日本ソフトウェア科学会第 32 回大会 (2015 年度) 講演論文集

参照を備えた多段階計算のための多相的型システム

小林 恵　五十嵐 淳

本稿では，参照・let 多相を備えた多段階計算体系に対する複数の型システムを提案し比較を行う．参照と let 多相を型安全に組み合わせる手法としては値多相が広く使われているが，これを素朴に多段階計算に適用すると体系の型安全性が失われるという問題が指摘されている．これは，通常は値と考えられる式が多段階計算では計算が発生する式となる場合があることに起因している．我々は，Tofte による命令的型変数のアイデアを元にして，ふたつの型安全な型システムを構築する．ひとつめの型システムでは，命令的型変数が具体化されるステージを考慮することでより多くのプログラムに型がつくようにしている．ふたつめの型システムは，ひとつめの型システムより制限が厳しいが，素朴な値多相による型システムに近く，その型安全性からは，CSP と呼ばれる多段階計算の機能を使わない限り，素朴な値多相で型安全性を保証できることがいえる．

1　はじめに

多段階プログラミングとは，実行時にコードの生成や操作，実行ができるプログラミングパラダイムのことである．こういった多段階プログラミング言語の型システムでは，単に生成するプログラムが安全に実行できることだけでなく，生成されたコードの安全性も保証できることが望ましい．

このような型安全な多段階プログラミングをサポートしている言語のひとつに，OCaml の拡張である MetaOCaml [8] [5] がある．MetaOCaml では bracket, escape, run の 3 つの操作による実行時コード生成を行える．bracket (.<e>.) はコードを作る操作であり，bracket によって作られたコード内は run (!. e) で実行したときに計算され，その値を得ることができる．escape (.~e) は，bracket 内で使われ，コード中に e を計算した結果のコードを埋め込むことができる．また，**let** x = 1 + 2 **in** .<not (x > 2)>.のように，bracket の外側で宣言された変数 x やライブラリ関数 not を bracket 内で参照することができる．この，早く計算されるステージの変数を後で計算されるステージで使うことができる機能は cross-stage persistence(CSP) と呼ばれている．MetaOCaml の型システムの理論は，計算効果を持たない純粋なプログラムについてはよく研究されており，MetaOCaml をモデル化した体系に対してその型安全性が示されている [9] [1].

しかし，この MetaOCaml の型システムは，参照など計算効果のある言語機能と素朴に組み合わせると問題があることが知られている．ひとつは Scope Extrusion という問題で，コード中に含まれる変数がスコープから出てしまうという問題である．例えば，

```
# let r = ref .<1>. in
  let _ = .< fun x → .~( r := .<x + 2>.; .<3>.)
        >. in
  !. (!r);;
```

のようなプログラムでは，int 型のコードを格納する参照 r に x+2 という関数パラメータを含むコードを代入し，それを実行しようとしてしまうが，MetaOCaml の型システムはこれを受理してしまう．(MetaOCaml の処理系 BER MetaOCaml では実行時例外が発生

Polymorphic Type Systems for Multi-Stage Programming with References

Megumi Kobayashi, Atsushi Igarashi, 京都大学情報学研究科, Graduate School of Informatics, Kyoto University.

する．) もうひとつは Kiselyov と Shan が指摘した値多相の問題である [6]．以下の MetaOCaml のプログラムには型がつくが，これは実行時型エラーを起こしてしまう．

```
# let c = .<let f () = .~(let x = ref [] in .<x>.)
            in f():=[1]; "123" :: !(f()) >. ;;
val c : string list code = .<
  let f_1 () = (* CSP x *) in (f_1 ()) := [1]; "123"
      :: (! (f_1 ()))>.
# !. c;;
Segmentation fault
```

このプログラムでは，f が束縛される式は関数抽象であるため，OCaml の型システムは f は値に束縛されると判断し，(Relaxed) Value Restriction [12][2] に基づいて，f には多相型 unit → 'a list ref を与える．一方で，実行時には，空リストへの参照の生成，CSP による参照のコードへの埋め込みが，c が束縛されるコードの実行より前に起こる．結局，!. c が実行されると，fは先に生成された参照を返す定数関数に束縛され，その参照への整数リストの代入と文字列リストとしての読み出しが発生し，最終的に返ってくる値は不正な値 ["123"; 1] になってしまう．

本稿では，先ほど挙げた問題のうち後者の値多相に関する問題を扱う．具体的には，多段階計算の機能と参照を含む言語 MiniML$^{\triangleright\%}$ と，それに対する型システムをふたつ定義し，その性質について議論する．ひとつめの型システムは，Tofte の命令的型変数のアイデア [11] を改良したもので，命令的型変数が具体化されるステージを考慮することでより多くのプログラムに型がつくようにしている．ふたつめの型システムは，ひとつめの型システムより制限が厳しいが，素朴な値多相による型システムに近く，その型安全性からは，CSP と呼ばれる多段階計算の機能を使わない限り，素朴な値多相で型安全性を保証できることを目指している．

本稿の構成は以下のとおりである．2 節で MiniML$^{\triangleright\%}$ とふたつの型システムのアイデアについて説明する．3 節で MiniML$^{\triangleright\%}$ の文法・意味論の定義を 4 節でそれに対する型システムの定義とその性質について述べる．最後に 5 節で関連研究について述べ，6 節でまとめる．

2 MiniML$^{\triangleright\%}$ と型システムの概要

本節では，MiniML$^{\triangleright\%}$ の概要を説明し，我々の型システムのアイデアを議論する．

2.1 MiniML$^{\triangleright\%}$ の概要

MiniML$^{\triangleright\%}$ は，λ 計算を let 式，参照で拡張し，さらに，$\lambda^{\triangleright\%}$ [3] に基く (MetaOCaml 風) 多段階プログラミングのための機構を加えたような言語である．

項 M に対して，MetaOCaml でいう bracket や escape のような働きをする構築子が $\blacktriangleright_\alpha$, $\blacktriangleleft_\alpha$ である．これらに 添字付けられている α は遷移変数と呼ばれている．遷移変数には，遷移変数の列である遷移を代入することができ，その遷移変数の列によって $\blacktriangleright$, $\blacktriangleleft$ の個数が変化する．この遷移変数に空の列を代入した場合に， $\blacktriangleright$ などが消え，コード本体の項の評価を進めることができるようになる．また，$\blacktriangleright$ や $\blacktriangleleft$ の入れ子の深さをこの遷移で表すことができる．$\blacktriangleright_\alpha$ で α ぶん入れ子が増え，$\blacktriangleleft_\alpha$ で α ぶん入れ子が減ると考えると，これが項が評価される段階 (ステージ) を表している．この遷移の列は部分列の関係で比較することができ，大きい列が何らかの代入によって空の列になるとき，その部分列もその代入によって必ず空の列になる．

コードを表す型にもこの遷移変数のラベル付けがされており，$\triangleright_\alpha T$ は遷移変数 α でラベル付されたコードの型を表している．

