情報処理学会論文誌 Vol.54 No.8 1996–2011 (Aug. 2013)

限定継続機構とfutureを持つ計算体系の透過的意味論

田中 麻峰[1,†1,a)] 亀山 幸義[1,b)]

受付日 2012年12月25日, 採録日 2013年5月18日

概要：継続機構は，種々の制御構造からモジュラで簡潔なウェブアプリケーションの記述まで，多岐にわたる応用を有する．並列計算による高性能化への要求が高まっている今日，継続機構を用いたプログラムを並列化するのは自然な要求である一方，逐次計算で考案された継続機構の並列化は困難な課題である．Moreau は，`call/cc` と `future` を持つ計算体系を提案し，並列意味論が逐次意味論と一致することを証明したが，彼の体系では `call/cc` を実行するとプログラム全体が逐次化されるという問題があった．本論文では，制御が及ぶ範囲をプログラムの一部に限定する継続機構を提供する `shift/reset` と `future` を持つ体系を提案し，先行研究の問題を解決した．また，提案する体系の意味論を逐次抽象機械と並列抽象機械によって与え，両者の意味が一致することを証明した．

キーワード：並列計算，コントロール・オペレータ，プログラム意味論，合流性

Transparent Semantics of a Calculus with Delimited Control and Future

Asami Tanaka[1,†1,a)] Yukiyoshi Kameyama[1,b)]

Received: December 25, 2012, Accepted: May 18, 2013

Abstract: Control operators for continuations have many applications such as implementation of various control structures, and modular description of web applications. It is a natural, but hard problem to parallelize the execution of programs with control operators due to the sequential nature of continuations. Moreau proposed a calculus with `call/cc` and `future`, and proved that parallel execution yields the same result as sequential execution does for his calculus. However, his solution is not fully satisfactory, since `call/cc` captures an unlimited continuation, and once it is called, the whole program must be executed sequentially. This paper presents a calculus with `future` and delimited-control operators `shift/reset`, which give an access to delimited continuation, part of the rest of computation. We define the semantics of the calculus in two ways by sequential and parallel abstract machines, and prove that these two semantics coincide.

Keywords: parallel computation, control operator, program semantics, confluence

1. はじめに

プログラムにおける多様な制御を抽象化し，モジュラリティを上げる手段の１つとして，コントロール・オペレータの利用があげられる．コントロール・オペレータは，主に関数型プログラム言語で研究されており，特に，プログラムの実行時の概念である「継続」（continuation）に対して，捕捉（キャプチャ）・利用・合成・破棄などの操作を行えるようにしたコントロール・オペレータを本論文では継続機構と呼ぶ．継続は，プログラムの実行時における「残りの計算」を意味する概念であり [1]，継続機構を用いることによって，複雑な制御を簡潔かつ理解しやすい形で記述できるようになることが知られている．継続を操作するコントロール・オペレータの代表は，Scheme や SML/NJ の `call/cc` である．

継続機構は「残りの計算のすべて」を操作するが，継続として扱う範囲を限定する方がアルゴリズムを記述しやすい場合があることから，「残りの計算の一部」を操作する限

[1] 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[†1] 現在，ニッセイ情報テクノロジー株式会社
Presently with Nissay Information Technology Co., Ltd.
[a)] asami@logic.cs.tsukuba.ac.jp
[b)] kameyama@acm.org

© 2013 Information Processing Society of Japan

定継続機構，特に，shift/reset [2] の研究がさかんである．プログラム変換 [3] やウェブアプリケーションへの応用 [4] など，制御機構の抽象化によるモジュラリティの高いプログラム例が多く知られている．

継続や限定継続の機構を用いて記述された精緻なプログラムを並列化することは，重要かつ困難な課題である．ハードウェアが進化し，マルチコアやメニーコアを用いた並列計算が主流となる中，既存のアルゴリズムを並列に実行する要求が高まっている．一方，継続や限定継続は，本質的に逐次計算に対して定義される概念であるため，並列計算とともに用いることは容易ではない．たとえば，素朴にコントロール・オペレータを追加した並列計算の体系は非決定的となってしまい，逐次計算の場合と同じ意味を持つことは保証されない．既存のアルゴリズムを並列に実行するためには，並列化が意味を変えないことを保証する必要がある．

Katz ら [5] や Moreau [6], [7] は，call/cc と，Lisp 系言語における明示的並列性を表すプリミティブである future を持つ体系を定義し，future が透過的であること，すなわち，future の有無によって意味が変わらないことを示した．しかし，彼らの体系では，透過性を維持するために並列性が大きく損なわれるケースが多い，という問題があった．コントロール・オペレータが実行されると並列計算を止めないとならないが，call/cc は残りの計算全体を継続として扱うため，この停止の影響がプログラム全体に及ぶことがその原因である．

我々は，このような状況を打破するため，限定継続機構を持つ並列計算体系を提案する．提案する体系は，明示的な並列計算を示す future と，限定継続機構を扱う shift/reset を持つ．この体系に対して，抽象機械に基づいた厳密な意味論を与え，その意味論のもとで future が透過的であること，すなわち，逐次機械と並列機械による意味が等しいことを証明する．call/cc ではなく，残りの計算の一部を扱う限定継続機構 shift/reset を採用する利点は 2 つあり，後者の方がより精密な計算エフェクトを表現できること，そして，後者は制御機構の有効範囲を限定できるため，並列計算を抑える範囲も限定でき，並列化の効果が得られやすいことである．我々は，この体系に対して，透過性を厳密に証明した．我々の証明法では，並列計算が合流性を持つことを，Takahashi の並行簡約 [8] の手法を用いて明らかにした．これにより，Moreau の先行研究における，煩雑なステップ数の議論がいっさい不要である，という点で，見通しの良い証明を構築することができた．

本論文で提案する体系には，破壊的変数の機能も持たせている．これは，本論文がモデル化の対象としているプログラム言語が Scheme であることの自然な帰結であるとともに，shift/reset と future というコントロール・エフェクトのみを含む計算体系ではなく，破壊的変数などの計算エフェクトも持つ，より実践的な言語でも，本研究のアプローチが成立することを確認するためである．

