aligned} R_b(M, M') &\Leftrightarrow I[\![M]\!]\phi \sim M' \\ R_{A\to B}(M, M') &\Leftrightarrow \forall R_A(M_1, M_1') \text{ について} \\ &\quad\ R_B(M@M_1, M'@M_!') \end{aligned}$$

ここで、ϕ は M の自由変数と M' の自由変数を関係づける環境である。また $\sim$ は call-by-name のもとでの等価性を示す。ここでは、M と M' で同じ変数名を使うこととし、ϕ は恒等写像とする。

上に示した論理関係は標準的なものだが、TDPE の証明には次のような補題が必要となる。

(1) $R_T(M\rho, M'\rho')$ ならば $I[\![\downarrow^T M]\!]\rho \sim M'\rho'$
(2) $I[\![M]\!] \sim M'$ ならば $R_T(\uparrow_T M, M')$

これらは、論理関係と、reify, reflect の間の関係を規定している。これらの補題は、型に関する帰納法で示すことができる。

論理関係の基本定理は次のようになる。
(定理)
$\Gamma \vdash M : T$ かつΓで定義されている全ての変数 x について $R_{\Gamma(x)}(\rho(x), \rho'(x))$ であれば
$R_T(M\rho, M\rho')$

ここで $M\rho$ は M に ρ を施した項である。この定理は、項に関する帰納法で証明することができる。

論理関係の基本定理と上の補題 (1) を使うと TDPE の正当性を示す次の系を示すことができる。

系　　$\vdash M : T$ ならば $I[\![\downarrow^T M]\!] \sim M$

6.2 call-by-value の正当性に向けて

Filinski [9] は、call-by-value の computational λ-calculus に対する TDPE を示しており、そこで let-insertion をモナドの形で導入している。その正当性についても、call-by-name の手法を computational λ-calculus の値と計算のレベルで相互再帰的に定義することで証明できると書いているが、詳細は記述されていない。現在、この方針で証明を試みているところである。

さらに、この証明を `shift/reset` 付きの TDPE の正当性の証明に拡張するには、以下の点を考慮する必要があると予想される。まず、第１に TDPE 中に静的な `shift` が実行されるため、論理関係の定義もそれを考慮する必要があると思われる。論理関係の $R_{A\to B}$ の場合は、任意の関係づいている引数を受け取ったとき、実行結果も関係づいていることを保証するが、`shift/reset` が使われる場合にはその継続の振る舞いも保証する必要がある。これには、`shift/reset` に対する通常の部分評価の正当性 [1] で使われた論理関係が参考になると思われる。

次に、`shift/reset` 付きの TDPE においては、入力プログラムの `shift` に加え、let-insertion を行うために $\texttt{shift}_2/\texttt{reset}_2$ が使われる。したがって、論理関係を `shift/reset` だけでなく $\texttt{shift}_2/\texttt{reset}_2$ も扱えるように拡張する必要があるかも知れない。現在、これらの考察をもとに、その証明を試みているところである。

7 結論

この論文では限定継続 `shift/reset` の TDPE を改良し、その実装と実行例を示した。また、限定継続 `shift/reset` の TDPE の正当性を証明するために足掛かりとなる call-by-name の TDPE の正当性を示し、今後の証明の方針を示した。

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. “On Typing Delimited Continuations: Three New Solutions to the Printf Problem,” To appear in *Higher-Order and Symbolic Computation*, 17 pages (2010).

[3] Asai, K., and Kameyama Y “Polymorphic Delimited Continuations”, In Z. Shao editor, *5th Asian Symposium on Programming Languages and Systems (APLAS 2007)*, pp. 239-254 (November 2007).