上に述べたとおり，空列を遷移変数に代入することでコード中の項が評価可能になる．それではコードの実行をどう行うかというと，遷移変数の束縛と，遷移への適用を使って行う．また，遷移変数の束縛を行った $\Lambda\alpha.M$ を遷移に適用すると，項 M 中の束縛されている遷移変数 α に遷移変数に遷移が代入される．とくに空列に適用した場合は $\blacktriangleright$ がすべて消え，その中の項の評価を進めることができるようになる．

項 M に型 T がつくとき，$\lambda^{\triangleright\%}$ では，遷移変数を束縛した項 $\Lambda\alpha.M$ には $\forall\alpha.T$ という型が付く．こういった型の出現を let 多相による型変数の一般化とま

とめたいため，MiniML$^{\triangleright\%}$ では遷移変数を束縛できるのは let で局所定義される項のみに制限する．

% は，CSP のための，α ぶん後のステージに項を持ち上げる操作である．これは espcape のように何かと打ち消しあうわけではなく，型付けのためのものだと思ってもよい．(この体系や $\lambda^{\triangleright\%}$ では) コードの実行の際に空列が代入されて消えることになっている．

2.2 型システムのアイデア

本稿ではこの MiniML$^{\triangleright\%}$ のための型システムをふたつ提案するが，ひとつめの型システムは，Tofte の手法[11] を改良したものであり，ふたつめの型システムはそれをさらに修正したものとなっている．

Tofte の手法は，型変数を参照型に現れることのない作用的型変数と参照型に現れる可能性のある命令的型変数に分け，let によって変数が束縛される項が値でない場合は，一般化できる型変数を作用的型変数のみに制限することで健全な型システムを得ている．Tofte の手法だけでも型安全性を保証できるが，これだけでは多くの安全に実行できるプログラムがはじかれてしまうため，ひとつめの型システムでは，命令的型変数をステージごとに細分化することで，単純に Tofte の方法を適用するよりも多くのプログラムに型をつけられるようにする．

どういった場合に命令的型変数が一般化できるかを考える．前節のような問題が起きるプログラムを MiniML$^{\triangleright\%}$ で書くと，以下のようになる．

$$\begin{aligned}&\text{let } c = \Lambda\alpha.\blacktriangleright_\alpha(\text{let } f = \lambda x.\%_\alpha(\text{ref } (\lambda y.y)) \text{ in}\\ &\qquad (f \text{ unit}) := (\lambda x.(x+1));\\ &\qquad !(f \text{ unit})\ true)\end{aligned}$$

こういった一見値に見えるが実際には先に参照が確保されてしまう場合には，その参照に現れる型変数を一般化してはいけない．逆に，以下のようなものでは f の型を一般化してもよい．なぜなら，この例では参照が確保されることはないからである．

$$\begin{aligned}&\text{let } c = \Lambda\alpha.\blacktriangleright_\alpha(\text{let } f = \lambda x.\text{ref } (\lambda x.x))\\ &\text{in } f\ unit := (\lambda x.(x+1)); !(f\ unit)\ true)\end{aligned}$$

この 2 つの例の間の違いを考えると，前者では let 式の属するステージよりも先に計算が起こるステージに参照を確保する項があり，後者では let 式の属するステージと同じステージに参照を確保する項があることがわかる．束縛されている項が関数の場合は，参照を確保する項が let 式のあるステージ以後にある場合はその参照は確保されないため，参照の型に現れる型変数を一般化することができる．

このステージの比較を行うため，let 式の型付けで一般化できる型変数をその let 式のあるステージでラベル付けされたものと参照の型に現れないもののみに制限する．型変数のラベルの直観的な意味は,「その型変数はラベルとなっているステージまでには具体化されている」というものである．参照の型付けでは，参照が確保されるときには参照の中身の型が決まっている必要がある．なので，参照の型付けに現れてもよい型変数は，その参照を確保する項が属するステージよりも先に計算されるステージのもののみに制限する．

また，let 式で束縛されるものが関数でない場合の例を見ると，

$$\begin{aligned}&\text{let } c = \text{let } f = \Lambda\alpha.\blacktriangleright_\alpha(\lambda x.\%_\alpha(\text{ref } (\lambda y.y))) \text{ in}\\ &\qquad ((f\ \varepsilon) \text{ unit}) := (\lambda x.(x+1));\\ &\qquad !((f\ \varepsilon) \text{ unit})\ true\end{aligned}$$

のように let 式と参照を確保する項が同じステージにある場合は参照が確保されてしまう．よって，let 式自体のあるステージでラベル付されたものは一般化することができない．

ふたつめの型システムは，項の先頭の構築子だけを見て値かどうかを判断する素朴な値制限だけでも，CSP を使わない限り健全になのではないか，という観察に基づき，ひとつめの型システムを変更したものである．具体的には，CSP の対象となる項の型にあらわれる型変数のうち参照型に現れるものにのみステージの制限が加わることになる．ふたつめの型システムは受理できる項の観点からはひとつめの型システムより制限が強いが，CSP を使わない限り命令的型変数を使わないため，素朴な値制限を採用した型システムと実質同じであり単純なものとなっている．

3 MiniML$^{\triangleright\%}$ の構文と操作的意味論

この節では，MiniML$^{\triangleright\%}$ の文法と操作的意味論を

定義する．

定義 1 (MiniML$^{\triangleright\%}$ の文法)．MiniML$^{\triangleright\%}$ の変数，遷移変数，遷移，ロケーション，項を以下のように定義する．

$$\begin{aligned}
&\text{変数 } x, y, z \in \Upsilon\\
&\text{遷移変数 } \alpha, \beta \in \Theta\\
&\text{遷移 } A, B \in \Theta^*\\
&\text{ロケーション } l \in \mathcal{L}\\
\text{項 } M, N ::= {}& x\ A_1 \cdots A_n \mid \lambda x.M \mid M\ N \mid \text{unit}\\
&\mid \text{ref } M \mid !M \mid M := N \mid l\\
&\mid \blacktriangleright_\alpha M \mid \blacktriangleleft_\alpha M \mid \%_\alpha M\\
&\mid \text{let } x = \Lambda\alpha_1 \cdots \alpha_n.M \text{ in } N
\end{aligned}$$

遷移は，遷移変数の長さ 0 以上の有限列である．遷移 A と B に対して． $AA' = B$ となるような遷移 A' が存在するとき，$A \leq B$ とかくことにする．ロケーションは，確保された参照のメモリ上での位置を表すもので，ロケーションの可算無限集合 $\mathcal{L}$ の元である．$\lambda^{\triangleright\%}$ では遷移変数に関する抽象を行うための項 $\Lambda\alpha.M$ があったが，型システムの簡略化のため，ここでは let で局所定義される項のみに制限する．ただし，複数の遷移変数での抽象を許している．対応して，遷移 (の列) に適用できるのは変数のみとなっている．以下，(n が重要でないところでは)$\Lambda\alpha_1 \cdots \alpha_n.$ を $\Lambda\overrightarrow{\alpha}.$ と，$x\ A_1 \cdots A_n$ を $x\ \overrightarrow{A}$ と書く．