次に，本研究と既存研究との関連を述べる．

(1) shift/reset を持つ項は，CPS (Continuation Passing Style) 変換を用いれば，純粋なラムダ計算の項へ変換できる [9]．よって，この方式で shift/reset を消去し，その後で future を用いた並列化を行うことが考えられる．しかし，CPS 変換は，プログラムを逐次化するものであり，もとのプログラムに内在していた並列実行の可能性を消去してしまうので，並列効果は得られない．

(2) shift/reset は，call/cc と破壊的変数 (値を更新できる変数) の組合せでマクロ定義することが可能である [10]．Moreau の提案する言語 [7] は，call/cc と破壊的変数および future を持つため，この手法により shift/reset と future を併せ持つ体系が得られるように見える．しかし，この手法で実現した shift/reset は，実行時に call/cc を呼ぶため，call/cc による計算の逐次化という問題が残ってしまう．たとえば，本論文の 2 章で述べる A 変換の並列化を Moreau の体系への帰着によって実現しようとしても，並列効果はほとんど得られない．

(3) 近年の研究で，ラムダ計算に future と破壊的変数を導入した体系 [11], [12] が提案されている．しかし，これらが提案する体系の future は，設計の思想が本研究と異なっており，透過性は成立しない．つまり，future の有無によってプログラムの実行結果は異なることがある．

本論文は，次のように構成される：まず，2 章では，shift/reset と future の直感的な説明と具体例を与える．3 章は，本研究で提案する言語とその抽象機械による意味論を定義する．4 章は，その抽象機械に基づいて，future が透過的であることを明らかにする．5 章で，本研究のまとめと今後の課題を述べる．

2. 具体例

この章では，まず shift/reset と future について，例を用いて説明する．その後，両者を組み合わせた例題を示し，透過性の意義を明らかにする．

2.1 コントロール・オペレータ shift/reset

shift オペレータは，それを実行した時点から，最も近い reset までの残りの計算（限定継続）を切り取って，変数に束縛する．限定継続は，文脈と呼ばれる □（穴）を 1 つだけ持つ式で表現できる．以下では，式を Scheme と同じように S 式として表記するが，誤解がない場合は括弧を省略することがある．$\langle e \rangle$ は reset で囲まれている式を，(shift $k.e$) は限定継続を切り取って変数 k に束縛する式をそれぞれ表す．また，$\rightsquigarrow$ は 1 ステップの簡約を，$\rightsquigarrow^*$ は 0 ステップ以上の簡約を表す．

© 2013 Information Processing Society of Japan

$$
\begin{aligned}
& (+\ 1\ \langle +\ 10\ (\texttt{shift}\ k.100)\rangle) \\
\rightsquigarrow\ & (+\ 1\ \langle 100\rangle) \rightsquigarrow^{*} 101 && (1)\\
& (+\ 1\ \langle +\ 10\ (\texttt{shift}\ k.(k\ 100))\rangle) \\
\rightsquigarrow\ & (+\ 1\ \langle k\ 100\rangle) \rightsquigarrow (+\ 1\ \langle\langle +\ 10\ 100\rangle\rangle) \rightsquigarrow^{*} 111 && (2)\\
& (+\ 1\ \langle +\ 10\ (\texttt{shift}\ k.(k\ (k\ 100)))\rangle) \\
\rightsquigarrow\ & (+\ 1\ \langle k\ (k\ 100)\rangle) \\
\rightsquigarrow\ & (+\ 1\ \langle k\ \langle +\ 10\ 100\rangle\rangle) \rightsquigarrow^{*} 121 && (3)
\end{aligned}
$$

(1) では，shift を含む式が実行される際，$\langle +\ 10\ \square\rangle$ という文脈が限定継続として切り取られ，変数 k に束縛される．このとき，元々あった reset は残る点に注意されたい．この場合は，k は使われないため，限定継続も使われない．(2) でも，同じ文脈が k に束縛される．それを $(k\ 100)$ の形で適用すると，評価文脈 $\langle +\ 10\ \square\rangle$ の穴に 100 が埋め込まれ，$\langle +\ 10\ 100\rangle$ という式となる．(3) では限定継続を $(k\ (k\ 100))$ の形で 2 回適用して，$\langle +\ 10\ \langle +\ 10\ 100\rangle\rangle$ という式になり，計算の結果 121 となる．

2.2 並列計算のためのオペレータ future

future [13] は，その引数の計算を，他の計算と並列に実行するオペレータである．ここでは模式的に，並列に動作する 2 つの計算を $e_1 \mid e_2$ と表現する．

$$
\begin{aligned}
& (\texttt{let}\ ((x\ (\texttt{future}\ (+\ 1\ 10))))\ (+\ 100\ 1000\ x)) && (4)\\
\rightsquigarrow & (\texttt{let}\ ((x\ p))\ (+\ 100\ 1000\ x)) \mid p = (+\ 1\ 10) && (5)\\
\rightsquigarrow & (+\ 1100\ p) \mid p = 11 && (6)\\
\rightsquigarrow & (+\ 1100\ 11) \rightsquigarrow 1111 && (7)
\end{aligned}
$$

(4) で future を含む式が実行されると，future の引数となる式 $(+\ 1\ 10)$ は，future 式の周りの文脈 $(\texttt{let}\ ((x\ \square))\ (+\ 100\ 1000\ x))$ と並列に実行される．この際，特殊な変数 p が生成され，(5) のとおり，$(+\ 1\ 10)$ の計算結果を他方のプロセスに渡す役割を果たす．この p を含む式の計算においては，p は値として振る舞い，(6) のとおり計算が進む．$(+\ 1100\ p)$ という式の計算は，p に値が入るまで進まず中断する．一方，$p = 11$ は，右側のプロセスの計算が終了したことを意味しており，p を介して値 11 が左側にわたり，右側のプロセスは解消されて (7) を得る．

