shift/reset の selective CPS 変換

上原 千裕, 浅井 健一

お茶の水女子大学

g1120506@is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要 shift/reset は限定継続を処理するオペレータであり, 非決定性プログラミングなど様々な目的で使われる. しかし, shift/reset を直接サポートしている言語は少ない. shift/reset をサポートしていない言語で shift/reset を使うには, プログラム全体を CPS 変換するという方法があるが, 多くの場合, CPS 変換すると効率は落ちる. そこで, 本稿では shift/reset を実装するのに必要な部分のみを, システムが自動で特定して CPS 変換する selective CPS 変換を提案する. Selective に CPS 変換することで, shift/reset に無関係な部分の効率を落とすことなく shift/reset を使うことができるようになる.

1 はじめに

継続とは, 計算中のある時点における残りの計算を表す. 継続の中でも限定継続は, 一定範囲内のみの残りの計算を表す. shift/reset [3] は限定継続を扱うことができるオペレータで, shift はその時点での継続を切り取ってくる命令, reset は切り取られる継続の範囲を限定する命令である. 限定継続命令を使うと, ユーザがプログラムの実行順序を制御できるため, 例外処理や非決定プログラミングなどいろいろな場面で使われる.

これまで, 限定継続命令をサポートするには, システムに手を加えてスタックを操作できるようにすることで直接実装する方法が多くとられてきた. Scheme 上での直接実装 [6], OCaml の Delimcc ライブラリ [8], Caml Light を拡張した OchaCaml [10] などはこのアプローチをとっている. 一方, ユーザプログラム全体を CPS 変換して既存の言語の上で shift/reset を使うという選択肢がある. この選択肢は, システムに手を自力で加えることや, shift/reset を使うために新しい言語を手元に用意することなく, shift/reset が使えるという利点がある. しかし, ユーザプログラム上の CPS 変換は, 多くの場合効率が落ちる. ここで, shift/reset に関係した部分のみを選んで変換, つまり selective に CPS 変換 [11] すれば, 効率を落とすことなく限定継続命令をサポートできる可能性がある.

本稿では, 関数型言語を対象にした, selective に CPS 変換するシステムを提案する. このシステムでは, 「式の評価中に shift が行われる可能性があるか」, つまり「CPS 変換を必要とするか」という情報を型システムの制約の形で表現する. 具体的には, 以下の手順で selective に CPS 変換を行う.

1. 型推論を行う. この時, 制約は保存しておく.
2. 制約を解くことにより, 「CPS 変換を必要とするか」という情報を注釈として式に付ける.
3. 解が得られたら CPS 変換が必要な場所が特定されるので, その情報に従って CPS 変換が必要な場所だけ選んで, つまり selective に CPS 変換を行う.

ユーザの視点からこのシステムを見ると, ユーザは注釈なしのプログラムをシステムに入力する. するとシステムが注釈を自動で式に付け, プログラムを selective に CPS 変換する. 得られたプログラムは shift/reset を含まないため, 既存の処理系で実行できる.

ここで, 制約解消は通常「最も良い解」を得られるように行うが, 本研究で扱う制約の解には「最も良い解」が必ずしも存在するとは限らないことを後に示す. その上で本稿では, 過度に型推論と制約解消を複雑にすることなく, ある程度の解を得られる制約解消法について述べる.

$$
\begin{array}{rcll}
\alpha & ::= & \epsilon \mid i & \text{annotation} \\
t & ::= & \tau \mid \texttt{bool} \mid \texttt{int} \mid t_1 \to t_2 \,@\mathrm{cps}[t_3, t_4, \alpha] & \text{monomorphic type} \\
T & ::= & t \mid \forall \tau. T & \text{polymorphic type} \\
v & ::= & n \mid x \mid \texttt{true} \mid \texttt{false} \mid \lambda^{\alpha} x.A \mid \texttt{fix}^{\alpha}\ f.x.A & \text{value} \\
e & ::= & v \mid A_1 \,@^{\alpha}\, A_2 \mid Sk.A \mid \langle A \rangle \mid \texttt{if}\ A_1\ \texttt{then}\ A_2\ \texttt{else}\ A_3 \mid \texttt{let}\, x = v^{\epsilon}\, \texttt{in}\ A & \text{expression} \\
A & ::= & e^{\alpha} & \text{expression with annotation}
\end{array}
$$

図 1. 対象言語

なお, Scala では, ユーザの注釈に従って shift/reset を実装するのに必要な部分を selective に CPS 変換するという方法 [13] がとられている. これに対して本研究が提案するシステムは, ユーザが注釈をプログラムに付ける必要はなく, システムが自動で注釈を付ける.

また, SML/NJ [1] のように, 内部表現として CPS を使ったコンパイラであれば, コンパイラに手を加えることで比較的容易に shift/reset をサポートできると思われるが, ここでは OCaml のように内部表現として CPS を使っていない言語を対象としている.

本稿は, 第 2 節で本稿の対象言語と注釈を定義する. 定義した型規則に沿って型推論をするが, 注釈を決定するための制約が発生する. そのため制約解消法を第 3 節で定義する. そして注釈が付いたプログラムのための selective CPS 変換を第 4 節で定義する. 第 5 節で, 成果として簡単なプログラムで実際に selective CPS 変換を行い, selective に CPS 変換することで効率が向上することをみる. 第 6 節でライブラリ関数を導入する方法を提案する. 第 7 節で関連研究, 第 8 節でまとめを述べる.

2 注釈, 対象言語, 型規則

本稿で扱うのは, let 多相の入った λ 計算に整数, 真偽値と if 文, 不動点演算子 fix, そして限定継続命令 shift/reset が加わった言語である. その定義を図 1 に示す. τ は型変数, n は整数, x は変数, shift を表す $Sk.A$ の k は変数である.

2.1 注釈と対象言語

α は「式の評価中に shift が行われる可能性があるか」, つまり「CPS 変換が必要かどうか」を示す注釈である. また, $A = e^{\alpha}$ は式 e に注釈 α が付いた式を示す. 注釈 ϵ は CPS 変換が不要であることを示す. CPS 変換が不要な場合, つまり shift によるコントロールエフェクトが起きない場合は, その式は pure である, という表現を使う. 一方, 注釈 i は CPS が必要であることを示す. そのような式は impure であるという. これはその式の実行中に shift が起こる可能性があることを示している. また, 注釈 i は impure の頭文字からとっている.

多相の let 文は value restriction を仮定して束縛される式は常に v としている. 値でない e における $\texttt{let}\, x = e^{\alpha}\, \texttt{in}\ A$ は, 内部的に $(\lambda x.\, A) \,@\, e^{\alpha}$ と扱う.

$Sk.\, A$ は shift 命令である. これで現在の継続を切り取って k に入れた上で A を行うことを示す. また $\langle A \rangle$ は reset 命令である. これで, A の中で起こる shift で切り取られる継続を A までに限定する. 以下で例を示すが, 注釈は省略している. これは, 注釈を無視した簡約規則が, 注釈付きの式に対する簡約規則であり, 簡約の本質的な意味に注釈は関係無いからである.

$$\langle (Sk.\, k \,@\, 2) - 1 \rangle$$

は, $(Sk.\, k \,@\, 2)$ を実行する時, 計算結果から 1 を引くことが継続である. この式では, $\lambda x.\, \langle x - 1 \rangle$ を k に入れて,

$$\langle (\lambda x.\, \langle x - 1 \rangle) \,@\, 2 \rangle$$

という式になる．後は関数適用が実行されて, 1 が返る．また `string_of_int` を, `int` を `string` に変える関数であるとして,

$$\langle (Sk.\,\texttt{string\_of\_int}\ (k\,@\,2)) - 1 \rangle$$

という式を考える．この式は実行すると

$$\langle \texttt{string\_of\_int}\ ((\lambda x.\,\langle x - 1\rangle)\,@\,2)\rangle$$

となるため, $(\lambda x.\,\langle x-1\rangle)\,@\,2$ の結果である `int` 型の 1 が `string` 型になって返る．この時, reset の中身が返す型を考える．

$$(Sk.\,\texttt{string\_of\_int}\ (k\,@\,2)) - 1$$

は引き算であるから `int` を返す予定だったが, 実際は `string` を返した．このような, reset の中身が返す型を answer type と呼ぶ．この例では, shift 命令により answer type が `int` から `string` に変化している．このことを answer type modification という．

関数の型は $(t_1 \to t_2\ @\mathrm{cps}[t_3, t_4, \alpha])$ という形になる．これで大雑把には「t_1 を受け取ったら t_2 を返す関数」を意味するが, この関数を実行すると answer type が t_3 から t_4 へ変化し, この関数の本体部分の注釈は α であることを示す．この α が ϵ なら, この関数の本体は pure であり, i なら impure であることを示す．

2.2 型規則

型規則を図 2 に示す．この型規則は, Asai and Kameyama [2] の型規則に注釈と制約を加えたものである．厳密には, Asai and Kameyama [2] の型規則とは異なる点があるが, それについては第 2.3 節で説明する．