定義 2 (値, 簡約基)．遷移で添字付けされた項の部分集合 V^A を以下のように定義する．このうち，V^ε の元を値と呼び，メタ変数は (添字のない) v を使う．

$$\begin{aligned}
\text{値 } v^\varepsilon \in V^\varepsilon ::= {}& \lambda x.M \mid \text{unit} \mid \blacktriangleright_\alpha v^\alpha \mid l\\
v^A \in V^A ::= {}& x\ \overrightarrow{B} \mid \lambda x.v^A \mid v^A v^A \mid \text{unit}\\
&\mid \text{ref } v^A \mid !v^A \mid v^A := v^A\\
&\mid \blacktriangleleft_\alpha v^{A'}\ (A'\alpha = A \text{ and } A' \neq \varepsilon)\\
&\mid \blacktriangleright_\alpha v^{A\alpha} \mid \%_\alpha v^{A'} (A'\alpha = A)\\
&\mid \text{let } x = \Lambda\overrightarrow{\alpha}.v^A \text{ in } v^A
\end{aligned}$$

また，簡約基を以下のように定義する．

$$\begin{aligned}
\text{簡約基 } R^\varepsilon ::= {}& (\lambda x.M)\ v^\varepsilon \mid \text{ref } v^\varepsilon \mid !l \mid l := v^\varepsilon\\
&\mid \text{let } x = \Lambda\overrightarrow{\alpha}.v^\varepsilon \text{ in } M\\
R^\alpha ::= {}& \blacktriangleleft_\alpha \blacktriangleright_\alpha M
\end{aligned}$$

おおまかにいうと，$V^{\alpha_1 \cdots \alpha_n}$ の元は，コード値 $\blacktriangleright_{\alpha_1} \cdots \blacktriangleright_{\alpha_n}$ の中に現れてよい項を表している．ここには基本的には全ての種類の項が現れてよいのだが，$\blacktriangleleft$ の出現には条件があるため，このような定義になっている．

次に，操作的意味論で用いる代入を定義する．

定義 3 (代入)．遷移 A 中の遷移変数 β への遷移 B の代入 $A[\beta := B]$ を以下のように定義する．

$$\begin{aligned}
&\varepsilon[\alpha := B] = \varepsilon\\
&(A\alpha)[\alpha := B] = (A[\alpha := B])B\\
&(A\alpha)[\beta := B] = (A[\beta := B])\beta \qquad (\text{if } \alpha \neq \beta)\\
&(\overrightarrow{A}\, A_{n+1})[\beta := B] = (\overrightarrow{A}[\beta := B])(A_{n+1}[\beta := B])
\end{aligned}$$

次に，項 M 中の遷移変数 α への遷移 A の代入を以下のように書く．

$$\begin{aligned}
(x\ \overrightarrow{A})[\alpha := A] &= x\ (\overrightarrow{A}[\alpha := A])\\
(\blacktriangleright_\alpha M)[\alpha := A] &= \blacktriangleright_A (M[\alpha := A])
\end{aligned}$$

ただし， $A = \alpha_1 \cdots \alpha_n$ $(n \geq 0)$ とすると，

$$\begin{aligned}
\blacktriangleright_A M &= \blacktriangleright_{\alpha_1} \cdots \blacktriangleright_{\alpha_n} M\\
\blacktriangleleft_A M &= \blacktriangleleft_{\alpha_n} \cdots \blacktriangleleft_{\alpha_1} M\\
\%_A M &= \%_{\alpha_n} \cdots \%_{\alpha_1} M
\end{aligned}$$

であり，特に $n = 0$ のとき $\blacktriangleright_\varepsilon M = \blacktriangleleft_\varepsilon M = \%_\varepsilon M = M$ となる．最後に，項 M 中の変数 x への $\Lambda\overrightarrow{\alpha}.N$ の代入 $M[x := \Lambda\overrightarrow{\alpha}.N]$ を定義する．

$$\begin{aligned}
(x\overrightarrow{A})[x := \Lambda\overrightarrow{\alpha}.N] &= N[\overrightarrow{\alpha} := \overrightarrow{A}]\\
(M\ N)[x := \Lambda\overrightarrow{\alpha}.N] &= (M[x := \Lambda\overrightarrow{\alpha}.N])\ (N[x := \Lambda\overrightarrow{\alpha}.N])\\
&\vdots
\end{aligned}$$

($[\overrightarrow{\alpha} := \overrightarrow{A}]$ は同時代入に拡張した表記である．)

代入で特に重要なのは，$(x\overrightarrow{A})[x := \Lambda\overrightarrow{\alpha}.N] = N[\overrightarrow{\alpha} := \overrightarrow{A}]$ の場合である．MiniML$^{\triangleright\%}$ の元となっている $\lambda^{\triangleright\%}$ [3] では，遷移変数による抽象 $\Lambda\alpha.M$ や遷移への適用 $M\ A$ が項であり，$(\Lambda\alpha.M)\ A \to M[\alpha := A]$ という簡約が与えられていた．一方，MiniML$^{\triangleright\%}$ では遷移変数に関する抽象が出現する位置を制限しており，$(\Lambda\alpha.N)A$ という項は構文的に許されていない．そのため，α_i を A_i で具体化するプロセスを簡約ではなく代入で一気に行ってしまうようにしたものが上で定義した代入である．

定義 4 (自由な変数，遷移変数)．項 M に自由に出現する変数，遷移変数の集合をそれぞれ $FV(M)$, $FTrV(M)$ と書き，それぞれ以下のように定義する．

(ただし，ここでは重要な場合のみ示している．)

$$FV(x\ \overrightarrow{A}) = \{x\}$$
$$FV(\lambda x.M) = FV(M) \setminus \{x\}$$
$$FV(\text{let } x = \Lambda\overrightarrow{\alpha}.M \text{ in } N) = FV(M) \cup (FV(N) \setminus \{x\})$$
$$FTrV(x\ \overrightarrow{A}) = FTrV(\overrightarrow{A})$$
$$FTrV(\blacktriangleright_\alpha M) = \{\alpha\} \cup FTrV(M)$$
$$FTrV(\text{let } x = \Lambda\overrightarrow{\alpha}.M \text{ in } N) = (FTrV(M) \setminus \{\overrightarrow{\alpha}\}) \cup FTrV(N)$$