2.3 A 変換

本節では，プログラム変換の 1 つで，A 正規形と呼ばれる形への変換である A 変換の例を述べる．A 正規形は，すべての関数適用において，引数が値になるよう分解したものであり，計算の中間結果には let 式で名前が付けられている．たとえば，(f (g x)) の A 正規形は，(let ((m1 (g x))) (let ((m2 (f m1))) m2)) である．

2.3.1 shift/reset による A 変換の実装

Asai [3] は，A 変換と本質的に同じ変換を shift/reset を用いて簡潔に定義できることを示した．本項と次項では Scheme の構文でプログラムを記述する．また，Scheme の擬似引用（quasi-quote）の機構を用いる．

```
1  (define (a exp)
2    (cond
3      ;; x
4      [(symbol? exp) exp]
5      ;; (lambda (x) e)
6      [(eq? (car exp) 'lambda)
7       (let ((var (car (car (cdr exp))))
8             (body (car (cdr (cdr exp)))))
9         `(lambda (,var)
10           ,(reset (a body))))]
11     ;; (e1 e2)
12     [else
13      (shift k
14        (let ((m (gensym 'm))
15              (n (gensym 'n)))
16          `(let ((,m ,(a (car e1)))
17                 (,n ,(a (car
18                           (cdr e2))))
19             ,(k `(,m ,n)))))]))
```

関数 a は，（A 変換されていない）ラムダ式（コード）exp を受け取り，それを A 変換した式（コード）を返す[*1]．exp が変数の場合，そのまま返す．exp が関数抽象の場合，関数抽象の本体部分を変換する．このとき，本体部分の変換の外側に reset を置いているが，その意味は後述する．exp が関数適用 (e1 e2) の場合，まず，shift を無視して定義を読むと，(1) 新しい変数を 2 つ生成し（それを m1, n1 とする），(2) e1 と e2 をそれぞれ A 変換したものを a1, a2 として，(3) (let ((m1 a1) (n1 a2)) (m1 n1)) を返す，という手順である．しかし，この手順でたとえば，(f (g x)) を変換すると，(let ((m1 f) (n1 (let ((m2 g) (n2 x)) (m2 n2)))) (m1 n1)) という式が得られるが，let が入れ子になっており A 正規形ではない．これを解消するため，上記の shift が必要であり，それを例で説明する．

式 (f (g x)) を A 変換する．ここで，f, g, x は変数とする．変換は reset に囲まれた形 (reset (a '(f (g x)))) から始まり，まず以下のようになる．

```
(reset
  (shift k `(let ((m1 ,(a 'f))
                  (n1 ,(a '(g x))))
     ,(k `(m1 n1)))))
```

次に shift が実行されるが，切り取られる文脈は空であるので，以下のようになる[*2]．

```
(reset `(let ((m1 ,(a 'f))
              (n1 ,(a '(g x))))
          (m1 n1)))
```

[*1] 実際の Scheme の挙動とは異なるが，この項では let の束縛は上から順番に評価されると仮定する．

[*2] 変数 k に束縛されているのは空の継続であり，計算の結果に影響を与えない．そのため，ここでは k を先に適用した．

© 2013 Information Processing Society of Japan

次に，(a 'f) と (a '(g x)) がそれぞれ実行される．

```
(reset
  `(let ((m1 f)
         (n1 ,(shift k1
                `(let ((m2 g) (n2 x))
                   ,(k1 `(m2 n2))))))
    (m1 n1)))
```

reset の後の let 以降は生成された式であるが，その式の内部にあるカンマのついた式は実行される．そのため，次に shift が `(let ((m1 f) (n1 [])) (m1 n1)) という文脈を切り取り，それを'(m2 n2) に適用する．

```
(reset `(let ((m2 g) (n2 x))
          (let ((m1 f) (n1 (m2 n2)))
            (m1 n1))))
```

この式から reset（および quote）を除去したものが最終的な結果となる[*3]．

2.3.2 future による A 変換の並列化

前項の A 変換を future により並列化する際，まず考えられるのは，関数適用 (e1 e2) の処理において，e1 と e2 の処理を並列化することである．しかし，e1 がさらに関数適用である場合は，その処理ですぐに shift が呼ばれ，本論文の体系では，計算を逐次化してしまう．その代わりに，関数抽象の処理を並列化する．これは，関数抽象の処理が reset で区切られているため，その中で shift が呼ばれても，外部に影響を及ぼさず，並列化が阻害されないためである．そのために，変換を以下のように変形する．

```
1  (define (a exp)
2    (cond
3      ; 変数の場合は同じ
4      [...]
5      ; 関数抽象の場合だけ変更
6      [(eq? (car exp) 'lambda)
7       (let ((var (car (car (cdr exp))))
8             (body (car (cdr (cdr exp)))))
9         `(lambda (,var)
10           ,(future (reset (a body)))))]
11     ; 関数適用の場合も同じ
12     [...]
13   ))
```

前の定義と異なるのは，関数抽象の場合の最後（プログラム中の 10 行目）で，再帰的に関数 a を呼ぶ際に，future によって囲んでいる点である．この並列化によって，reset で区切られる計算はすべて並列に実行される．特に，関数適用において，関数部分と引数部分がそれぞれ関数抽象であれば，それらの変換は並列に行われる．

一方で，Moreau の提案する体系では，継続機構の影響を一部にとどめられないため，上記のような並列化はできない．このことを確認するため，まず，Filinski の手法 [10] により，call/cc と破壊的変数を用いて shift/reset を表現したオペレータ（それらを SFT，RST と呼ぶ）を定義し，それらを用いて上述の A 変換を実装する．この SFT と RST は，reset の外側の継続を管理するための破壊的変数が必要になる[*4]．

たとえば，((lambda (x) e1) (lambda (y) e2)) という式を A 変換することを考える．SFT と RST を用いた A 変換関数を future によって並列化したとすると，その変換の途中結果は，模式的に以下のように表せる．

```
(RST `(let ((m1 (lambda (x)
                  ,(future
                     (RST (a e1))))))
        (let ((m2 (lambda (y)
                    ,(future
                       (RST (a e2))))))
          (m1 m2))))
```