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

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

[6] Danvy, O., and A. Filinski “Abstracting Control,” *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[7] Filinski, A. “Representing Monads,” *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

```
(define-struct base (base-type))
(define-struct func-p (domain range))            ; domain -> range
(define-struct func-sr (domain bef range aft))   ; domain / bef -> range / aft
(define-struct prod (type1 type2))               ; type1 * type2

(define wrap-stack '())

(define (init)
  (set! wrap-stack (cons '() wrap-stack)))

(define (wrap e)
  (let ([lst (car wrap-stack)])
    (set! wrap-stack (cdr wrap-stack))
    (wrap-gen lst e)))

(define (add e)
  (let ([g (gensym)])
    (set! wrap-stack (cons (cons (list g e) (car wrap-stack))
                           (cdr wrap-stack)))
    g))

(define (wrap-gen list body)
  (cond [(null? list) body]
        [else (wrap-gen (cdr list)
                        `(reset (let ((,(car (car list)) ,(car (cdr (car list)))))) ,body)))]))

(define (residualize v t)
  (letrec
     ([reify (lambda (t v)
               (cond [(base? t) v]
                     [(func-p? t) (let ([t1 (func-p-domain t)][t2 (func-p-range t)][x1 (gensym)])
                                    `(lambda (,x1) ,(reify t2 (v (reflect t1 x1)))))]
                     [(func-sr? t) (let ([t1 (func-sr-domain t)] [t2 (func-sr-range t)]
                                         [bef (func-sr-bef t)] [aft (func-sr-aft t)]
                                         [x1 (gensym)] [k1 (gensym)])
                                     `(lambda (,x1)
                                        (shift ,k1 ,(begin (init) (wrap (reify aft (reset
                                          (let ([v1 (v (reflect t1 x1))])
                                            (reflect bef (add `(,k1 ,(reify t2 v1)))))))))))])
                     [(prod? t) (let ([t1 (prod-type1 t)] [t2 (prod-type2 t)])
                                  `(cons ,(reify t1 (car v)) ,(reify t2 (cdr v))))])))]
      [reflect (lambda (t e)
                 (cond [(base? t) e]
                       [(func-p? t) (let ([t1 (func-p-domain t)][t2 (func-p-range t)])
                                      (lambda (v1) (reflect t2 `(,e ,(reify t1 v1)))))]
                       [(func-sr? t) (let ([t1 (func-sr-domain t)] [t2 (func-sr-range t)]
                                           [bef (func-sr-bef t)] [aft (func-sr-aft t)] [x1 (gensym)])
                                       (lambda (v1)
                                         (shift k1 (reflect aft (add `(reset
                                           (let ([,x1 (,e ,(reify t1 v1))])
                                             ,(begin (init)
                                                     (wrap (reify bef (k1 (reflect t2 x1))))))))))))]
                       [(prod? t) (let ([t1 (prod-type1 t)] [t2 (prod-type2 t)])
                                    (cons (reflect t1 `(car ,e)) (reflect t2 `(cdr ,e))))]))])
    (reify t v)))
```

図 8. Scheme での限定継続のための TDPE の実装

```
> (residualize (lambda (x) ((lambda (f) f) x)) (parse '(A -> A)))
(lambda (x) x)

> (residualize (lambda (f) (f (shift k (k (shift k2 (k2 (+ 1 2)))))))
               (parse '((I / A -> C / B) / A -> C / B)))
(lambda (f) (shift h
   (reset (let ((g3 (reset (let ((g1 (f 3))) (reset (let ((g2 (h g1))) g2)))))) g3))))

> (residualize (lambda (f) (f (shift k (k (k (+ 1 2))))))
               (parse '((I / A -> B / I) / A -> B / I)))
(lambda (f) (shift h
  (reset (let ((g3 (reset (let ((g1 (f 3))) (reset (let ((g2 (h g1))) g2))))))
    (reset (let ((g6 (reset (let ((g4 (f g3))) (reset (let ((g5 (h g4))) g5))))))
             g6))))))

; (printf)
> (define (str x) x)
> (define (% to_str) (shift k (lambda (x) (k (to_str x)))))
> (residualize (lambda (f) (reset (f (cons (% str) (f (cons " is " (% str)))))))
       (parse '(((S * S) / A -> S / A) / A -> (S / A -> (S / A -> S / A) / A) / A)))
(lambda (f) (shift k
  (reset (let ((g9 (k
           (lambda (x) (shift k1
             (reset (let ((g8 (k1
                      (lambda (y) (shift k2
                        (reset (let ((g6 (reset (let ((g3 (f (cons " is " y))))
                                                  (reset (let ((g5
                                                     (reset (let ((g4 (f (cons x g3)))) g4))))
                                                     g5))))))
                          (reset (let ((g7 (k2 g6)))
                            g7)))))))))
                    g8)))))
           g9))))

> (residualize (lambda (f) (reset (f (cons (% str) (f (cons " is " (% str)))))))
       (parse '(((S * S) -> S) -> (S -> (S -> S)))))
(lambda (f) (lambda (x) (lambda (y) (f (cons x (f (cons " is " y)))))))
```

図 9. 限定継続のための TDPE の Scheme での実行例

[8] 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).

[9] 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).

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

[11] Sumii, E., and N. Kobayashi "A Hybrid Approach to Online and Offline Partial Evaluation," *Higher-Order and Symbolic Computation*, Vol. 14, Nos. 2/3, pp. 101–142, Kluwer Academic Publishers (2001).

[12] Thiemann, P. J. "Cogen in Six Lines," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pp. 180–189 (May 1996).

[13] Tsushima, K., and Asai, K. "Towards Type-Directed Partial Evaluation for Shift and Reset" *Normalization of Evaluation 09*, (August 2009).