項のステージや値呼びであることを考慮した簡約の定義のため，評価文脈を定義する．

定義 5 (評価文脈)．簡約規則を定めるための評価文脈を，図 1 で定義する．評価文脈はふたつの遷移 A, B で添字付けされており．直観的には，遷移 A がその評価文脈自体のあるステージ，遷移 B が $\square$ のステージを表している．また，$E_B^A[M]$ で E_B^A 中の $\square$ を M で置き換えた項を表す，

$$\begin{aligned}
E_B^\varepsilon \in ECtx_B^\varepsilon ::= {} & \square\ (\text{if } B = \varepsilon) \mid E_B^\varepsilon\ M \mid v^\varepsilon\ E_B^\varepsilon \\
& \mid \text{ref } E_B^\varepsilon \mid !E_B^\varepsilon \mid E_B^\varepsilon := M \\
& \mid v^\varepsilon := E_B^\varepsilon \mid \blacktriangleright_\alpha E_B^\alpha \\
& \mid \text{let } x = \Lambda\overrightarrow{\alpha}.E_B^\varepsilon \text{ in } M \\
E_B^A \in ECtx_B^A ::= {} & \square\ (\text{if } A = B) \mid \lambda x.E_B^A \mid E_B^A\ M \\
& \mid v^A\ E_B^A \mid \text{ref } E_B^A \mid !E_B^A \\
& \mid E_B^A := M \mid v^A := E_B^A \\
& \mid \blacktriangleright_\alpha E_B^{A\alpha} \mid \blacktriangleleft_\alpha E_B^{A'}(A'\alpha = A) \\
& \mid \%_\alpha E_B^{A'}(A'\alpha = A) \\
& \mid \text{let } x = \Lambda\overrightarrow{\alpha}.E_B^A \text{ in } M \\
& \mid \text{let } x = \Lambda\overrightarrow{\alpha}.v^A \text{ in } E_B^A
\end{aligned}$$

図 1　評価文脈

定義 6 (ストア)．各参照と，それに保持されている値の対応であるストアを，ロケーションの集合 $\mathcal{L}$ から値の集合 V^ε への有限部分写像とする．

また，簡約規則などでは以下のような略記を用いる．

- $(\mu, l \mapsto v)$ で μ に $l \mapsto v$ を追加したストアをあらわす (ただし $l \notin dom(\mu)$ とする)．
- $[l \mapsto v]\mu$ で μ 中の $l \mapsto v'$ を $l \mapsto v$ に置き換えたストアを表す (ただし $l \in dom(\mu)$ とする)．

定義 7 (簡約規則)．MiniML$^{\triangleright\%}$ の簡約関係は，

$$M \,|\, \mu \ \rightarrow\ M' \,|\, \mu'$$

という形の項とストアの組の二項関係であり，図 2 で与える簡約規則からなる最小の関係である．

Scope extrusion はこの型システムでは関知しないため，E-RefE ，E-AssignE の 2 つの規則で，参照の中に変数が自由に出現するような項が代入された時点で実行時エラーを発生させる．

4　型システム

本節では，前節で定義した言語に対し，型システムを与える．

4.1　定義

まず，型変数と，作用的・命令的型変数を区別するためのカインドを定義する．

定義 8 (型変数，カインド)．

$$\text{型変数 } X, Y, Z \in \Phi$$
$$\text{カインド } K \in \{\mathbf{app}\} \cup \Theta^*$$

型変数が作用的である場合は **app**，命令的である場合は そのステージを表す遷移がそのカインドとなる．

定義 9 (型，型スキーム)．遷移変数の列 $\alpha_1 \cdots \alpha_n$ に対して $\forall\alpha_1.\cdots.\forall\alpha_n$ を $\forall\overrightarrow{\alpha}$ と，型変数とカインドからなる列 $X_1 :: K_1 \cdots X_m :: K_m$ に対して $\forall X_1 :: K_1.\cdots.\forall X_m :: K_m$ を $\forall\overrightarrow{X :: K}$ とかく．

以下のように型 T と型スキーム S を定義する．

$$\begin{aligned}
T ::= {} & b \mid X \mid T \to T \\
& \mid \text{Unit} \mid \text{Ref } T \mid \triangleright_\alpha T \\
S ::= {} & \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T
\end{aligned}$$

ただし，b は基底型である．

型環境は，変数とその型，もしくは，型変数とそのカインドの対応関係の宣言の並びである．

定義 10 (型環境)．型環境 Γ を以下のように定義する．

$$\text{型環境 } \Gamma ::= \emptyset \mid \Gamma, x : S@A \mid \Gamma, X :: K$$

変数は他の多段階計算体系と同様に型だけでなくステージ (遷移) とも関連付けられている．

定義 11 (型・型スキーム・型環境の自由な遷移変数・型変数)．型，型スキーム，型環境に自由に出現する型変数の集合をそれぞれ $FTV(T)$，$FTV(S)$，$FTV(\Gamma)$

$$E_\varepsilon^A[(\lambda x.M)\ v]\,|\,\mu\ \to\ E_\varepsilon^A[M[x := v]]\,|\,\mu\ \text{(E-App)} \qquad \frac{l \notin dom(\mu) \text{ and } FTrV(v) = \emptyset}{E_\varepsilon^A[\text{ref } v]\,|\,\mu\ \to\ E_\varepsilon^A[l]\,|\,(\mu, l \mapsto v)}\ \text{(E-Ref)}$$

$$\frac{FV(v) \neq \emptyset}{E_\varepsilon^A[\text{ref } v]\,|\,\mu \to SEError}\ \text{(E-Ref)} \qquad \frac{FV(v) = \emptyset}{E_\varepsilon^A[l := v]\,|\,\mu\ \to\ E_\varepsilon^A[\text{unit}]\,|\,[l \mapsto v]\mu}\ \text{(E-Assign)}$$

$$\frac{FV(v) \neq \emptyset}{E_\varepsilon^A[l := v]\,|\,\mu \to SEError}\ \text{(E-Assign)} \qquad \frac{l \in dom(\mu)}{E_\varepsilon^A[!l]\,|\,\mu\ \to\ E_\varepsilon^A[\mu(l)]\,|\,\mu}\ \text{(E-Deref)}$$

$$E_\varepsilon^A[\text{let } x = \Lambda\overrightarrow{\alpha}.v \text{ in } N]\,|\,\mu\ \to\ N[x := E_\varepsilon^A[\Lambda\overrightarrow{\alpha}.v]]\,|\,\mu\ \text{(E-Let)}$$

$$E_\alpha^A[\blacktriangleright_\alpha \blacktriangleleft_\alpha M]\,|\,\mu\ \to\ E_\alpha^A[M]\,|\,\mu\ \text{(E-}\blacktriangleright\blacktriangleleft\text{)}$$

図 2　簡約規則

と書き，以下のように定義する．

$$\begin{aligned} FTV(X) &= \{X\} \\ FTV(\forall\overrightarrow{\alpha}.\forall\overrightarrow{X}.T) &= FTV(T) \setminus \overrightarrow{X} \\ FTV(\Gamma) &= \bigcup_{x:S@A\in\Gamma} FTV(S) \end{aligned}$$