型判断は,

$$\Gamma \vdash e^{\alpha} : t_1\ @\mathrm{cps}[t_2, t_3, \alpha]$$

の形をしている．これは, 型環境 Γ のもとで式 e^{α} は型 t_1 を持ち, answer type を t_2 から t_3 に変え, 注釈 α を持つことを意味する．注釈 α は式 e が pure か impure かを示す．式 e に付く注釈 α は, $@\mathrm{cps}[t_2, t_3, \alpha]$ のように式 e の answer type に付く注釈 α と一致する．式と answer type の両方に注釈を付けるのは, selecitve CPS 変換は式だけを見て行い, 型推論は型だけを見て行うためである．また, var などの pure な型判断の結論部だけに表れる $@\mathrm{cps}[t_2, t_2, \epsilon]$ では, answer type が変わらないことを示すだけなので t_2 は free である．すると, pure な型判断に付く $@\mathrm{cps}[t_2, t_2, \epsilon]$ はわざわざ書く必要がないように思える．しかし pure な型判断に付く $@\mathrm{cps}[t_2, t_2, \epsilon]$ を消すと, impure な型判断を pure な型判断にする, 以下の規則が必要になる．

$$\frac{\Gamma \vdash e^{\epsilon} : t_1}{\Gamma \vdash e^{\epsilon} : t_1\ @\mathrm{cps}[t_2, t_2, \epsilon]}$$

この型規則は導出する式 e が限定されていないため, 型推論の際に, この型規則を使う場所はすぐには判断できない．本研究は, 型推論を過度に複雑にしないことを掲げているため, Pure な型判断に $@\mathrm{cps}[t_2, t_2, \epsilon]$ を付けることで, 全ての型規則が導出する式を 1 種類にして, 型推論を単純にしている．

他の規則を見る前に制約の説明をする．制約は以下の 3 種類からなる．

$$(\_ = \_) \qquad (\_ \le \_) \qquad (\_ \neq \_ \Rightarrow \_ = i)$$

これらの制約が満たされる場合を全て書き下すと図 3 になる．

最初の制約は両辺が等しくなければならないことを示す．この制約は図 2 の型規則には出て来ないが, 後に制約解消を行う際に使われる．次の制約 $(\alpha \le \alpha')$ は, α が i なら α' も i にならなければならず, α' が ϵ なら α も ϵ にならなければならないことを示す．ϵ と i の大小関係は $\epsilon < i$ である．

$$\frac{(x^{\epsilon} : T \in \Gamma \text{ and } t_1 \subseteq T)}{\Gamma \vdash x^{\epsilon} : t_1 \,@\mathrm{cps}[t_2, t_2, \epsilon]}\ \text{var} \qquad \frac{(c^{\epsilon} \text{ is a constant of basic type } t_1)}{\Gamma \vdash c^{\epsilon} : t_1 \,@\mathrm{cps}[t_2, t_2, \epsilon]}\ \text{const}$$

$$\frac{\Gamma, x : t_1 \vdash e^{\,\alpha_2} : t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_2] \quad (\alpha_2 \le \alpha_1) \quad (t_3 \neq t_4 \Rightarrow \alpha_2 = i)}{\Gamma \vdash (\lambda^{\alpha_1} x.e^{\,\alpha_2})^{\,\epsilon} : (t_1 \to t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_1]) \,@\mathrm{cps}[t_5, t_5, \epsilon]}\ \text{fun}$$

$$\frac{\Gamma, f^{\,\epsilon} : (t_1 \to t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_2]), x^{\,\epsilon} : t_1 \vdash e^{\,\alpha_2} : t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_2] \quad (\alpha_2 \le \alpha_1) \quad (t_3 \neq t_4 \Rightarrow \alpha_2 = i)}{\Gamma \vdash (\mathtt{fix}^{\alpha_1}\ f.x.e^{\,\alpha_2})^{\,\epsilon} : (t_1 \to t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_1]) \,@\mathrm{cps}[t_5, t_5, \epsilon]}\ \text{fix}$$

$$\frac{\begin{array}{c}(\alpha_1 \le \alpha) \quad (\alpha_2 \le \alpha) \quad (\alpha_3 \le \alpha) \quad (t_3 \neq t_6 \Rightarrow \alpha_3 = i) \quad (t_5 \neq t_4 \Rightarrow \alpha_1 = i) \quad (t_6 \neq t_5 \Rightarrow \alpha_2 = i) \\ \Gamma \vdash e_1{}^{\,\alpha_1} : (t_2 \to t_1 \,@\mathrm{cps}[t_3, t_6, \alpha_3]) \,@\mathrm{cps}[t_5, t_4, \alpha_1] \qquad \Gamma \vdash e_2{}^{\,\alpha_2} : t_2 \,@\mathrm{cps}[t_6, t_5, \alpha_2]\end{array}}{\Gamma \vdash (e_1{}^{\,\alpha_1} \,@^{\alpha_3}\, e_2{}^{\,\alpha_2})^{\,\alpha} : t_1 \,@\mathrm{cps}[t_3, t_4, \alpha]}\ \text{app}$$

$$\frac{\Gamma \vdash e^{\,\alpha_1} : t_2 \,@\mathrm{cps}[t_2, t_1, \alpha_1] \qquad (t_2 \neq t_1 \Rightarrow \alpha_1 = i)}{\Gamma \vdash \langle e^{\,\alpha_1} \rangle^{\,\epsilon} : t_1 \,@\mathrm{cps}[t_3, t_3, \epsilon]}\ \text{reset}$$

$$\frac{\Gamma, k^{\,\epsilon} : \forall \tau_5.(t_1 \to t_3 \,@\mathrm{cps}[\tau_5, \tau_5, \epsilon]) \vdash e^{\,\alpha_1} : t_2 \,@\mathrm{cps}[t_2, t_4, \alpha_1] \qquad (t_2 \neq t_4 \Rightarrow \alpha_1 = i)}{\Gamma \vdash (Sk.e^{\,\alpha_1})^{\,i} : t_1 \,@\mathrm{cps}[t_3, t_4, i]}\ \text{shift}$$

$$\frac{\Gamma \vdash v^{\,\epsilon} : t_1 \,@\mathrm{cps}[t_5, t_5, \epsilon] \qquad \Gamma, x^{\,\epsilon} : \mathtt{Gen}(t_1; \Gamma) \vdash e^{\,\alpha_1} : t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_1] \qquad (t_3 \neq t_4 \Rightarrow \alpha_1 = i)}{\Gamma \vdash (\mathtt{let}\ x = v^{\,\epsilon}\ \mathtt{in}\ e^{\,\alpha_1})^{\,\alpha_1} : t_2 \,@\mathrm{cps}[t_3, t_4, \alpha_1]}\ \text{let}$$

$$\frac{\begin{array}{c}(\alpha_1 \le \alpha) \quad (\alpha_2 \le \alpha) \quad (\alpha_3 \le \alpha) \quad (t_4 \neq t_3 \Rightarrow \alpha_1 = i) \quad (t_2 \neq t_4 \Rightarrow \alpha_2 = i) \quad (t_2 \neq t_4 \Rightarrow \alpha_3 = i) \\ \Gamma \vdash e_1{}^{\,\alpha_1} : \mathtt{bool} \,@\mathrm{cps}[t_4, t_3, \alpha_1] \qquad \Gamma \vdash e_2{}^{\,\alpha_2} : t_1 \,@\mathrm{cps}[t_2, t_4, \alpha_2] \qquad \Gamma \vdash e_3{}^{\,\alpha_3} : t_1 \,@\mathrm{cps}[t_2, t_4, \alpha_3]\end{array}}{\Gamma \vdash (\mathtt{if}\ e_1{}^{\,\alpha_1}\ \mathtt{then}\ e_2{}^{\,\alpha_2}\ \mathtt{else}\ e_3{}^{\,\alpha_3})^{\,\alpha} : t_1 \,@\mathrm{cps}[t_2, t_3, \alpha]}\ \text{if}$$

図 2. 型規則

$$\begin{array}{ccccc}(\epsilon = \epsilon) & (i = i) & (\epsilon \le \epsilon) & (\epsilon \le i) & (i \le i) \\ (t_1 \neq t_1 \Rightarrow \epsilon = i) & & (t_1 \neq t_2 \Rightarrow i = i) & & \\ (\alpha_1 \neq \alpha_1 \Rightarrow \epsilon = i) & & (\alpha_1 \neq \alpha_2 \Rightarrow i = i) & &\end{array}$$

図 3. 制約

最後の制約は, 図 2 の型規則では, 最初の 2 つには型が入り

$$(t_1 \neq t_2 \Rightarrow \alpha = i)$$

という形になる. これで, t_1 と t_2 が異なる型ならば α は i にならなければならないことを示す. 言い換えると, α が ϵ にならば, t_1 と t_2 は同じでなければならない.

図 2 の型規則の前提部に $@\mathrm{cps}[t_1, t_2, \alpha]$ が出てくる度に $(t_1 \neq t_2 \Rightarrow \alpha = i)$ という制約が加わっている. これで, answer type が変わるなら α は i になり, α が ϵ なら $t_1 = t_2$ となることを示している.

なお, 制約解消の段階では

$$(\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i)$$

という形の制約も使用する. この制約では, 最初の 2 つの部分には注釈が入っている.

$(\alpha \le \alpha')$ の形の制約が図 2 の型規則のどこに出てくるかを順に見る.