このとき，(a e1) の計算と同時に (a e2) も計算が行われようとするが，それぞれの RST は共通する破壊的変数へアクセスしようとするため，後者の計算は前者の計算が完了して初めて実行される[*5]．そのため，このやり方では並列化ができていないことが分かる．

2.4 木の探索

並列化されたプログラムで shift/reset を活用する例として，木の探索を考える．対象とするデータは二分木とし，探索関数を以下のように定義する．

```
1  (define (search tree pred proc)
2    (cond
3      [(null? tree) '()]
4      [(pred tree) (proc tree)]
5      [else (begin
6              (future (search (cadr tree)
7                              pred proc))
8              (search (caddr tree)
9                      pred proc))]))
```

search 関数は，二分木 tree，判定関数 pred，探索結果を処理する関数 proc を引数にとる．対象としているデータは，'(data left right) のような，データ，左部分木，右部分木の 3 つからなるリストである．また，両部分木も再帰的に同じ構造をしているものとする．プログラム中の cadr と caddr は，それぞれ，リストの 2 番目，3 番目の要素を取り出すプリミティブ関数である．4 行目で，pred を満たす場合の proc による処理を行う．6 行目の再帰的に左部分木の探索を行う部分で，future による並列化をしている．

探索の解を発見した場合の処理（proc）として，一番単

[*3] ここでは，変数を式と見なして名前をつけたため，本来の A 正規形とは若干異なるが，本質的には同じである．

[*4] ここでは，Scheme での実装 [14] を想定している．

[*5] 共通する破壊的変数へ並列にアクセスする場合の具体的な動作例を，3.3 節に記載しているので，参照されたい．

© 2013 Information Processing Society of Japan

純なものは解を画面に表示するものであるが，次のようなprocを与えると，2つ目以降の解を探索するかどうかを後で決められる．

```
(define (proc tr)
  (shift k (cons tr k)))
```

関数procは，発見したデータtrとその時点の限定継続kをペアにして，直近のresetに戻す．この限定継続を後で呼び出すことで，探索を再開することができる．

ここで，上のproc関数がshiftを用いて切り取った限定継続の中には，並列に動作している他の部分木の探索を実行しているプロセスが含まれる．つまり，並列に動作しているプロセスをshiftが切り取る，という動作が生じる．素朴に考えると，これは，他のすべてのプロセスを強制的に停止させることになり，並列性が著しく失われるという問題がある．本研究では，この問題に一定の解決を与えることができた．その解決法は3章の抽象機械の定義で述べ，解決法の発想については4章の最後の節で述べる．

3. 抽象機械に基づく操作的意味論

この章では，提案する体系λ_{sr}^{ft}の構文とその意味論を，抽象機械AM-PFSRとAM-FSRに基づく操作的意味論として定義する．前者はfutureによって並列実行を行う抽象機械であり，後者はfutureを無視して逐次的に実行する抽象機械である．

3.1 構文の定義

λ_{sr}^{ft}の構文を**図1**に示す．ここで，shiftは記号$\mathcal{S}$で表し，resetに囲まれた項は$\langle M\rangle$と表す．項Mは，変数，関数抽象，関数適用のほか，2章で説明したshift/resetとfuture，定数，プリミティブ関数から構成される．また，破壊的変数（値を書き換え可能な変数）の生成make，値の書き換えset!と参照derefの操作を持つ．

項の定義で，$(\lambda x.M)$はM中の自由な変数xを束縛し，$(\mathcal{S}k.M)$はM中の自由な変数kを束縛する．自由変数，束縛変数は通常どおり定義し，α同値な項は同一視する．

3.2 AM-PFSRの状態

抽象機械AM-PFSRの状態，文脈，ストアの定義を**図2**に示す．なお，図1と同じ定義となるものは記載しない．

状態Sは，ストアθと項Mのペアである．項Mの定義で，fletというコンストラクタが増えている．これは，futureを評価した際に現れる実行時の項である．後述する遷移規則では，flet項は必ずresetで囲われることが分かるので，構文の段階でresetで囲われた形のみに限定している．また，図1では関数などをすべてMに含んでいたが，ここでは値はVとして区別し，MはこのVを含んでいるものとする．Vには，pやbが加わっている．pは，futureを評価した際に生成される特殊な変数であり，コミュニケーション変数と呼ぶ．bは，破壊的変数を表す．以降では，通常の変数，コミュニケーション変数，破壊的変数をまとめて変数と呼ぶ．Aはアンサで，評価の最終結果となる．

$$
\begin{aligned}
M &::= x \mid a \mid f \mid (\lambda x.M) \mid (M_1\ M_2) \\
&\quad \mid \langle M\rangle \mid (\mathcal{S}k.M) \mid (\texttt{future}\ M) \\
a &::= 0 \mid 1 \mid 2 \cdots \\
f &::= \texttt{make} \mid \texttt{set!} \mid \texttt{deref} \\
x &::= x \mid y \mid z \mid k \mid v \cdots
\end{aligned}
$$

図1 λ_{sr}^{ft}の構文
Fig. 1 Syntax of λ_{sr}^{ft}.

状態

$$
\begin{aligned}
S &::= \{\!|\theta,\ M|\!\} \\
M &::= V \mid (M_1\ M_2) \mid \langle M\rangle \mid (\mathcal{S}k.M) \\
&\quad \mid (\texttt{future}\ M) \mid \langle \texttt{flet}\ (p\ M)\ S\rangle \\
V &::= c \mid x \mid b \mid p \mid (\lambda x.M) \mid (\texttt{set!}\ V) \qquad c ::= a \mid f \mid \textbf{void} \\
b &::= b \mid b_1 \mid b_2 \cdots \qquad p ::= p \mid p_1 \mid p_2 \cdots \\
A &::= c \mid \textbf{box} \mid \textbf{procedure} \mid \textbf{error}
\end{aligned}
$$

文脈

$$
\begin{aligned}
C &::= \square \mid (\lambda x.C) \mid (C\ M) \mid (M\ C) \mid \langle C\rangle \mid (\mathcal{S}k.C) \\
&\quad \mid (\texttt{future}\ C) \mid \langle \texttt{flet}\ (p\ C)\ S\rangle \mid \langle \texttt{flet}\ (p\ M)\ \{\!|\theta,\ C|\!\}\rangle \\
E &::= G \mid E[F] \\
F &::= \langle G\rangle \mid \langle \texttt{flet}\ (p\ G)\ S\rangle \\
G &::= \square \mid (G\ M) \mid (V\ G)
\end{aligned}
$$

ストア

$$
\theta ::= \{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\} \quad (b_i \text{ は互いに相異なる})
$$

図2 AM-PFSRの状態と文脈およびストアの定義
Fig. 2 State, context and store of AM-PFSR.