また， 型，型スキーム，型環境に自由に出現する遷移変数の集合をそれぞれ $FTrV(T)$，$FTrV(S)$，$FTrV(\Gamma)$ と書き，以下のように定義する．

$$\begin{aligned} FTrV(\triangleright_\alpha T) &= \{\alpha\} \\ FTrV(\forall\overrightarrow{\alpha}.\forall\overrightarrow{X}.T) &= (FTrV(T) \cup FTrV(\overrightarrow{X})) \setminus \{\overrightarrow{\alpha}\} \\ FTrV(\Gamma) &= \bigcup_{x:S@A\in\Gamma} (FTrV(S) \cup FTrV(A)) \end{aligned}$$

また，型 T 中で参照の型の中に現れる型変数の集合 $FITV(T)$ を以下のように定義する．

$$\begin{aligned} FITV(b) &= \emptyset \\ FITV(X) &= \emptyset \\ FITV(T_1 \to T_2) &= FITV(T_1) \cup FITV(T_2) \\ FITV(\text{Unit}) &= \emptyset \\ FITV(\text{ref } T) &= FTV(T) \\ FITV(\triangleright_\alpha T) &= FITV(T) \end{aligned}$$

例 12. $FTV(X \to \text{ref } Y) = \{X, Y\}$ が成立する．また $FITV(X \to \text{ref } Y) = \{Y\}$ が成立する．

定義 13 (型への代入)**.** 型 T 中の型変数 X への T' の代入 $T[X := T']$ を以下のように定義する．

$$\begin{aligned} b[X := T'] &= b \\ X[X := T'] &= T \\ Y[X := T'] &= Y \qquad (X \neq Y) \\ (T_1 \to T_2)[X := T'] &= (T_1[X := T']) \to (T_2[X := T']) \\ \text{Unit}[X := T'] &= \text{Unit} \\ (\text{Ref } T)[X := T'] &= \text{Ref } (T[X := T']) \\ (\triangleright_\alpha T)[X := T'] &= \triangleright_\alpha (T[X := T']) \end{aligned}$$

型に対する遷移の代入 $T[\alpha := A]$ を以下のように定義する．

$$\begin{aligned} b[\alpha := A] &= b \\ X[\alpha := A] &= X \\ (T_1 \to T_2)[\alpha := A] &= (T_1[\alpha := A]) \to (T_2[\alpha := A]) \\ \text{Unit}[\alpha := A] &= \text{Unit} \\ (\text{Ref } T)[\alpha := A] &= \text{Ref } (T[\alpha := A]) \\ (\triangleright_\alpha T)[\alpha := A] &= \triangleright_A (T[\alpha := A]) \\ (\triangleright_\beta T)[\alpha := A] &= \triangleright_\beta (T[\alpha := A]) \qquad (\beta \neq \alpha) \end{aligned}$$

ただし，$\triangleright_A$ は $\blacktriangleright_A$ と同様に定義する．

型に現れる型変数とステージの制約を表現するため，以下のような関係 $\Gamma \vdash T \leq K$ を定義する．

定義 14. $\Gamma \vdash T \leq K \overset{\text{def.}}{\iff} K = \textbf{app}$ または，ある A について $K = A$ かつ $\forall X \in FTV(T).X :: B \in \Gamma$ かつ $B \leq A$.

定義 15 (ストア型付け)**.** ロケーションの型付けを行うため，ストアと同様にロケーションの集合 $\mathcal{L}$ から型の集合 **Types** への有限部分写像であるストア型付

けを導入する．また，ストアと同様に以下のような略記を用いる．

- $l \notin dom(\Sigma)$ のとき $(\Sigma, l \mapsto T)$ で Σ に $l \mapsto T$ を追加したストアをあらわす．
- $l \in dom(\Sigma)$ のとき $[l \mapsto T]\Sigma$ で Σ 中の $l \mapsto T'$ を $l \mapsto T$ に置き換えたストアを表す．

定義 16 (型付け関係)．図 3 の共通の型付け規則と型付け規則 1 のみから $\Gamma; \Sigma \vdash M : T\,@A$ が導出できるとき，

$$\Gamma; \Sigma \vdash_1 M : T\,@A$$

と書く．同様に，共通の型付け規則と型付け規則 2 から $\Gamma; \Sigma \vdash M : T\,@A$ が導出できるとき，

$$\Gamma; \Sigma \vdash_2 M : T\,@A$$

と書く．

型システム 1 で特に重要な規則は， T-Ref, T-LetLam, T-Let である．let 式の型付け規則が 2 種類あり，定義されるのがラムダ抽象である場合の型付け (T-LetLam) では，参照の型では使われない作用的型変数と，let 式のあるステージを表す遷移をカインドとする命令的型変数のみ一般化できる．それ以外の項の場合には，作用的型変数と，let 式のあるステージを表す遷移にさらに束縛されている遷移変数を含む列を加えた遷移をカインドとする型変数のみが一般化できる．参照の型付けでは，参照のあるステージよりも前のステージがその型に現れる型変数として使われていなければならない．これらの制限によって，let 式全体が簡約されるより先に確保される参照の型変数は一般化されないようになる．

型システム 2 は，ほとんど通常の値制限の型システムになっており，ラムダ式やコード以外の項は let に現れても単相的にしか使えない (T-Let)．一方，参照の型付けにあった条件が，CSP 操作の型付け規則に移っている．これにより，CSP によってステージをまたいで使われるロケーションの型に現れる型変数が後のステージで束縛されることを防いでいる．

4.2 型システム 1 の性質

上で定義した型システム 1 が健全であることを subject reduction, progress の性質を証明することで示す．

まず，いくつかの準備的定義を行う．ストアとストア型付けが正しく対応していることを示す以下の関係を導入する．

定義 17 (ストアの正しい型付け)．$dom(\mu) = dom(\Sigma)$ かつ $\forall l \in dom(\mu).\emptyset; \Sigma \vdash_1 \mu(l) : \Sigma(l)\,@\varepsilon$ のときストア型付け Σ のもとでストア μ に型が付くといい，

$$\Sigma \vdash_1 \mu$$

と書く．

また，健全性の条件となる型環境の条件 ε-free を以下のように定義する．

定義 18 (ε-free)．$\forall (x : S@A) \in \Gamma.A \neq \varepsilon$ であるとき，Γ は ε-free であるという，

まず，型システム 1 で型がつく項は，評価文脈と Redex に一意に分解できる，すなわち型が付く項では評価結果が一意に定まる．

定理 19 (型システム 1 の Unique Decomposition)．$\Gamma; \Sigma \vdash_1 M : T\,@A$ かつ Γ が ε-free ならば，$M \in V^A$ であるか，または $M = E_B^A[R^B]$ となるようなペア (E_B^A, R^B) が一意に存在する．ただし， B は ε か，単一の遷移変数からなる列 β である．

証明. $\Gamma; \Sigma \vdash_1 M : T\,@A$ に関する帰納法で示すことができる． □

次に，型システム 1 の健全性として，Progress と Subject reduction を証明する．

Progress は，型が付く項は正しく評価を進めることができるか，あるいは Scope extrusion が起きるかであることを示している．

定理 20 (型システム 1 の Progress)．$\Gamma; \Sigma \vdash_1 M : T\,@A$, $\Sigma \vdash_1 \mu$ かつ Γ が ε-free ならば，$M \,|\, \mu \;\rightarrow\; M' \,|\, \mu'$ となるような項 M' とストア μ' が存在するか，$M \,|\, \mu \rightarrow SEError$ となる．