規則 fun と fix では, 関数の本体部分の注釈 α_2 と返される関数の型の注釈 α_1 の間に, $(\alpha_2 \le \alpha_1)$ という制約が導入されている. これは, 関数本体が impure なら関数の型も impure になることを示している. $(\alpha_2 = \alpha_1)$ ではなく $(\alpha_2 \le \alpha_1)$ となっているのは, 関数本体が pure でも, 周りの状況によってその関数を impure と扱いたい場合があるためである. 例えば,

$$\lambda f.\, \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ f \,@\, (\lambda z.\, (Sk.\, k \,@\, (k \,@\, z)))\ \mathtt{else}\ f \,@\, (\lambda x.\, x)$$

という項の $\lambda x.\, x$ である. 図 2 の型規則に合うように $(\lambda z.\, (Sk.\, k \mathbin{@} (k \mathbin{@} z)))$ に型と注釈を付けると,

$$[\,] \vdash (\lambda^{i} z.\, (Sk.\, (k^{\epsilon} \mathbin{@}^{\epsilon} (k^{\epsilon} \mathbin{@}^{\epsilon} z^{\epsilon})^{\epsilon})^{\epsilon})^{i})^{\epsilon} : (t_1 \to t_1 \,@\mathrm{cps}[t_1, t_1,\, i]) \,@\mathrm{cps}[t_2, t_2,\, \epsilon]$$

となるため, f の引数の型は $(t_1 \to t_1 \,@\mathrm{cps}[t_1, t_1,\, i])$ に固定される. この型を $\lambda x.\, x$ が持つ必要があるため, この例のプログラム中の $\lambda x.\, x$ は, 図 2 の型規則に合うように型に注釈を付けると

$$[\,] \vdash (\lambda^{i} x.\, x^{\epsilon})^{\epsilon} : (t_1 \to t_1 \,@\mathrm{cps}[t_1, t_1,\, i]) \,@\mathrm{cps}[t_4, t_4,\, \epsilon]$$

となる. 従ってこの例は, 型規則 fun において $\alpha_1 = i, \alpha_2 = \epsilon$ となる例である.

規則 app は関数呼び出しである. 3 つの不等式制約で, 関数部分, 引数部分, 関数本体のどこかが impure なら, 全体として impure になることを示している. また, すべて pure だったとしても, 全体として impure に扱うことも許している.

規則 reset は, 本体部分では shift が使われるかもしれないが, その効果はこの reset までで外には漏れないので, reset 自身は pure として良いことを示す.

規則 shift は impure であること, ここで捕捉される継続 k は pure であることを示す. ここが唯一, impure であることを強制する場所である. 導出される式には impure であることを示す $@\mathrm{cps}[t_3, t_4,\, i]$ が付く.

規則 let は vaule restriction がかかっているため, v は pure で全体の注釈は e の注釈と一致する. また, $\mathrm{FTV}(t)$ が t の自由な型変数の集合を表す時, $\mathtt{Gen}(t_1; \Gamma) = \forall \tau_1 ... \tau_n.t_1$（ただし $\tau_1 ... \tau_n = \mathrm{FTV}(t_1) \backslash \mathrm{FTV}(\Gamma)$）である.

最後に, 規則 if は条件部分, then 部分, else 部分のいずれかが impure だったら全体も impure となることを示す. その際, 規則 app と同様に, 全て pure だったとしても全体として impure に扱うことも許している.

2.3 Asai and Kameyama の型規則との差異

Asai and Kameyama [2] の型規則は, pure な型判断 $\Gamma \vdash_p e : t_1$ と一般の型判断 $\Gamma \vdash e : t_1 \,@\mathrm{cps}[t_2, t_3]$ からなる. 前者は本稿の $\Gamma \vdash e : t_1 \,@\mathrm{cps}[t_2, t_2, \epsilon]$, 後者は $\Gamma \vdash e : t_1 \,@\mathrm{cps}[t_2, t_3, \alpha]$（ただし $\alpha = \epsilon$ または $\alpha = i$）に対応する.

Asai and Kameyama [2] の型規則には exp 規則という以下の規則がある.

$$\frac{\Gamma \vdash_p e : t_1}{\Gamma \vdash e : t_1 \,@\mathrm{cps}[t_2, t_2]}\ \mathrm{exp}$$

この規則は pure な型判断を一般の型判断に変換するため, 一般の型判断の形をとっていても実質的に pure な式があった. 従って answer type が同じだったとき, それは shift が起きるがたまたま answer type が変わらなかったのか, shift が起きないために answer type が変わらなかったのか判断できない. 例えば,

$$\mathtt{if\ true\ then}\ \lambda x.\, Sk.k \mathbin{@} (k \mathbin{@} x)\ \mathtt{else}\ \lambda y.\, y$$

では, if 文の then 部分と else 部分で同じ型が付き,

$$\lambda x.\, Sk.k \mathbin{@} (k \mathbin{@} x) \,:\, (t_1 \to t_1 \,@\mathrm{cps}[t_1, t_1])$$

であるから,

$$\lambda y.\, y \,:\, (t_1 \to t_1 \,@\mathrm{cps}[t_1, t_1])$$

という型が付く. Shift が起きる場所だけを CPS 変換することを考えると, then 部分の関数の本体は CPS 変換が必要で, else 部分の関数 g の本体は CPS 変換を必要としないものとしたい. しかし, どちらも answer type が t_1 から t_1 への変換であり, CPS 変換を必要とするのか判断ができない.

一方本稿では, 注釈を付けることで impure な場所を特定する. この注釈によって CPS 変換が必要かどうかを判断することができる.

本稿で示す型規則は, Asai and Kameyama [2] の型規則から exp 規則を取り除き, それ以外の規則に注釈と制約を加えたものだが, 両者の型付け能力は等価である.

定理 2.1 Asai and Kameyama の型規則で型が付くなら, 本稿の型規則でも型が付く. 逆に, 本稿の型規則で型が付くなら, Asai and Kameyama の型規則でも型が付く.

これを示すには, お互いの型規則を模倣できることを確認すれば良い. 具体的な証明は第一著者の修士論文 [14] で行っているため, ここでは言葉で説明する.

本稿の型規則を Asai and Kameyama [2] の型規則で模倣するには, 基本的には単に注釈を無視して対応する型規則を使えば良い. ただ, Asai and Kameyama [2] の型規則では value が pure な型判断であるが, 本稿の型規則では前提部分に pure な型判断が出てくることはない. そのため, 本稿の型規則で前提部分に value が出て来た場合は, 変換時に前提部分に exp 規則を使うことになる.

一方, Asai and Kameyama [2] の型規則を本稿の型規則で模倣するには, 以下のようにすれば良い.

- Pure な型規則は, 本稿の型規則で注釈を ϵ とすることで模倣する.
- それ以外の型規則は, 本稿の型規則で注釈を i とすることで模倣する.

本稿の型規則が Asai and Kameyama [2] の型規則と異なるところは, 注釈によって pure と判断できる部分を増やしていることである. 上に示した模倣では, pure な型規則以外はすべて注釈を i としたが, 場合によっては ϵ にできる場合もある. それを可能にしているのが本稿の型規則である.

3 制約解消

この第 3 節では, 注釈が i か ϵ かまだ定まっていない注釈変数を α と表す.

入力の式を与えられたら, まず図 2 の型規則に従って型推論を行う. その際, 注釈と制約はそのまま保持しておく. 注釈と制約を除けば図 2 の型規則はこれまでの規則とほぼ同一なので, 普通の let 多相の入った型推論をすれば良い. その結果, 入力の式の各部分式や λ, fix, @ にはユニークな注釈の変数が割り振られ, それに対する制約が生成される.

制約に対する解とは, 全ての注釈の変数に対する注釈の割り当てのうち, 全ての制約を満たしている（図 3 の形になっている）ものである. 生成された制約に対して, 全ての注釈に i を入れると解になっているが, これは次の 3 点を確認することで理解できる.

- $(t_1 \neq t_2 \Rightarrow \alpha = i)$ の形の制約は α に i を入れれば満たされている.
- Value の各構成子及び reset $\langle \_ \rangle$ に対する型規則では, 結論部に ϵ が現れているが, 全ての規則において, 前提部の注釈は $(\alpha_1 \leq \alpha_2)$ の形の制約の左辺にしか出てきていない. 従って, 右辺に i が入れば制約は満たされる.
- Let の型規則の最初の前提部に ϵ が現れているが, value の型規則の結論部は全て ϵ になっている. 従って, ϵ が $(\alpha_1 \leq \alpha_2)$ の形の制約の右辺に出てくることはない.

従って, 解は必ず 1 つは存在することが分かる. 本稿の制約解消の目標は, なるべく小さい解, つまりなるべく ϵ が多い解を見つけることである.

しかし, なるべく小さい解を求めるのは簡単ではない. 一般には「最良の解」は存在しないことが分かっている. 以下では, 型推論と制約解消が過度に複雑にならない範囲で, ある程度の解が得られる制約解消法を示す. その際, 別の方法を取ったらより良い解を見つけられる可能性のある場合は, それを明示する.

制約の解消は, 次の 4 段階からなる.

1. 制約を注釈に関する制約のみに変換する
2. 自明な制約を解消する
3. 自明でない制約を解消する
4. 以上で確定しなかった注釈は ϵ とする

以下, それぞれについて説明する.

3.1 制約を注釈に関する制約のみに変換する

まず, $(t_1 \neq t_2 \Rightarrow \alpha = i)$ という, 型 t_1 と t_2 に言及する制約を, 型に言及しない制約 $(\alpha = i)$, または $(\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i)$ に変換する. 変換は t_1 と t_2 の場合分けで行い, その方針は以下の 2 つである.