文脈には4種類ある．Cは任意の文脈を表す．$C[M]$は，文脈Cの持つ穴$\square$に項Mを埋めて得られる項を表す．この際，Mの自由変数が束縛されることがある．EとFはcall-by-value（値呼び）における評価文脈であり，項の中の次に簡約の対象となる部分項（これをレデックスと呼ぶ）を決定する役割を持つ．Eは一般の（特に制限のない）評価文脈であり，Fは，一番外側にresetを持ち，それ以外には$\square$の周りにresetを持たない評価文脈である．Eを一般評価文脈，Fを純粋評価文脈と呼ぶ．Gは補助的に使う文脈である．

ストアθは破壊的変数とその値からなる環境である．ストアの定義域を，$Dom(\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}) =$

© 2013 Information Processing Society of Japan

$$\langle\!\langle \theta,\ E[((\lambda x.M)\ V)]\rangle\!\rangle \to \langle\!\langle \theta,\ E[M\{x := V\}]\rangle\!\rangle \qquad (app)$$

$$\langle\!\langle \theta,\ E[\langle V\rangle]\rangle\!\rangle \to \langle\!\langle \theta,\ E[V]\rangle\!\rangle \qquad (rv)$$

$$\langle\!\langle \theta,\ E[F[(\mathcal{S}k.M)]]\rangle\!\rangle \to \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle]\rangle\!\rangle \quad (v \text{ fresh}) \qquad (rs)$$

$$\langle\!\langle \theta,\ E[(\texttt{make}\ V)]\rangle\!\rangle \to \langle\!\langle \theta \cup \{b \mapsto V\},\ E[b]\rangle\!\rangle \quad (b \text{ fresh}) \qquad (make)$$

$$\langle\!\langle \theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]\rangle\!\rangle \to \langle\!\langle \theta \cup \{b \mapsto V\},\ E[\textbf{void}]\rangle\!\rangle \qquad (set)$$

$$\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]\rangle\!\rangle \to \langle\!\langle \theta \cup \{b \mapsto V\},\ E[V]\rangle\!\rangle \qquad (deref)$$

図 3 逐次実行に関する遷移規則
Fig. 3 Transition rules for sequential execution.

$$\langle\!\langle \theta,\ E[F[(\texttt{future}\ M)]]\rangle\!\rangle \to \langle\!\langle \theta,\ E[\langle \texttt{flet}\ (p\ M)\ \langle\!\langle \emptyset,\ F[p]\rangle\!\rangle\rangle]\rangle\!\rangle \quad (p \text{ fresh}) \qquad (fork)$$

$$\langle\!\langle \theta,\ E[\langle \texttt{flet}\ (p\ V)\ \langle\!\langle \theta_1,\ M_1\rangle\!\rangle\rangle]\rangle\!\rangle \to \langle\!\langle \theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]\rangle\!\rangle \qquad (join)$$

$(Dom(\theta) \cap Dom(\theta_1) = \emptyset$ のとき$)$

$$\langle\!\langle \theta,\ C[\langle \texttt{flet}\ (p\ M)\ S\rangle]\rangle\!\rangle \to \langle\!\langle \theta,\ C[\langle \texttt{flet}\ (p\ M)\ S'\rangle]\rangle\!\rangle \quad (S \to S' \text{ のとき}) \qquad (spec)$$

$$\langle\!\langle \theta \cup \{b \mapsto V\},\ M\rangle\!\rangle \to \langle\!\langle \theta \cup \{b \mapsto V'\},\ M\rangle\!\rangle \quad (\langle\!\langle \emptyset,\ V\rangle\!\rangle \to \langle\!\langle \emptyset,\ V'\rangle\!\rangle \text{ のとき}) \qquad (store)$$

図 4 並列実行に関する遷移規則
Fig. 4 Transition rules for parallel execution.

$\{b_1, \cdots, b_n\}$ と定義する．

項 $\langle \texttt{flet}\ (p\ M)\ S\rangle$ において，S 中に自由に現れるコミュニケーション変数 p は束縛される．また，状態 $\langle\!\langle \theta,\ M\rangle\!\rangle$ において，$b \in Dom(\theta)$ であるとき，M 中に自由に現れる破壊的変数 b は束縛される．また，$\langle\!\langle \theta,\ M\rangle\!\rangle$ の中の，束縛されていない破壊的変数は自由であるという．2 つの項・文脈・評価文脈・ストア・状態が，束縛変数の名前の違いを除いて一致するとき，その両者を同一視する[*6]．項が $\langle M\rangle$ の形か値 V の形で，かつ，破壊的変数以外には自由な変数を持たない項をプログラムと呼び，メタ変数 P で表す．$\langle\!\langle \theta,\ P\rangle\!\rangle$ の形の状態を，特に，プログラム状態と呼ぶ．

3.3 AM-PFSR の遷移規則

AM-PFSR の遷移規則を，**図 3** と**図 4** の 2 つに分けて示す．図 3 には，逐次的な遷移の規則を定義する．いくつかの遷移規則は，項が `reset` で囲まれていることを前提としている．状態遷移は，その状態の持つ項がプログラムであることを想定しているためである．

(app) は関数適用の規則である．ここで，$M\{x := V\}$ は自由変数の捕捉を避ける，通常の代入である．

(rv) は `reset` の内側の項が値になった際に `reset` を除去する規則である．(rs) は，`shift` による限定継続の切り取りを行う規則である．その時点での最も近い `reset` までの評価文脈を切り取る．切り取られた評価文脈 F は，関数 $(\lambda v.F[v])$ に変換され，変数 k に束縛される．