Progress に関しては， Unique Decomposition を使って簡単に示すことができる．

証明. Unique Decomposition より，$M = E_B^A[R^B]$ となる評価文脈と Redex が一意に存在する．よって，いずれかの簡約規則を用いて評価を進めることができる．特に E-Deref の場合， $R^B = !l$ となるが，

共通の型付け規則

$$\frac{x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T@A \in \Gamma \qquad \Gamma \vdash T_i \leq K_i}{\Gamma;\, \Sigma \vdash x\ \overrightarrow{A} : T[\overrightarrow{\alpha} := \overrightarrow{A}][\overrightarrow{X} := \overrightarrow{T}]\,@A}\ (\text{T-Var}) \qquad \frac{\Gamma, x : T_1@A;\, \Sigma \vdash M : T_2\,@A}{\Gamma;\, \Sigma \vdash \lambda x.M : T_1 \to T_2\,@A}\ (\text{T-Abs})$$

$$\frac{\Gamma;\, \Sigma \vdash M : T_1 \to T_2\,@A \qquad \Gamma;\, \Sigma \vdash N : T_1\,@A}{\Gamma;\, \Sigma \vdash M\ N : T_2\,@A}\ (\text{T-App})$$

$$\frac{\Gamma;\, \Sigma \vdash M : \text{Ref}\ T\,@A \qquad \Gamma;\, \Sigma \vdash N : T\,@A}{\Gamma;\, \Sigma \vdash M := N : \text{Unit}\,@A}\ (\text{T-Assign}) \qquad \frac{\Gamma;\, \Sigma \vdash M : \text{Ref}\ T\,@A}{\Gamma;\, \Sigma \vdash !M : T\,@A}\ (\text{T-Deref})$$

$$\frac{l \in dom(\Sigma)}{\Gamma;\, \Sigma \vdash l : T\,@\varepsilon}\ (\text{T-Loc}) \qquad \frac{\Gamma;\, \Sigma \vdash M : T\,@A\alpha}{\Gamma;\, \Sigma \vdash \blacktriangleright_\alpha M : \triangleright_\alpha T\,@A}\ (\text{T-}\blacktriangleright) \qquad \frac{\Gamma;\, \Sigma \vdash M : \triangleright_\alpha T\,@A}{\Gamma;\, \Sigma \vdash \blacktriangleleft_\alpha M : T\,@A\alpha}\ (\text{T-}\blacktriangleleft)$$

$$\frac{\Gamma, \overrightarrow{X :: K};\, \Sigma \vdash \lambda x.M : T'\,@A \qquad K_i \in \{\mathbf{app}, A\} \qquad \Gamma, x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T'@A;\, \Sigma \vdash N : T\,@A}{\Gamma;\, \Sigma \vdash \text{let}\ x = \Lambda\overrightarrow{\alpha}.\lambda x.M\ \text{in}\ N : T\,@A}\ (\text{T-LetLam})$$

型付け規則 1

$$\frac{\Gamma;\, \Sigma \vdash M : T\,@A \qquad \Gamma \vdash T \leq A}{\Gamma;\, \Sigma \vdash \text{ref}\ M : \text{Ref}\ T\,@A}\ (\text{T-Ref}) \qquad \frac{\Gamma;\, \Sigma \vdash M : T\,@A}{\Gamma;\, \Sigma \vdash \%_\alpha M : T\,@A\alpha}\ (\text{T-}\%)$$

$$\frac{\begin{array}{c}\Gamma, \overrightarrow{X :: K};\, \Sigma \vdash \lambda x.M : T'\,@A \qquad K_i \in \{\mathbf{app}\} \cup \{AA'\alpha \mid A' \in \Theta^*, \alpha \in \overrightarrow{\alpha}\} \\ \Gamma, x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T'@A;\, \Sigma \vdash N : T\,@A\end{array}}{\Gamma;\, \Sigma \vdash \text{let}\ x = \Lambda\overrightarrow{\alpha}.M\ \text{in}\ N : T\,@A}\ (\text{T-Let})$$

型付け規則 2

$$\frac{\Gamma;\, \Sigma \vdash M : T\,@A}{\Gamma;\, \Sigma \vdash \text{ref}\ M : \text{Ref}\ T\,@A}\ (\text{T-Ref}) \qquad \frac{\Gamma;\, \Sigma \vdash M : T\,@A \qquad \forall X \in FITV(T).\Gamma \vdash X \leq A}{\Gamma;\, \Sigma \vdash \%_\alpha M : T\,@A\alpha}\ (\text{T-}\%)$$

$$\frac{\begin{array}{c}\Gamma, \overrightarrow{X :: K};\, \Sigma \vdash \blacktriangleright_\alpha M : T'\,@A \qquad K_i \in \{\mathbf{app}, A\alpha\} \qquad \alpha \in \overrightarrow{\alpha} \\ \Gamma, x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T@A;\, \Sigma \vdash N : T\,@A\end{array}}{\Gamma;\, \Sigma \vdash \text{let}\ x = \Lambda\overrightarrow{\alpha}.\blacktriangleright_\alpha M\ \text{in}\ N : T\,@A}\ (\text{T-Let}\blacktriangleright)$$

$$\frac{\Gamma;\, \Sigma \vdash M : T'\,@A \qquad \Gamma, x : \forall\overrightarrow{\alpha}.T'@A;\, \Sigma \vdash N : T\,@A}{\Gamma;\, \Sigma \vdash \text{let}\ x = \Lambda\overrightarrow{\alpha}.M\ \text{in}\ N : T\,@A}\ (\text{T-Let})$$

図 3　型付け規則

$\Sigma \vdash_1 \mu$ より $l \in dom(\mu)$ である．　□

Subject reduction では，正しく型付けされた項とストアから正しく評価が進んだとき，評価した結果の項とストアもまた正しく型がつくことを示している．

定理 21 (型システム 1 の Subject Reduction)**.** $\Gamma;\, \Sigma \vdash_1 M : T\,@A$, $\Sigma \vdash_1 \mu$, $M \mid \mu \to M' \mid \mu'$ がなりたち，T，Γ には ステージ ε の命令的な型変数が自由に出現しないならば，$\Gamma;\, \Sigma' \vdash_1 M' : T\,@A$ かつ $\Sigma' \vdash_1 \mu'$ が成り立つようなストアの型付け $\Sigma' \supseteq \Sigma$ が存在する．