- t_1 と t_2 が異なるなら $\alpha = i$ という制約に変換する
- t_1 と t_2 が同じなら, この制約は自明に満たされているので, この制約を削除する

例えば, $(\texttt{bool} \neq \texttt{int} \Rightarrow \alpha = i)$ という制約は, $\texttt{bool} \neq \texttt{int}$ であるから $(\alpha = i)$ となる. また, $(\texttt{int} \neq \texttt{int} \Rightarrow \alpha = i)$ という制約は, 前提が成り立たないため, 既に制約を満たしている. そのため, 制約自体を削除する. t_1 と t_2 が関数型

$$t_1 = (t_{11} \to t_{12}\,@\mathrm{cps}[t_{13}, t_{14}, \alpha_1]) \qquad t_2 = (t_{21} \to t_{22}\,@\mathrm{cps}[t_{23}, t_{24}, \alpha_2])$$

だった場合は, 再帰的に両者が同じかどうかを判断する. 対応する型が 1 つでも異なれば両者は異なる型なので, $\alpha = i$ という制約に変換される. 一方, 注釈も含めて全て同じになったら, 両者は同じ型となり, この制約は削除される.

しかし, この方針だと, 型が全て同じだが注釈のみ異なる場合に困る. 注釈の値はまだ確定していないので, 両者が異なるかどうかをこの段階では判断できない. そこで, この場合は異なる注釈 α_1 と α_2 を使って

$$(\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i)$$

という制約に変換する. これでもとの意味を保ったまま関数型に関する制約を注釈に関する制約に変換することができる. 部分式の中が関数型になっており, その中に異なる注釈がある場合には, $(\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i)$ の形の制約が複数作られることになることに注意する.

以上で, t_1, t_2 が型変数以外の場合の変換が終了した. 以上の変換は, 変換前後で解の集合を変化させない. 従って, 変換後の制約に対して制約解消を行えば, もとの制約に対して制約解消を行ったのと同じ解が得られることになる.

一方, t_1, t_2 の片方, あるいは両方が型変数の場合は, 両者が同じ型変数である場合を除くと, この変換を行うのは簡単ではない. 例として,

$$\texttt{let}\, f = \lambda g.\, g\,@\,1\,\texttt{in}\; f\,@\,(\lambda x.\, x)$$

を考える. $\lambda g.\, g\,@\,1$ に対して型推論を行うと以下のようになる. 式に付く注釈は省略している. また,

$$\Gamma' = g : (\texttt{int} \to \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_4])$$

とする. この例の更に詳しい解説は著者の修士論文 [14] で行っている.

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash g : (\texttt{int} \to \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_4])\,@\mathrm{cps}[\_, \_, \epsilon]}\,\mathrm{var} \qquad \dfrac{}{\Gamma' \vdash 1 : \texttt{int}\,@\mathrm{cps}[\_, \_, \epsilon]}\,\mathrm{const}}{\Gamma' \vdash g\,@\,1 : \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_3]}\,\mathrm{app}}{[\,] \vdash \lambda g.\, g\,@\,1 : ((\texttt{int} \to \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_4]) \to \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_2])\,@\mathrm{cps}[\_, \_, \epsilon]}\,\mathrm{fun}$$

f に渡される関数を表す g は $(\texttt{int} \to \tau_2\,@\mathrm{cps}[\tau_3, \tau_4, \alpha_4])$ という型を持っており, それに伴って

$$(\tau_3 \neq \tau_4 \Rightarrow \alpha_4 = i)$$

という制約が生成されている. ここで τ_3 と τ_4 が異なるかどうかはすぐには判断できない. 上記のプログラムの場合は, f に渡される関数が $\lambda x.\, x$ という pure な関数であるから $\tau_3 = \tau_4$ とすれば α_4 は ϵ にできる.

しかし f が in 以下で $f\,@\,(\lambda x.\, Sk.\, x)$ のように impure な関数を渡されたとすると $\tau_3 \neq \tau_4$ となり α_4 は i でなくてはならない. つまり, f に渡される関数を pure とできるかどうかは in 以下で f がどのように使われるかに依存する. これは, let 多相の考え方と相反するものである. Let 多相で

$$
\begin{array}{rcl}
C \cup \{\alpha = i\} & \Rightarrow & C[i/\alpha] \\
C \cup \{\alpha = \epsilon\} & \Rightarrow & C[\epsilon/\alpha] \\
 & & \\
C \cup \{\epsilon \leq \epsilon\} & \Rightarrow & C \\
C \cup \{\epsilon \leq i\} & \Rightarrow & C \\
C \cup \{\epsilon \leq \alpha_2\} & \Rightarrow & C \\
C \cup \{i \leq \epsilon\} & \Rightarrow & \text{型エラーなので終了} \\
C \cup \{i \leq i\} & \Rightarrow & C \\
C \cup \{i \leq \alpha_2\} & \Rightarrow & C \cup \{\alpha_2 = i\} \\
C \cup \{\alpha_1 \leq \epsilon\} & \Rightarrow & C \cup \{\alpha_1 = \epsilon\} \\
C \cup \{\alpha_1 \leq i\} & \Rightarrow & C \\
C \cup \{\alpha_1 \leq \alpha_2\} & \Rightarrow & \text{自明でないので制約を残す}
\end{array}
$$

$$
\begin{array}{rcl}
C \cup \{\alpha_1 \neq \alpha_2 \Rightarrow \epsilon = i\} & \Rightarrow & C \cup \{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_1\} \\
C \cup \{\alpha_1 \neq \alpha_2 \Rightarrow i = i\} & \Rightarrow & C \\
C \cup \{\epsilon \neq \epsilon \Rightarrow \alpha = i\} & \Rightarrow & C \\
C \cup \{\epsilon \neq i \Rightarrow \alpha = i\} & \Rightarrow & C \cup \{\alpha = i\} \\
C \cup \{\epsilon \neq \alpha_2 \Rightarrow \alpha = i\} & \Rightarrow & C \cup \{\alpha_2 \neq \epsilon \Rightarrow \alpha = i\} \\
C \cup \{i \neq \epsilon \Rightarrow \alpha = i\} & \Rightarrow & C \cup \{\alpha = i\} \\
C \cup \{i \neq i \Rightarrow \alpha = i\} & \Rightarrow & C \\
C \cup \{i \neq \alpha_2 \Rightarrow \alpha = i\} & \Rightarrow & C \cup \{\alpha_2 \neq i \Rightarrow \alpha = i\} \\
C \cup \{\alpha_1 \neq \epsilon \Rightarrow \alpha = i\} & \Rightarrow & \text{自明でないので制約を残す} \\
C \cup \{\alpha_1 \neq i \Rightarrow \alpha = i\} & \Rightarrow & \text{自明でないので制約を残す} \\
C \cup \{\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i\} & \Rightarrow & \text{自明でないので制約を残す}
\end{array}
$$

図 4. 制約の解消規則

は, `let` $f = \lambda g.\, g\,@\,1$ `in` の部分だけを見て f の引数となる関数の型を含む f そのものの型を定め, generalize した上で in 以下で f を使う. しかし, 注釈の部分は, in 以下でどのように f が使われているかを見ない限り決められない. 本稿ではこのような場合, 型変数は自分以外の型変数や他の具体的な型とは異なると捉えて, $(\tau' \neq \tau'' \Rightarrow \alpha = i)$ という制約を $(\alpha = i)$ という制約に変換するという方針をとる. これは, 上の例で言えば f の型を

$$\forall\, \tau_2\, \tau_3\, \tau_4.\, ((\mathtt{int} \to \tau_2\, @\mathrm{cps}[\tau_3, \tau_4,\, i]) \to \tau_2\, @\mathrm{cps}[\tau_3, \tau_4,\, i])$$

とすることを意味している. この選択をすることで, より良い解を捨てている可能性がある. 一方, この選択で, in 以下での f の使われ方を見てから f に渡される関数が pure かどうかを判断するという, let 多相の考え方と相反する解き方を避けている.

以上が, 制約を注釈に関する制約のみに変換する方法の説明である. これを規則にしたものは著者の修士論文 [14] に載っている.

3.2 自明な制約を解消する

制約が注釈に関する制約のみになったら, 次に自明な制約解消を行う. その規則を図 4 に示す. 最初の 2 つは, 等式制約は代入を実際に行うことで制約解消を行うことを示している. その後の規則は, $(\_ \leq \_)$ の形の制約解消である. 注釈の代入を行った結果 $(i \leq \epsilon)$ が出てきてしまったら, 型エラーである. 図 2 の型規則では, ϵ は $(\_ \leq \_)$ の形の左側にしか出てこない. そのため $(i \leq \epsilon)$ の場合は通常では起こり得ない. しかし後に触れるライブラリ関数等を導入しようとする際には, 型規則の中の関数型に ϵ が入っている場合がある. すると ϵ が $(\_ \leq \_)$ の形の右側に出て来る場合がある. 最後の $(\alpha_1 \leq \alpha_2)$ の場合は, 現時点では制約解消できないためそのまま残す.