破壊的変数は，$(make)$ により生成される．$(make)$ によって生成される変数名 b は，その時点で一意なものが生成される．破壊的変数は，(set) によって値の書き換えを，$(deref)$ によって値の読み出しをそれぞれ行う．

続いて，並列実行のための規則を図 4 に示す．$(fork)$ は，`future` を並列化オペレータとして処理する規則である．この規則が適用されると，`flet` というコンストラクタをともなう項（これを `flet` 項と呼ぶ）を持つ状態へと遷移する．`flet` 項は「同時に計算を進められる複数の項を持つ」ことを表しており，後述する $(spec)$ によって計算を並列に進められるようになる．$(make)$ と同様，p も全体で一意であるとし，また $\emptyset$ は空のストアを表す．

$(join)$ は，直感的には，`flet` 項が持つ値 V と状態 $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle$（が持つストアと項）を 1 つの状態にまとめる規則である．遷移後の状態は，次のようになる：ストアは，θ と，θ_1 中の p に V を代入したストア $\theta_1\{p := V\}$ の和集合であり，項は，M_1 中の p に V を代入した項 $M_1\{p := V\}$ を，評価文脈 E の中に与えたものになる．$(join)$ で遷移する際，2 つのストアの定義域が互いに重ならないことが条件となっているが，ストアで束縛された破壊的変数の名前は名前変えされるため，一般性は失われない．

$(spec)$ は，項 $\langle \texttt{flet}\ (p\ M)\ S\rangle$ の持つ状態 S を遷移させる規則である．ここで重要なことは，任意の文脈 C の下でこの S が遷移できることであり，たとえば，関数抽象の中に含まれる `flet` 項であっても適用できる．これが，前章の最後の例で述べた，捕捉された他のプロセスの計算を妨げない仕組みである．なお，関数抽象の本体部分に `flet`

[*6] コミュニケーション変数や破壊的変数は，フレッシュな名前が動的に生成されるが，その名前の違いは意味がない．

© 2013 Information Processing Society of Japan

項が含まれるのは，flet 項を含む文脈を shift が捕捉する場合のみである．

(*store*) は，ストアに含まれる値 V に対する遷移を表す．ここで，ストアに記録されている項は必ず値であり，値に対して適用しうる遷移規則は (*spec*) のみである．そのため，(*store*) の条件節における $\langle\!\langle \emptyset,\ V \rangle\!\rangle \to \langle\!\langle \emptyset,\ V' \rangle\!\rangle$ という遷移は，必ず (*spec*) 規則によるものであることに注意されたい．

以降では，(*spec*) と (*store*) による遷移を投機的遷移（speculative transition）と呼び，それ以外の遷移規則による遷移を必須遷移（mandatory transition）と呼ぶ．投機的遷移は，必ずしも実行が必要ではない遷移である．

状態 $S = \langle\!\langle \theta,\ M \rangle\!\rangle$ について，M が値でなく，かつ上記のいずれの遷移規則でも遷移できないとき，状態 S は stuck しているという．特に，適用できる必須遷移規則が 1 つもないとき，必須遷移について stuck しているという．

ここで，AM-PFSR の状態遷移の例を示す．以下では，(let ((x e_1)) e_2) は，((λx.e_2) e_1) を表す．項 M を

```
(let ((x (make 0)))
  (let ((y (future ((set! x) 10))))
    (let ((z ((set! x) 20))) (deref x))))
```

とし，$S_0 = \langle\!\langle \emptyset,\ \langle M \rangle \rangle\!\rangle$ とする．この例は，2 つのプロセスが，同じ破壊的変数を書き換えようとする状況を表している．その遷移は以下のようになる．

```
S0 →⟦{b1 ↦ 0},
      ⟨(let ((x b1))
         (let ((y (future ((set! x) 10))))
           (let ((z ((set! x) 20)))
             (deref x)))))⟩⟧                          (make)
→⟦{b1 ↦ 0},
      ⟨(let ((y (future ((set! b1) 10))))
         (let ((z ((set! b1) 20)))
           (deref b1))))⟩⟧                            (app)
→⟦{b1 ↦ 0},
      ⟨flet (p ((set! b1) 10))
         ⟦∅, ⟨(let ((y p))
                (let ((z ((set! b1) 20)))
                  (deref b1))))⟩⟩⟧                    (fork)
→⟦{b1 ↦ 0},
      ⟨flet (p ((set! b1) 10))
         ⟦∅, ⟨(let ((z ((set! b1) 20)))
                (deref b1)))⟩⟩⟧                      (spec)
```

上の最後の遷移は，flet 項の中の状態の遷移であるため，適用される規則は (*spec*) である点に注意されたい．上記の最後の状態で遷移の対象となる項は，((set! b_1) 10) と ((set! b_1) 20) である．しかし，項 ((set! b_1) 20) を持つプロセスは，それが持つストアに b_1 がないため，他のプロセスの進行を待つ．

```
→⟦{b1 ↦ 10},
      ⟨flet (p void)
         ⟦∅, ⟨(let ((z ((set! b1) 20)))
                (deref b1)))⟩⟩⟧                      (set)
→⟦{b1 ↦ 10},
      ⟨(let ((z ((set! b1) 20)))
         (deref b1)))⟩⟧                               (join)
→⟦{b1 ↦ 20},
      ⟨(let ((z void))(deref b1)))⟩⟧                  (set)
→⟦{b1 ↦ 20}, ⟨(deref b1)⟩⟧                           (app)
→⟦{b1 ↦ 20}, ⟨20⟩⟧                                   (deref)
→⟦{b1 ↦ 20}, 20⟧                                     (rv)
```

図 3 や図 4 に定義される遷移規則は，future が透過的であるように設計されている．実際にこの例で示したとおり，2 つの set!項の実行される順番を制限することで，future を取り除いた式の実行結果と等しい結果が得られている．