Subject reduction の証明のために，いくつかの補題を証明する．

まず，代入に関して型付けが保存されることを示す代入補題を示す．そのために，型スキーム，カインド，型環境への型・遷移の代入を定義する．

定義 22. 型スキームに対する型の代入 $(\forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.T)[X := T']$ を以下のように定義する．$(\forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.T)[X := T'] = \forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.(T[X := T'])$ ただし $X \notin \overrightarrow{Y}$ かつ $FTV(T') \cap \overrightarrow{Y} = \emptyset$ かつ $FTrV(T') \cap \overrightarrow{\alpha} = \emptyset$. 次に，型環境に対する型の代入 $\Gamma[X := T']$ を以下のように定義する．

$$\begin{aligned}&(\Gamma, x : S@A)[X := T'] \\ &\qquad = (\Gamma[X := T']), x : (S[X := T'])@A \\ &(\Gamma, Y :: K)[X := T'] \\ &\qquad = (\Gamma[X := T']), Y :: K \qquad (X \neq Y)\end{aligned}$$

型スキームに対する遷移の代入 $(\forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.T)[\beta := B]$ を以下のように定義する．$(\forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.T)[\beta := B] = \forall\overrightarrow{\alpha}.\forall\overrightarrow{Y :: K}.(T[\beta := B])$ (ただし $\beta \notin \overrightarrow{\alpha}$ かつ $FTV(B) \cap \overrightarrow{\alpha} = \emptyset$). 最後に，型環境に対する遷

移の代入 $\Gamma[\beta := B]$ を以下のように定義する.

$(\Gamma, x : S@A)[\beta := B]$
$= (\Gamma[\beta := B]), x : (S[\beta := B])@(A[\beta := B])$
$(\Gamma, X :: K)[\beta := B]$
$= (\Gamma[\beta := B]), X :: (K[\beta := B])$

ただし，カインドに対する遷移の代入 $K[\beta := B]$ は $A[\beta := B]$ の定義を $\mathbf{app}[\beta := B] = \mathbf{app}$ で拡張すればよい.

補題 23 (型システム 1 の代入補題)**.**

1. $\Gamma, x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}.T@A; \Sigma \vdash_1 M : T\,@A$ かつ $\Gamma; \Sigma \vdash_1 N : T'\,@B$ かつ $\forall\alpha \in \overrightarrow{\alpha}.\alpha \notin FTV(A) \cup FTV(\Gamma)$ ならば
   $\Gamma; \Sigma \vdash_1 M[x := \Lambda\overrightarrow{\alpha}.N] : T\,@A.$
2. $\Gamma; \Sigma \vdash_1 M : T\,@A$ ならば
   $\Gamma[X := T']; \Sigma[X := T'] \vdash_1 M : T[X := T']\,@A$
3. $\Gamma; \Sigma \vdash_1 M : T\,@A$ ならば
   $\Gamma[\alpha := A]; \Sigma[\alpha := A] \vdash M[\alpha := A]$
   $: T[\alpha := A]\,@A[\alpha := A]$

証明.

1. $\Gamma, x : \forall\overrightarrow{\alpha}.\forall\overrightarrow{X :: K}; \Sigma \vdash_1 M : T\,@A$ に関する帰納法で示せる.
2. $\Gamma; \Sigma \vdash_1 M : T\,@A$ に関する帰納法で示せる.
3. $\Gamma; \Sigma \vdash_1 M : T\,@A$ に関する帰納法で示せる.

□

この代入補題を用いて，さらに 2 つの補題を証明する.

補題 24 (Redex の型の保存)**.**

1. Redex $\mathrm{ref}\, v$ について，$\mathrm{ref}\, v \,|\, \mu \to N \,|\, \mu'$, $\Gamma; \Sigma \vdash_1 \mathrm{ref}\, v : T\,@B$, $\Sigma \vdash_1 \mu$ であり，$\forall X \in FTV(T).X :: \varepsilon \notin \Gamma$ ならば，$\Gamma; \Sigma' \vdash_1 N : T\,@B$, $\Sigma' \vdash_1 \mu'$ が成り立つようなストア型付け $\Sigma' \supseteq \Sigma$ が存在する.
2. Redex $R^B (\neq \mathrm{ref}\, v)$ (B は ε または単一の遷移変数からなる列 β) について，$R^B \,|\, \mu \to N \,|\, \mu'$, $\Gamma; \Sigma \vdash_1 R^B : T\,@B$, $\Sigma \vdash_1 \mu$ ならば，$\Gamma; \Sigma \vdash_1 N : T\,@B$, $\Sigma \vdash_1 \mu'$ が成り立つ.

証明. 代入補題より証明できる. □

補題 25 (Redex の型付け中の型変数)**.** $\Gamma; \Sigma \vdash_1 M : T\,@A$ かつ $M = E_B^A[\mathrm{ref}\, v]$ となるような評価文脈 E_B^A と Redex $R = \mathrm{ref}\, v$ が存在し，$R^B = \mathrm{ref}\, v$ かつ $\forall X \in FTV(T) \cup FTV(\Gamma).X :: \varepsilon \notin \Gamma$ ならば，導出中の Redex R^B の型付け $\Gamma; \Sigma \vdash_1 R : T'\,@B$ で $FTV(T') = \emptyset$ が成り立つような $\Gamma; \Sigma \vdash_1 M : T\,@A$ の導出が存在する.

証明. 評価文脈 E_B^A に関する帰納法で示すことができる. □

以上の補題より， Subject Reduction を示すことができる.

型システム *1* の *Subject Reduction* の証明. 補題 25 より，Redex の型に型変数が現れないような $\Gamma; \Sigma \vdash_1 M : T\,@A$ の導出が存在する．補題 24 より，型に型変数が自由に現れないような Redex の簡約については型が保存される．よって，項全体の簡約として見ても $\Gamma; \Sigma' \vdash_1 M : T\,@A, \Sigma' \vdash_1 \mu'$ が成り立つようなストア型付け $\Sigma' \supseteq \Sigma$ が存在する. □

4.3 型システム 2 の性質

型システム 2 も，型システム 1 と同じように $\Sigma \vdash_2 \mu$ を定義することで，Unique Decomposition と Progress が成り立つ.

定理 26 (型システム 2 の Unique Decomposition)**.** $\Gamma; \Sigma \vdash_2 M : T\,@A$ かつ Γ が ε-free ならば，$M \in V^A$ であるか，または $M = E_B^A[R^B]$ となるようなペア (E_B^A, R^B) が一意に存在する．ただし， B は ε か，単一の遷移変数からなる列 β である.

証明. 型システム 1 の Unique Decomposition と同様に，$\Gamma; \Sigma \vdash_2 M : T\,@A$ に関する帰納法で証明することができる. □

定理 27 (型システム 2 の Progress)**.** $\Gamma; \Sigma \vdash_2 M : T\,@A$, $\Sigma \vdash_2 \mu$ かつ Γ が ε-free ならば，$M \,|\, \mu \to M' \,|\, \mu'$ となるような項 M' とストア μ' が存在するか，$M \,|\, \mu \to SEError$ となる.

証明. 型システム 1 の Progress と同様に，Unique

Decomposition によって簡単に証明できる. □

また, Subject Reduction も成り立つと予想される.