右の列は $(\_ \neq \_ \Rightarrow \_ = i)$ の形の制約の場合である. 基本的に

- $\Rightarrow$ の左側が成り立つ制約は $\Rightarrow$ の右側のみの制約に変換
- $\Rightarrow$ の左側が成り立たない制約は削除
- $\Rightarrow$ の右側が成り立つ制約も削除
- $\Rightarrow$ の右側が成り立たない制約は, 左側が成り立たないという制約に変換

という操作を行っている. 最初の規則は α_1 と α_2 が等しいことを $(\_ \leq \_)$ という形の制約で表現している. 最後の 3 つは, 現時点では制約解消できないので, そのまま残す.

以上を繰り返し行うと, 最終的に残る制約の形は

$$(\alpha_1 \neq \epsilon \Rightarrow \alpha = i) \qquad (\alpha_1 \neq i \Rightarrow \alpha = i) \qquad (\alpha_1 \neq \alpha_2 \Rightarrow \alpha = i) \qquad (\alpha_1 \leq \alpha_2)$$

となる. 図 4 で示した変換が解の集合を変化させないことは自明である.

3.3 自明でない制約を解消する

最後に残った 4 種類の制約のうち, 最初の 3 つの解消は簡単ではない. 任意の命題 A, B において $A \Rightarrow B$ は not A or B と書けるため, 最初の 3 つの制約は以下のように書き直すことができる.

$$\alpha_1 = \epsilon \text{ or } \alpha = i \qquad \alpha_1 = i \text{ or } \alpha = i \qquad \alpha_1 = \alpha_2 \text{ or } \alpha = i$$

このような制約が存在すると, どちらをとって良いか分からず, 単純に制約解消をすることはできない. 全数探索などの方法は考えられるが, そもそも第 3.5 節で紹介するように, 最も良い解は一般には存在しないため, 全数探索してもどれを選べば良いのかは明らかではない. 更に, 最初の制約で, より ϵ の多い解を求めたいために不用意に $\alpha_1 = \epsilon$ を選択すると, 場合によってはそれが他のところで他の制約と衝突することも考えられる. 例えば

$$\alpha_1 = \epsilon \text{ or } \alpha = i \qquad \alpha_1 = i \text{ or } \alpha_2 = i \qquad \alpha_2 \leq \alpha_1 \qquad \alpha_1 \leq \alpha_2$$

となっていたとする. 最初の制約で $\alpha_1 = \epsilon$ を選ぶと, 2 つ目の制約で $\alpha_2 = i$ となる. すると, 3 つ目の制約で $\alpha_1 = i$ となり最初の $\alpha_1 = \epsilon$ と衝突する. しかし, $\alpha, \alpha_1, \alpha_2$ に i を入れればすべての制約を満たすので解は存在する. つまり, 貪欲に $\alpha_1 = \epsilon$ を選ぶことはできないことになる.

以上の考察を踏まえて, 本稿では安全な方法として最初の 3 つの制約は全て $\alpha = i$ として解消するという方針を取る. この選択も, より良い解があるのに, それを捨てている可能性がある. 一方で, 解を全数探索に行く等のことをすることなく解を求めることができるようになっている.

なお, 実際に, ここで全数探索をしていればより良い解が見つかるような例を作ることができる. しかし, かなり人為的な例 [14] となり, これまで我々が試した限られた実用的な例の中では, このような例に遭遇していない. 全数探索など, より強力な制約解消が必要かどうかを考察することは今後の課題である.

3.4 以上で確定しなかった注釈は ϵ とする

これまでの制約解消の結果, 残った制約は $\alpha_1 \leq \alpha_2$ の形のものだけである. この形の制約は, この時点で確定していない注釈に全て ϵ を入れて解消する. こうしてなるべく pure な解を求めている.

3.5 最も良い解が存在しない例

最も良い解が存在しない例として, 以下の例がある.

$$\lambda f.\, \lambda g.\, (\langle (f\,@\,1) + (g\,@\,2) \rangle = \texttt{true})$$

これは, 関数 $f\,@\,1$ と $g\,@\,2$ の結果を加えたら, それが `true` になるかどうかを真偽値で返す関数である. 加えた結果が `true` になっているということは, f あるいは g の少なくともどちらかが answer type を `int` から `bool` に変える関数である必要がある. ここで, f と g の型は以下のようになる.

$$f : (\texttt{int} \to \texttt{int}\,@\mathrm{cps}[\tau_1, \texttt{bool}, \alpha_1]) \qquad g : (\texttt{int} \to \texttt{int}\,@\mathrm{cps}[\texttt{int}, \tau_1, \alpha_2])$$

f は answer type を τ_1 から `bool` に変え, g は `int` から τ_1 に変えるような関数となる. 左から実行するとして, f の中で shift が実行されると継続として $\lambda y.\, \langle y + g\,@\,2 \rangle$ が取られるが, この継続は, g の中で shift が実行されると answer type として τ_1 を返す. f の中でその τ_1 を `bool` に変える. 全体として, answer type は `int` から `bool` になるので, 上の式には型が付く.

f と g に対して以下の制約が生成される.

$$(\tau_1 \neq \texttt{bool} \Rightarrow \alpha_1 = i) \qquad (\texttt{int} \neq \tau_1 \Rightarrow \alpha_2 = i)$$

本稿が示す制約解消では, 型変数 τ と int や bool は常に異なるとするので, 結局, $\alpha_1 = \alpha_2 = i$ となり, f も g も impure となる. しかし, 実際には両方とも impure である必要はなく, f と g のどちらか片方が impure であれば十分である. この 2 つの解, つまり f のみが impure な解と g のみが impure な解は, 前者では $\alpha_1 = i, \alpha_2 = \epsilon$ なのに対し, 後者は $\alpha_1 = \epsilon, \alpha_2 = i$ となっており, どちらがより良いかを比べることのできない解になっている.

なお, f のみが impure な解と g のみが impure な解が, この例における一番 ϵ が多い解であるから, この 2 つの解が「極大の解」である. この例は, 本稿が複雑な型推論と制約解消法を避けているために「極大の解」が得られない場合があることを意味する.

4 Selective CPS 変換

第 3 節で制約解消が済み, 式の各部に ϵ か i の注釈が付いたら, それに従って i の注釈が付いている箇所のみ CPS 変換するのは比較的簡単である. その定義を図 5 に示す. 基本的には, Danvy and Filinski [4] による administrative redex を変換時に簡約する形の CPS 変換を, selective に拡張したものである. ここで, オーバーラインは CPS 変換時に実行される式, アンダーラインは CPS 変換結果として返される構文である. 変換結果の構文は, 標準的な let 多相の入った λ 計算である. 従って, 構文を変換すればそのまま OCaml などの処理系で実行することができる.

図 5 の変換は, 変換される式の一番外側の注釈が i か ϵ かによって 2 種類に分かれている. 基本的に, impure の場合は通常の CPS 変換を行い, pure の場合は直接形式のまま返す. Impure な式の中から pure な式を呼び出す場合は, その時点での継続 k に pure な式の変換結果をそのまま渡す格好になる. 逆に pure な式の中から impure な式を呼び出すのは reset の場合しかあり得ず, その際には恒等関数を継続として渡す.

最終的に, pure な式 e^{ϵ} の selective CPS 変換は

$$\| e^{\epsilon} \|_{\epsilon}$$

となり, impure な式 e^{i} の selective CPS 変換は

$$\underline{\lambda} k. \| e^{i} \|_{i} \ \overline{@}\ (\overline{\lambda} m. k \,\underline{@}\, m)$$

となる.

本稿では簡約規則を示していないが, 図 2 の型規則は注釈を無視した簡約規則のもとで subject reduction を満たすことを証明できる. その上で, 図 5 で示す selective CPS 変換は以下の定理を満たす. ただし, E はアンダーラインの付いた式, つまり CPS 変換後の式の任意の評価文脈である.

定理 4.1

$$e^{\epsilon} \to^{*} v^{\epsilon} \wedge \vdash e^{\epsilon} : t_1 \,@\mathrm{cps}[t_2, t_2, \epsilon] \ \Rightarrow\ \| e^{\epsilon} \|_{\epsilon} \to^{*} \| v^{\epsilon} \|_{\epsilon}$$

$$e^{i} \to^{*} v^{\epsilon} \wedge \vdash e^{i} : t_1 \,@\mathrm{cps}[t_2, t_2, i] \ \Rightarrow\ (\underline{\lambda} k. \| e^{i} \|_{i} \ \overline{@}\ (\overline{\lambda} m. k \,\underline{@}\, m)) \,\underline{@}\, (\underline{\lambda} x. E[x]) \to^{*} E[\| v^{\epsilon} \|_{\epsilon}]$$

これは, 注釈のついたプログラムの実行結果と selective CPS 変換後のプログラムの実行結果が対応することを示す. この定理は Danvy and Filinski [4] の証明に沿った形で示すことができる [14].

また, 本稿の注釈付きの型と selective CPS 変換後の, 型と環境の変換を以下に定義する.