この節の最後に，AM-PFSR の項に関する性質を明らかにする．

補題 1 *AM-PFSR* の任意の項 M について，以下のいずれか 1 つだけが成立する．(*1*) $M = V$，(*2*) $M = E[(V_1\ V_2)]$ (ただし，V_1 は set!ではない)，(*3*) $M = E[\langle V \rangle]$，(*4*) $M = E[(\mathcal{S}k.M_1)]$，(*5*) $M = E[(\texttt{future}\ M_1)]$，(*6*) $M = E[\langle \texttt{flet}\ (p\ V)\ S \rangle]$．また，(*2*) から (*6*) のケースでは，分解の仕方（特に E の取り方）は一意的である． □

証明．項 M の構造に関する帰納法で示す．ここでは，$M = (M_1\ M_2)$ の場合のみを示す．帰納法の仮定より，M_1 は上記の (*1*) から (*6*) のいずれかの形である．もし (*2*) から (*6*) なら M 自身も同じケースに該当する．そこで，M_1 が (*1*) であると仮定する．M_2 も，帰納法の仮定から，(*1*) から (*6*) のいずれかであり，(*1*) 以外なら，M 自身が M_2 と同じケースに該当する．

M_1, M_2 がともに (*1*) のとき，M_1 の形に応じて場合分けをする．M_1 が，(λx.M_3)，a，x，b，p，**void**，make，deref，(set! V_3) のとき，M は (*2*) である．M_1 が，set!のとき，M は (*1*) である．

また，M に対して該当するケースが一意的であることと，(*2*) から (*6*) の各ケースで E が一意的であることは，

© 2013 Information Processing Society of Japan

$$
\begin{array}{l}
Unload : \mathbb{V} \to \mathbb{A} \\
\quad Unload(c) = c \qquad Unload((\lambda x.M)) = \textbf{procedure} \\
\quad Unload(b) = \textbf{box} \qquad Unload((\texttt{set!}\ V)) = \textbf{procedure} \\
\quad Unload(p) = \textbf{error} \qquad Unload(x) = \textbf{error}
\end{array}
$$

$$
\begin{array}{l}
eval_p : \mathbb{P} \to \mathrm{Power}(\mathbb{A} \cup \{\bot\}) \\
\quad eval_p(P) = \{Unload(V) \mid [\![\emptyset,\ P]\!] \to^* [\![\theta,\ V]\!]\} \\
\qquad \cup \begin{cases} \{\textbf{error}\} & [\![\emptyset,\ P]\!] \text{ から始まる，必須遷移について} \\ & \texttt{stuck} \text{ する列が存在するとき} \\ \{\} & \text{そのような列が存在しないとき} \end{cases} \\
\qquad \cup \begin{cases} \{\bot\} & \text{無限列 } [\![\emptyset,\ P]\!] \to \cdots \text{ で無限個の} \\ & \text{必須遷移を含むものが存在するとき} \\ \{\} & \text{上記の無限列が存在しないとき} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
eval_s : \mathbb{P} \to \mathrm{Power}(\mathbb{A} \cup \{\bot\}) \\
\quad eval_s(P) = \begin{cases} \{Unload(V) \mid [\![\emptyset,\ P]\!] \to^* [\![\theta,\ V]\!]\} & \\ \qquad \text{無限列でないとき} & \\ \{\textbf{error}\} & [\![\emptyset,\ P]\!] \text{ からの遷移が } \texttt{stuck} \text{ するとき} \\ \{\bot\} & [\![\emptyset,\ P]\!] \text{ からの遷移列が無限列であるとき} \end{cases}
\end{array}
$$

図 5 評価関数
Fig. 5 Evaluation functions.

E の構造に関する場合分けで示すことができる． ■

補題 1 は，値でない項は，評価文脈とレデックス候補に一意的に分解できることを意味している．ただし，(2) から (6) までの形の項に対して遷移が起きるとは限らない．たとえば，項 $(p\ V)$ は (2) の形に該当するが，p に値が代入されるまで遷移は起きない．項が一意的に分解できることから，任意の状態に適用されうる必須遷移規則は，たかだか 1 つであることが分かる．

3.4 AM-PFSR の評価関数

AM-PFSR の状態遷移の結果を返す評価関数 $eval_p$ と，その補助関数 *Unload* を**図 5** に定義する．図 5 と以降の説明では，$\mathbb{P}$，$\mathbb{V}$，$\mathbb{A}$ が，プログラムの集合，値の集合，アンサの集合をそれぞれ表すものとする．

関数 *Unload* は，値 V をアンサ A に変換する．この際，関数や破壊的変数などは具体的な値を消去し定数に変換する．

関数 $eval_p$ は，プログラムを引数とし，$\mathbb{A} \cup \{\bot\}$ の部分集合を返す関数である[*7]．ここで，$\to^*$ は，$\to$ の反射的推移的閉包を表す．この関数は，初期状態 $[\![\emptyset,\ P]\!]$ から，それ以上遷移できなくなるまで状態遷移を繰り返し，遷移が止まった際の値 V を *Unload* に与えて，その結果の集合を返す．この遷移列が無限となり，かつ，その中に無限個の必須遷移が含まれているとき，$eval_p$ が返す集合に $\bot$ が含まれる．$\bot$ は停止しない計算を表す．

ここで，必須遷移が無限個含まれている遷移列に限定しているのは，AM-PFSR では，値になった後も投機的遷移が続くことがあるからである．投機的遷移は，計算結果を得るのに無関係な遷移を含むことがあり，そのような遷移を無限回繰り返す場合は，実質的な無限計算列とは見なせない（たとえば，$\Omega = ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ とすると，$[\![\theta,\ (\lambda x.\langle \texttt{flet}\ (p\ 0)\ [\![\emptyset,\ \Omega]\!]\rangle)]\!]$ から無限回の投機的遷移が可能である）．