予想 28 (型システム 2 の Subject Reduction). $\Gamma; \Sigma \vdash_2 M : T\,@A$ かつ $\Sigma \vdash_2 \mu$ かつ $M\,|\,\mu \to M'\,|\,\mu'$ がなりたち, T, Γ には ステージ ε の命令的な型変数が自由に出現しないならば, $\Gamma; \Sigma' \vdash_2 M' : T\,@A$ かつ $\Sigma' \vdash_2 \mu'$ が成り立つようなストアの型付け $\Sigma' \supseteq \Sigma$ が存在する.

4.4 比較

型システム 1 と型システム 2 を比較したとき, 型システム 2 で型がつくプログラムは型システム 1 でもまた型が付くと予想される.

予想 29. $\Gamma; \Sigma \vdash_2 M : T\,@A \Rightarrow \Gamma; \Sigma \vdash_1 M : T\,@A$

型システムとしては型システム 1 は 型システム 2 よりも強い. 関数適用などであっても, 参照の操作を含まないようなものならば多相型をつけることができる.

しかし, 現在広く使われている値制限により近いのは型システム 2 である. とくに CSP を使わない場合には単純にコードや関数を値とみなして型をつけることができ, 型変数のステージを気にせずにプログラムを書くことができる.

5 関連研究

本稿で提案する型システムは既に述べたように Tofte による命令的型変数を使った参照のための多相型システム [11] に基づいている. Tofte の手法だけでも, MiniML$^{\triangleright\%}$ に対する健全な型システムは構築できるが, コードの内と外を区別しないので, ref を含むコードを純粋な計算で操作する時に多相性を失うという欠点がある. 例えば,

$$\text{let } c = \Lambda\alpha.\blacktriangleright_\alpha \text{let } f = \lambda z.\blacktriangleleft_\alpha((\lambda x.x)\ (\blacktriangleright_\alpha(\text{ref}\ (\lambda y.y)))) \text{ in } \ldots$$

のようなプログラムを考える. これは, 単にエスケープ部分で恒等関数をコード片に適用しているだけで, 結果として

$$\text{let } c = \Lambda\alpha.\blacktriangleright_\alpha \text{let } f = \lambda z.\text{ref}\ (\lambda y.y) \text{ in } \ldots$$

となるので, f は多相的に使えてもよいのだが, Tofte の型システムをそのまま適用すると, ref $(\lambda y.y)$ の型には命令的型変数が現れるため一般化することができない. 一方, 我々の型システム 1 では引用中の ref の出現と引用の外のそれとをカインドで区別することができるので, 上の f を多相的に使うことができる.

このように命令的型変数をさらに詳細化するアイデアは, Standard ML of New Jersey (SML/NJ) で採用されていた weak type variable [4] とも共通するものといえる. SML/NJ では, 関数抽象の中に現れるために関数適用が発生するまでは参照を生成しない ref の出現と, そうでない出現を区別し, 適用が何度起こると参照が確保されるかの情報を表すために命令的型変数を自然数で添字付けしたような型システムを採用していた. 一方, 我々の型システムはどの遷移変数に ε が代入されると参照が確保されるかの情報を表すために命令的型変数に遷移を加えていると考えることができる.

今回の多段階プログラミングのモデルとなる計算体系は $\lambda^{\triangleright\%}$ [3] をベースとしたが, より MetaOCaml に近い型システムである λ^{α} [9] や λ^{i} [1] をベースとしても, 今回と同様なアイデアが適用できると考えているが, これらの体系では, 明示的に遷移の受渡しが行われなかったり, コードの評価の意味が $\lambda^{\triangleright\%}$ と微妙に異なるため, その点は考慮する必要があるだろう.

その他にも, closure typing [7], エフェクトシステム [10] などを使い参照と多相性を型安全に両立させるための研究が数多くされてきた. これらを多段階プログラミングのための型システムと組み合わせてみることは今後の研究として興味深い. 特にエフェクトシステムは scope extrusion 問題への解決策としても研究されているため, 値多相の問題も解決できるのであれば, ふたつの問題を同時に解決できる可能性がある.

6 おわりに

本稿では, 参照と let 式を備えた多段階プログラミング言語 MiniML$^{\triangleright\%}$ を定義し, それに対する型システムをふたつ提案した. また, ふたつめの型システムに関しては一部予想ではあるが, これらの型システムの健全性や関係についての定理および予想を示した.

今後の課題としては, 実用に近づくためにはこの

型システムに対応する型推論アルゴリズムを考えることがあげられる．もうひとつの問題である Scope extrusion については本稿の型システムでは静的に検出することができないため，この問題を解決することも課題の一つである．

参 考 文 献

[1] Calcagno, C., Moggi, E., and Taha, W.: ML-Like Inference for Classifiers, *Proceedings of European Symposium on Programming (ESOP'04)*, Springer LNCS, Vol. 2986, 2004, pp. 79–93.

[2] Garrigue, J.: Relaxing the Value Restriction, *Proc. of International Symposium on Functional and Logic Programming (FLOPS 2004)*, Spritger LNCS, Vol. 2998, 2004.

[3] Hanada, Y. and Igarashi, A.: On Cross-Stage Persistence in Multi-Stage Programming, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, 2014, pp. 103–118.

[4] Hoang, M., Mitchell, J., and Viswanathan, R.: Standard ML-NJ weak polymorphism and imperative constructs, *Proc. of IEEE Symposium on Logic in Computer Science (LICS'93)*, 1993, pp. 15–25.

[5] Kiselyov, O.: The Design and Implementation of BER MetaOCaml: System Description, *Proc. of International Symposium on Functional and Logic Programming (FLOPS 2014)*, Springer LNCS, Vol. 8475, 2014, pp. 86–102.

[6] Kiselyov, O. and Shan, C.-c.: Staged let-generalization may be unsound, `http://okmij.org/ftp/meta-programming/calculi.html#staged-poly`, 2010.

[7] Leroy, X. and Weis, P.: Polymorphic Type Inference and Assignment, *Proc. of ACM Symposium on Principles of Programming Languages (POPL'91)*, 1991, pp. 291–302.

[8] Taha, W.: A Gentle Introduction to Multi-stage Programming, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, Lengauer, C., Batory, D. S., Consel, C., and Odersky, M.(eds.), Lecture Notes in Computer Science, Vol. 3016, Springer, 2003, pp. 30–50.

[9] Taha, W. and Nielsen, M. F.: Environment Classifiers, *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, 2003, pp. 26–37.

[10] Talpin, J.-P. and Jouvelot, P.: The Type and Effect Discipline, *Information and Comptation*, Vol. 111, No. 2(1994), pp. 245–296.

[11] Tofte, M.: Type Inference for Polymorphic References, *Information and computation*, Vol. 89, No. 1(1990), pp. 1–34.

[12] Wright, A. K.: Simple Imperative Polymorphism, *Lisp and Symbolic Computation*, Vol. 8(1995).