定義 4.2 型と環境の変換 *

$$\texttt{bool}^{\star} = \texttt{bool} \qquad \texttt{int}^{\star} = \texttt{int} \qquad \tau^{\star} = \tau \qquad (\forall \tau. T)^{\star} = \forall \tau^{\star}. T^{\star}$$

$$(t_1 \to t_2 \,@\mathrm{cps}[t_3, t_3, \epsilon])^{\star} = ({t_1}^{\star} \to {t_2}^{\star}) \qquad (t_1 \to t_2 \,@\mathrm{cps}[t_3, t_4, i])^{\star} = ({t_1}^{\star} \to (({t_2}^{\star} \to {t_3}^{\star}) \to {t_4}^{\star}))$$

$$[]^{\star} = [] \qquad (\Gamma, x^{\epsilon} : t)^{\star} = \Gamma^{\star}, x : t^{\star}$$

$\| e^{\,i} \|_i$: Transformation of $e^{\,i}$ to continuation passing style

$$
\begin{array}{rcl}
\| (v_1{}^{\epsilon} \mathbin{@^i} e_2{}^{i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, (\| v_1{}^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} v_2) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v)) \\
\| (e_1{}^{\epsilon} \mathbin{@^i} e_2{}^{i})^{\,i} \|_i & = & \overline{\lambda} k.\, (\underline{\lambda} v_1.\ \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, (v_1 \mathbin{\underline{@}} v_2) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v))) \mathbin{\underline{@}} \| e_1{}^{\,\epsilon} \|_\epsilon \qquad (e_1 \text{ is NOT value}) \\
\| (e_1{}^{\,i} \mathbin{@^i} e_2{}^{\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, (v_1 \mathbin{\underline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v)) \\
\| (e_1{}^{\epsilon} \mathbin{@^i} e_2{}^{\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, (\| e_1{}^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \\
\| (e_1{}^{\,i} \mathbin{@^i} e_2{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, (v_1 \mathbin{\underline{@}} v_2) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v))) \\
\| (v_1{}^{\epsilon} \mathbin{@^\epsilon} e_2{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, k \mathbin{\overline{@}} (\| v_1{}^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} v_2)) \\
\| (e_1{}^{\epsilon} \mathbin{@^\epsilon} e_2{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, (\underline{\lambda} v_1.\ \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, k \mathbin{\overline{@}} (v_1 \mathbin{\underline{@}} v_2))) \mathbin{\underline{@}} \| e_1{}^{\,\epsilon} \|_\epsilon \qquad (e_1 \text{ is NOT value}) \\
\| (e_1{}^{\,i} \mathbin{@^\epsilon} e_2{}^{\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, k \mathbin{\overline{@}} (v_1 \mathbin{\underline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon)) \\
\| (e_1{}^{\,i} \mathbin{@^\epsilon} e_2{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, \| e_2{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_2.\, k \mathbin{\overline{@}} (v_1 \mathbin{\underline{@}} v_2))) \\
\| (e_1{}^{\epsilon} \mathbin{@^\epsilon} e_2{}^{\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, k \mathbin{\overline{@}} (\| e_1{}^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon) \\
\| (Sx.e^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{let}}\ x \underline{=} \underline{\lambda} a.\, (k \mathbin{\overline{@}} a)\ \underline{\mathtt{in}}\ (\| e^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} y.y)) \\
\| (Sx.e^{\,\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{let}}\ x \underline{=} \underline{\lambda} a.\, (k \mathbin{\overline{@}} a)\ \underline{\mathtt{in}}\ \| e^{\,\epsilon} \|_\epsilon \\
\| (\mathtt{let}\ x = v^{\,\epsilon}\ \mathtt{in}\ e^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{let}}\ x \underline{=} \| v^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{in}}\ (\| e^{\,i} \|_i \;\overline{@}\, k) \\
\| (\mathtt{if}\ e_1{}^{\,i}\ \mathtt{then}\ e_2{}^{\,i}\ \mathtt{else}\ e_3{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, \underline{\mathtt{if}}\ v_1\ \underline{\mathtt{then}}\ \| e_2{}^{\,i} \|_i \;\overline{@}\, k\ \underline{\mathtt{else}}\ \| e_3{}^{\,i} \|_i \;\overline{@}\, k) \\
\| (\mathtt{if}\ e_1{}^{\,\epsilon}\ \mathtt{then}\ e_2{}^{\,i}\ \mathtt{else}\ e_3{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{if}}\ \| e_1{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{then}}\ \| e_2{}^{\,i} \|_i \;\overline{@}\, k\ \underline{\mathtt{else}}\ \| e_3{}^{\,i} \|_i \;\overline{@}\, k \\
\| (\mathtt{if}\ e_1{}^{\,i}\ \mathtt{then}\ e_2{}^{\,\epsilon}\ \mathtt{else}\ e_3{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, \underline{\mathtt{if}}\ v_1\ \underline{\mathtt{then}}\ k \mathbin{\overline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{else}}\ \| e_3{}^{\,i} \|_i \;\overline{@}\, k) \\
\| (\mathtt{if}\ e_1{}^{\,i}\ \mathtt{then}\ e_2{}^{\,i}\ \mathtt{else}\ e_3{}^{\,\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.\, \underline{\mathtt{if}}\ v_1\ \underline{\mathtt{then}}\ \| e_2{}^{\,i} \|_i \;\overline{@}\, k\ \underline{\mathtt{else}}\ k \mathbin{\overline{@}} \| e_3{}^{\,\epsilon} \|_\epsilon) \\
\| (\mathtt{if}\ e_1{}^{\,\epsilon}\ \mathtt{then}\ e_2{}^{\,\epsilon}\ \mathtt{else}\ e_3{}^{\,i})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{if}}\ \| e_1{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{then}}\ k \mathbin{\overline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{else}}\ \| e_3{}^{\,i} \|_i \;\overline{@}\, k \\
\| (\mathtt{if}\ e_1{}^{\,\epsilon}\ \mathtt{then}\ e_2{}^{\,i}\ \mathtt{else}\ e_3{}^{\,\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \underline{\mathtt{if}}\ \| e_1{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{then}}\ \| e_2{}^{\,i} \|_i \;\overline{@}\, k\ \underline{\mathtt{else}}\ k \mathbin{\overline{@}} \| e_3{}^{\,\epsilon} \|_\epsilon \\
\| (\mathtt{if}\ e_1{}^{\,i}\ \mathtt{then}\ e_2{}^{\,\epsilon}\ \mathtt{else}\ e_3{}^{\,\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, \| e_1{}^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v_1.k \mathbin{\overline{@}} (\underline{\mathtt{if}}\ v_1\ \underline{\mathtt{then}}\ \| e_2{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{else}}\ \| e_3{}^{\,\epsilon} \|_\epsilon)) \\
\| (\mathtt{if}\ e_1{}^{\,\epsilon}\ \mathtt{then}\ e_2{}^{\,\epsilon}\ \mathtt{else}\ e_3{}^{\,\epsilon})^{\,i} \|_i & = & \overline{\lambda} k.\, k \mathbin{\overline{@}} (\underline{\mathtt{if}}\ \| e_1{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{then}}\ \| e_2{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{else}}\ \| e_3{}^{\,\epsilon} \|_\epsilon)
\end{array}
$$

$\| e^{\,\epsilon} \|_\epsilon$: Transformation of $e^{\,\epsilon}$ to direct style

$$
\begin{array}{rcl}
\| x^{\,\epsilon} \|_\epsilon & = & x \\
\| n^{\,\epsilon} \|_\epsilon & = & n \\
\| \mathtt{true}^{\,\epsilon} \|_\epsilon & = & \mathtt{true} \\
\| \mathtt{false}^{\,\epsilon} \|_\epsilon & = & \mathtt{false} \\
\| (\lambda^i x.e^{\,i})^{\,\epsilon} \|_\epsilon & = & \underline{\lambda} x.\, \underline{\lambda} k'.\ \| e^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v.\, k' \mathbin{\underline{@}} v) \\
\| (\lambda^i x.e^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\lambda} x.\, \underline{\lambda} k'.k' \mathbin{\underline{@}} \| e^{\,\epsilon} \|_\epsilon \\
\| (\lambda^\epsilon x.e^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\lambda} x.\, \| e^{\,\epsilon} \|_\epsilon \\
\| (e_1{}^{\,\epsilon} \mathbin{@^\epsilon} e_2{}^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \| e_1{}^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} \| e_2{}^{\,\epsilon} \|_\epsilon
\end{array}
\qquad
\begin{array}{rcl}
\| (\mathtt{fix}^i\ f.x.e^{\,i})^{\,\epsilon} \|_\epsilon & = & \underline{\mathtt{fix}}\ f.x.(\underline{\lambda} k'.\, \| e^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} v.\, k' \mathbin{\underline{@}} v)) \\
\| (\mathtt{fix}^i\ f.x.e^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\lambda} x.\, \underline{\lambda} k'.\, k' \mathbin{\underline{@}} (\| (\mathtt{fix}^\epsilon\ f.x.e^{\,\epsilon})^{\,\epsilon} \|_\epsilon \mathbin{\underline{@}} x) \\
\| (\mathtt{fix}^\epsilon\ f.x.e^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\mathtt{fix}}\ f.x.\, \| e^{\,\epsilon} \|_\epsilon \\
\| \langle e^{\,i} \rangle^{\,\epsilon} \|_\epsilon & = & \| e^{\,i} \|_i \;\overline{@}\, (\overline{\lambda} x.x) \\
\| \langle e^{\,\epsilon} \rangle^{\,\epsilon} \|_\epsilon & = & \| e^{\,\epsilon} \|_\epsilon \\
\| (\mathtt{let}\ x = v^{\,\epsilon}\ \mathtt{in}\ e^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\mathtt{let}}\ x \underline{=} \| v^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{in}}\ \| e^{\,\epsilon} \|_\epsilon \\
\| (\mathtt{if}\ e_1{}^{\,\epsilon}\ \mathtt{then}\ e_2{}^{\,\epsilon}\ \mathtt{else}\ e_3{}^{\,\epsilon})^{\,\epsilon} \|_\epsilon & = & \underline{\mathtt{if}}\ \| e_1{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{then}}\ \| e_2{}^{\,\epsilon} \|_\epsilon\ \underline{\mathtt{else}}\ \| e_3{}^{\,\epsilon} \|_\epsilon
\end{array}
$$

図 5. Selective CPS 変換

この時, 以下が式 e の derivation に関する帰納法で証明できる [14].

定理 4.3 Type Preservation

$$\Gamma \vdash e^{\epsilon} : t \,@\mathrm{cps}[t_1, t_1, \epsilon] \;\Rightarrow\; \Gamma^{\star} \vdash \| e^{\epsilon} \|_{\epsilon} : t^{\star}$$
$$\Gamma \vdash e^{i} : t \,@\mathrm{cps}[t_1, t_2, i] \;\Rightarrow\; \Gamma^{\star} \vdash \underline{\lambda} k.\ \| e^{i} \|_{i}\ \overline{@}\,(\overline{\lambda} m.\, k \,\underline{@}\, m) : ((t^{\star} \to {t_1}^{\star}) \to {t_2}^{\star})$$

5 実行例

本研究で提案するシステム, つまり型推論, 制約解消そして selective CPS 変換を実装した. この実装したシステムは, answer type が変わるようなものも扱えるようになっている.

その上で, 本稿で示した selective CPS 変換の効果を見るため, shift/reset を使って具体的なプログラムを書いた. それを selective CPS 変換した結果と CPS 変換した結果の実行時間を計測した. 具体的なプログラムは, prefix 関数と N-Queen 問題を解く関数の 2 つである. 以下順に説明する.

なお実行環境は MacOSX 10.9.4, メモリが 4GB, CPU が 1.3GHz の Intel Core i5 である. 変換結果のプログラムを OCaml 4.02.2 を使ってネイティブコードにコンパイルし, user time を計測した.

また, システムに直接手を加えて限定継続命令をサポートしている言語との比較はしない. Scheme 上での直接実装 [6] は動的型付けであるため OCaml と単純な比較はできないこと, OCaml の Delimcc ライブラリ [8] は直接は answer type modification を扱えないこと, Caml Light を拡張した OchaCaml [10] はネイティブコンパイラがないことが理由である.

5.1 Prefix 関数

Prefix 関数を shift/reset を使って非決定性プログラミングの形で記述した [9]. この関数は, 受け取ったリストのプリフィックスのリストを返す. 例えば `prefix [1; 2; 3]` は `[[1]; [1; 2]; [1; 2; 3]]` となる. プログラムは以下である.

```
let rec visit = match lst with
    [] -> shift k -> []
  | a :: rest -> a :: (shift k -> (k []) :: <k (visit rest)>) in
let prefix lst = <visit lst>
```

リストの次に付けるべき要素 `a` までを shift を使って継続 `k` として取り出す. `k []` で `a` までが付いたリストを, `<k (visit rest)>`で `a` 以降に `rest` のプリフィックスが付いた複数のリストを返す.

5.1.1 実行時間の比較

`prefix` の selective CPS 変換の結果と CPS 変換の結果にリストを渡して実行時間を計測した. 実行時間の計測結果を図 6 に示す. 図 6 の「Selective CPS」が, プログラムを selective CPS 変換した結果にリストを渡して実行した時にかかった時間, 「CPS」がプログラムを全て CPS 変換した結果にリストを渡して実行した時にかかった時間である. 渡すリストは要素の数字が全て 0 で, 長さが 750, 1000, 2500, 5000, 7500, 10000 の場合を実行している. 表から分かるように約 11% ほどの実行時間の改善が見られる.

The execution time (ms) of prefix lst whose length is 750, 1000, 2500, 5000, 7500, 10000, which is user time

	length is 750	length is 1000	length is 2500	length is 5000	length is 7500	length is 10000
selective CPS	16	24	177	771	1796	3233
CPS	18	37	199	842	2050	3618
selective CPS / CPS	0.89	0.89	0.89	0.92	0.88	0.89

図 6. `loop n` の実行時間の比較

5.1.2 変換後のプログラムの比較

`prefix` を CPS 変換した結果は以下である.

```
let rec visit = fun k' ->
  match x with [] -> let k = fun a' -> fun k'' -> k'' (k' a') in []
   | a :: rest ->
     let k = fun a' -> fun k'' -> k'' (k' (a :: a')) in
       (k []) (fun x -> x :: ((visit rest) ((fun x -> (k x) (fun x -> x))))) in
let prefix = fun lst -> fun k -> k ((visit lst) (fun x -> x))
```

また, selective CPS 変換した結果は以下である.

```
let rec visit = fun k' ->
  match x with [] -> let k = fun a' -> k' a' in []
   | a :: rest -> let k = fun a' -> k' (a :: a') in
                    (k []) :: ((visit rest) (fun x -> k x)) in
let prefix = fun lst -> (visit lst) (fun x -> x)
```

２つのプログラムの差異は, shift が切り取る継続 `k` が引数として継続を取るか否かである. Shift が捕捉した継続は pure であるため, selective な方では `k` への呼び出しが直接形式で行われる. また, リストと match 文を使っているが, 本稿の枠組にこれらをサポートするのは簡単である.

5.1.3 Queen 関数

本稿で示した selective CPS 変換の効果を見るため, N-Queen 問題を解くプログラムを shift/reset を使って非決定性プログラミングの形で記述し [9], その実行時間を計測した.

最初に, 1 から n までを非決定的に選択する関数 `choice` を定義する. この関数は, 現在の継続 `k` を切り取ってきて, `k` に値 `num` を渡した後, さらに `k` に `j-1` 以下の値について再帰した値を渡すため, 実質的に `num` から `1` までの数を非決定的に返す関数とみなせる [9].

```
(* 引数として数字 num を受け取り, num ~ 1 を順番に返す関数. *)
let rec choice num =
  if num = 1 then 1 else shift k -> (k num; k (choice (num - 1)))
```

N-Queen 問題を解くメインルーチンは以下のようになる. ここで `print_solution` は解が見つかった時にそれを表示する関数, また `is_safe` は Queen を他の Queen とぶつかることなく置けるかをチェックする関数で, いずれも別途, 定義する必要がある.

The execution time (ms) of queen n which is user time

	queen 8	queen 9	queen 10	queen 11	queen 12	queen13	queen14	queen15
selective CPS	3	9	34	184	1074	6812	45807	329027
CPS	5	12	48	254	1490	9524	63450	458004
selective CPS / CPS	0.6	0.75	0.71	0.72	0.72	0.72	0.72	0.72

図 7. queen n の実行時間の比較

```
(* 引数として数字 n を受け取り，n*n の Queen 問題を解いた答えを出力する関数 *)
let queen = fun n ->
  (* loop 関数は，i 列目以前，Queen を何行目に入れるかを決定 *)
  (* 引数 solution は，i - 1 列目以降，各列の何行目に Queen を入れるかのリスト *)
  let rec loop (i, solution) =
    if i = 0 then print_solution solution  (* 答えを出力 *)
    else
      let j = choice n in
      let solution2 = j :: solution in
        if is_safe solution2                (* i 列目は j 行目が答えになる場合 *)
          then loop (i - 1, solution2)      (* i - 1 列目以前の答えを求める *)
          else ()                           (* choice で別の j を選んで続ける *)
  in
<loop (n, [])> in

(queen n) (* n は引数となる数字 *)
```

choice 関数がバックトラックを全て引き受けるため, queen 関数は単に, Queen を置く場所を順に選び, 最後まで成功したら結果を表示するという処理を記述すれば良い. プログラム中に出てくる let 文は定義される式が値ではないため, 多相の let 文ではなく単相の let 文, つまり fun 文と関数呼び出しとして扱われる.

queen 関数が, 2 つの引数をペアにして受け取ってくるのは, カリー化して順番に受け取るようにすると, そのたびに CPS 変換されてしまい, 全体を CPS 変換する方法にとって著しく不利になるためである. Selective に CPS 変換していれば, 1 つ目の引数を渡す部分は pure になるため CPS 変換されず, 従って非効率にはならない.

5.1.4 実行時間の比較

queen のプログラムを, プログラム全体を CPS 変換した場合と, 本稿に示す selective CPS 変換をした場合の 2 通りについて実行し, その実行時間を計測した. その結果を図 7 に示す. 図 7 の「Selective CPS」が, プログラムを selective CPS 変換して queen n を実行した時にかかった時間,「CPS」がプログラムを全て CPS 変換して queen n を実行した時にかかった時間である. 表から分かるように約 28% ほどの実行時間の改善が見られる.

5.1.5 変換後のプログラムの比較

choice を CPS 変換した結果は以下のようになる.

```
let rec choice j = fun k' ->
```

```
  if j = 1 then k' 1
  else let k = fun a -> fun k'' -> k'' (k' a) in
       (k j) (fun x -> (choice (j - 1)) (fun x' -> (k x') (fun x'' -> x; x'')))
```

また, selective CPS 変換した結果は以下である.

```
let rec choice = fun k' ->
  if j = 1 then k' 1
  else let k = fun a -> k' a in (k j; (choice (j - 1)) (fun x' -> k x')))
```

choice は中で shift を使っているため, selective CPS 変換でも基本的には CPS 変換する必要がある. しかし, selective な方では, 捕捉した継続が pure であることを利用して k への呼び出しが直接形式で行われるようになっている.

次に, 関数 queen を比べる. CPS 変換した結果を載せて, selective CPS 変換した結果が違う場合は, CPS 変換の結果 selective CPS 変換の結果 と書く. CPS 変換の結果が selective CPS 変換の結果に含まれない場合は, selective CPS 変換の結果を書かないで, CPS 変換の結果 だけ書く.

```
let queen = fun n -> fun kk ->
  let rec loop (i, solution) =
    fun k -> if i = 0 then (print_solution solution) (fun x -> k x)
                           k (print_solution solution)
             else (choice n)
                  (fun x ->                    (* x = choice n *)
                    ((fun j ->                 (* j = x *)
                       fun k' ->               (* k' = fun x -> k x *)
                         ((fun solution2 ->    (* solution2 = j :: solution *)
                            fun k'' ->         (* k'' = fun x -> k' x *)
                            (is_safe solution2) (fun x -> if x
                            if is_safe solution2
                               then (loop (i - 1, solution2)) (fun x -> k'' x)
                               else k'' () ) )
                          (j :: solution)) (fun x -> k' x))
                     x) (fun x -> k x))
  in
 (kk ((loop (n, [])) (fun x -> x)) )
```

変換結果の違いは, 以下の 3 点である.

- 上記のコードでは省略したが, print_solution と is_safe は pure な関数であるため selective CPS 変換では CPS に変換されない.
- 従って queen の中からこれらの関数を呼び出す部分も selective な方では直接形式で呼び出す.
- loop 関数は selective な方でも CPS 変換されているが, queen 関数自体は CPS 変換されていない. これは loop 関数の定義は pure であり, loop 関数の呼び出しも reset の中に入っているため pure であることが理由である.

最後に queen 関数の呼び出し部分を比べる. 全てを CPS 変換する方では (queen n) (fun x -> x) となるが, selective な方では queen n となる. これは queen が pure と判定されたためである.

6 ライブラリ関数の導入

CPS 変換の結果は shift/reset を含まないプログラムになるため, OCaml など一般の処理系で実行できる. このことから, OCaml に既に存在するライブラリ関数の使用を考えることができる.

ライブラリ関数を使うには 2 通りの方法が考えられる. 1 つは, ライブラリ関数は CPS 形式では書かれていないため, 全て pure とする方法である. 例えば List.map は $(t_1 \to t_2\,) \to t_1\,\texttt{list} \to t_2\,\texttt{list}$ という型を持つが, これに次のような pure な注釈を付ける.

$$\frac{}{\Gamma \vdash \mathrm{map} : ((t_1 \to t_2\ @\mathrm{cps}[t_3, t_3,\ \epsilon]) \to (t_1\,\texttt{list} \to t_2\,\texttt{list}\ @\mathrm{cps}[t_4, t_4,\ \epsilon])\ @\mathrm{cps}[t_5, t_5,\ \epsilon])\ @\mathrm{cps}[t_6, t_6,\ \epsilon]}\ \mathrm{map}$$

このようにしておくと, List.map は必ず pure となり, 従って OCaml の List.map をそのまま使うことができるようになる. この場合, 制約解消の段階で $(\_ \leq \_)$ の右側に ϵ が入ることが起こり得る.

もう 1 つは, List.map を定義し直して, 本稿の型システムで型を付け直す方法である. このようにすると List.map に impure な関数も渡せるようになるが, OCaml に用意されている List.map を使うことはできなくなる.

7 関連研究

Rompf ら [13] は本稿と異なり, selective CPS 変換をするための注釈をユーザが与えるように定義している. そのため, ユーザが適切な注釈を与えた場合, 対象言語の「最善の解」や「極大の解」を得られる. その代わり, 複雑なプログラムであると, ユーザが注釈を自力で付けることが困難な場合がある. 本稿は, 制約解消法を定義しているため, 「最善の解」や「極大の解」を得られない場合がある代わりに, 注釈を自力で付けるというユーザの負担を軽くしている.

Kim and Yi and Danvy [7] は, 本稿が shift/reset が影響する部分だけを CPS 変換するのに対し, 例外が起こる箇所だけを CPS 変換するという selective CPS 変換を定義した. また, Reppy の local CPS 変換 [12] は, 直接形式であるプログラム中の, 特定の関数呼び出しのみを CPS 変換することにより, ネストしたループ等のコンパイル結果の効率を上げる.

Danvy and Hatcliff [5] は, call by name のプログラムの CPS 変換を定義した. まず, strictness analysis によって call by name の影響を受ける箇所に注釈を付け, 注釈に沿ってプログラムを遅延評価を含む call by value のプログラムに変換する. そして, 遅延評価を含む call by value のプログラムの CPS 変換をする. これは selective CPS 変換ではないが, 本稿と同じように, 注釈に沿ってプログラムを selective に変換する.

8 まとめ

本稿では shift/reset を含むプログラムの selective CPS 変換を定義した. 具体的には以下を行った.

- プログラムに CPS 変換が必要かどうかの情報を注釈として含む型規則を定義した.
- 型推論で生成された制約に対する, 制約の解消法を定義した. この規則は, 過度に型推論と制約解消を複雑にすることなく, ある程度の解を得られるものである.
- 注釈が付いた項に対する selective CPS 変換を定義した.

ユーザの視点からみると, ユーザは注釈が付いていないプログラムを本研究が提案するシステムに入力すれば, システムが自動で注釈を付ける. そしてプログラムの selective CPS 変換の結果が出力として得られる.

今のところ, 最善の解が得られない例は, 一般的にはあまり使わない. よって現時点では, 制約解消の性能を上げても, 一般的なプログラムの性能はさほど上がらない. ただ, 我々は限られたプログ

ラムしか考察できていない. そのため, 本稿の制約解消法よりも強力な制約解消法が必要なプログラム, その中でも特に一般的なものを探すことは, 今後の課題である.

また, 制約解消法を考える上で, プログラムの中で CPS 変換をするかしないかで大きく実行時間が変わる箇所, つまり注釈の判断がシビアな箇所を断定することは大切である. 現段階でそのような箇所の断定には至っておらず, 今後の課題である.

なお, selective CPS 変換の前後を比べると, 構文は異なるが同じ意味のプログラムになっており, 実行速度以外にも, 検証を通るかなど変換の前後で違う結果を出すものは色々あり得る. このことから, selecitve CPS 変換の潜在的な応用例を本稿で述べたもの以外で見つけることも今後の課題である.

謝辞

本研究において, 筑波大学の亀山幸義教授に, 熱心なご指導と多くのご助言を頂きました. 深く感謝致します. また, 国立情報学研究所の対馬かなえ助教に, 本研究において非常に重要である, 第 3.5 節の最も良い解が存在しない例を考案して頂きました. 深く感謝致します. そして, 多くの有益なコメントを下さった査読者の皆様に深く感謝致します.

参考文献

[1] Appel, A. W.: Compiling with Continuations, Cambridge University Press, 1992

[2] Asai, K. and Kameyama, Y.: Polymorphic Delimited Continuations, *5th Asian Symposium on Programming Languages and Systems (APLAS 2007)*, pp.239–254, November 2007

[3] Danvy, O. and Filinski, A.: Abstracting control, *the 1990 ACM Conference on Lisp and Functional Programming*, pp.151–160, 1990

[4] Danvy, O. and Filinski, A.: Representing Control : a Study of the CPS Transformation, *Mathematical Structures in Computer Science*, Volume 2, Issue 4, pp.361–391, December 1992

[5] Danvy, O. and Hatcliff, J.: CPS transformation after strictness analysis, *ACM Letters on Programming Languages and Systems*, Volume 1, Issue 3, pp.195–212, September 1992

[6] Gasbichler, M. and Sperber, M.: Final Shift for Call/cc: Direct Implementation of Shift and Reset *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp.271–282, October 2002

[7] Kim, J. and Yi, K. and Danvy, O.: Assessing the Overhead of ML Exceptions by Selective CPS Transformation, *the 1998 ACM SIGPLAN Workshop on ML*, pp.103–114, September 1998

[8] Kiselyov, O.: Delimited Control in OCaml, Abstractly and Concretely, System Description *Functional and Logic Programming (LNCS 6009)*, pp.304–320, April 2010

[9] 増子 萌: MinCaml コンパイラにおける shift/reset の実装, 第１１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論文集, pp.163–177, 2009

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

[11] Nielsen, L. R.: A Selective CPS Transformation, BRICS Research Report RS-01-30, Department of Computer Science, Aarhus University, 2001

[12] Reppy, J.: Local CPS conversion in a direct-style compiler, *the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, pp.13–22, January 2001

[13] Rompf, T. and Maier, I. and Odersky, M.: Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform, *The 2009 ACM SIGPLAN International Conference On Functional Programming*, pp.317–328, 2009

[14] 上原 千裕: 限定継続命令 shift/reset の Selective CPS 変換, お茶の水女子大学修士論文 (to appear), 2017