[*7] $\mathrm{Power}(X)$ で X のべき集合を表す．

3.5 AM-FSR の状態，遷移規則，評価関数

逐次計算を行う抽象機械 AM-FSR を定義する．AM-FSR は，`future` を無視して逐次的に計算する点が AM-PFSR と異なる．AM-FSR の項，状態，値，文脈は，AM-PFSR の対応するものから，コミュニケーション変数と `flet` コンストラクタを除いたものである．また，評価文脈 E，F，G はそれぞれ以下のように定義される．

$$
\begin{array}{l}
E ::= G \mid E[F] \\
F ::= \langle G \rangle \\
G ::= \square \mid (G\ M) \mid (V\ G) \mid (\texttt{future}\ G)
\end{array}
$$

AM-PFSR の評価文脈との違いは，G の定義で $(\texttt{future}\ G)$ が追加された点である．AM-FSR でも E を一般評価文脈，F を純粋評価文脈と呼ぶ．

AM-FSR の遷移規則は，図 3 の AM-PFSR の遷移規則に以下のものを追加する．

$$[\![\theta,\ E[(\texttt{future}\ V)]]\!] \to [\![\theta,\ E[V]]\!] \qquad (\mathit{fid})$$

また，図 4 の遷移規則は含まない．

2 つの抽象機械の遷移の違いは，$E[(\texttt{future}\ M)]$ という項を計算する際に顕在化する．AM-PFSR では，(*fork*) によりコミュニケーション変数やプロセスが生成され，並列計算が開始される．一方，AM-FSR は，評価文脈が増えたことにより，この形のまま M の計算を始める．M の計算結果が値になると，(*fid*) 規則により `future` が除去される．つまり，`future` は存在してもしなくても計算結果には影響を及ぼさない．

AM-FSR に対しても，補題 1 と同様の性質が成立する．

補題 2 *AM-FSR* の任意の項 M について，以下のいずれか 1 つだけが成立する．(1) $M = V$，(2) $M = E[(V_1\ V_2)]$（ただし，V_1 は `set!`ではない），(3) $M = E[\langle V \rangle]$，(4) $M = E[(\mathcal{S}k.M_1)]$，(5) $M = E[(\texttt{future}\ V)]$．また，(2) から (5) のケースでは，分解の仕方（特に E のとり方）は

© 2013 Information Processing Society of Japan

一意的である． □

証明．補題 1 と同様に，項 M の構造に関する帰納法で示される． ■

また，補題 2 と AM-FSR の遷移規則から，以下の定理が導かれる．

定理 1 *AM-FSR* の状態 S_0 に対して，$S_0 \to S_1$ となる S_1 はたかだか *1* つである．

AM-FSR の評価関数 $eval_s$ は，図 5 で定義する．定理 1 より明らかに，$eval_s$ は要素が 1 つである集合を返す．

4. future の透過性

この章では，本論文の主定理である透過性を証明する．

定理 2（透過性） 任意のプログラム P に対して，$eval_p(P) = eval_s(P)$ である． □

この定理は，並列機械による意味論が決定的であり，逐次機械による意味論と一致することを意味する．すなわち，`future` の有無によってプログラムの意味が変わらないこと（透過性）を意味している．

4.1 逐次実行と並列実行の関係

本論文では，逐次機械の遷移は，そのままでは並列機械の遷移とはならないため，逐次機械の意味が並列機械の意味に含まれることも証明が必要である．

定理 3 任意のプログラム P に対して，$eval_s(P) \subseteq eval_p(P)$ である． □

証明．ここでは概要のみを述べ，詳細な定義と証明は付録 A.1 で与える．

AM-FSR の項 M に対して，M 中の $F[(\texttt{future}\ M)]$ という形の部分項（ただし F は `future` を含まない）のいくつかを，$\langle \texttt{flet}\ (p\ M)\ \{\!|\emptyset,\ F[p]|\!\}\rangle$ に置き換えて N が得られるとき，$M \sim N$ と定義する．ただし，1 回の置換えにより新たに置換え可能な部分項が発生することがあり，$M \sim N$ は，繰り返し置換えを行った場合も含むように定義する．この $\sim$ に対して，AM-FSR の遷移が AM-PFSR の必須遷移を含む遷移列により模倣（シミュレート）されることが証明できる（付録の定理 6）．

これにより，AM-FSR で $\{\!|\emptyset,\ P|\!\}$ から $\{\!|\theta^s,\ V^s|\!\}$ に至る遷移列があれば，AM-PFSR でも $\{\!|\theta^p,\ V^p|\!\}$ に至る遷移列があり，$V^s \sim V^p$ が成立することから $Unload(V^s) = Unload(V^p)$ がいえる．よって，$eval_s(P) \subseteq eval_p(P)$ である．また，AM-FSR で $\{\!|\emptyset,\ P|\!\}$ の計算が `stuck` するときや停止しないときは，AM-PFSR でも同様になることが示せる．以上から，$eval_s(P) \subseteq eval_p(P)$ である． ■

4.2 並列機械の遷移の合流性

合流性は，透過性の証明の鍵となる性質であり，以下のように定式化できる．

定理 4（合流性） 並列機械 *AM-PFSR* の遷移 $\to$ と状態 S_1，S_2，S_3 に対して，$S_1 \to^* S_2$ かつ $S_1 \to^* S_3$ ならば，ある状態 S_4 に対して，$S_2 \to^* S_4$ かつ $S_3 \to^* S_4$ となる． □

本論文では，合流性を証明するために，Takahashi の並行簡約の手法を拡張して使う．この手法は，単一ステップの遷移 $\to$ を拡張した並列遷移 $\Rightarrow$ と，極大並列遷移 $(-)^*$ を定義し，$\Rightarrow$ がいわゆるダイアモンド性を満たすことを証明することで，$\to$ の合流性を示す．$S_1 \Rightarrow S_2$ は，直感的には，状態 S_1 で簡約可能な部分項のいくつか（0 個以上）を同時に簡約して S_2 が得られることを意味する．たとえば，$M_1 = ((\lambda x.x)\ 3)$，$M_2 = ((\lambda x.x)\ 5)$ とするとき，以下の S_1，S_2，S_3 の間に $S_1 \Rightarrow S_2 \Rightarrow S_3$ が成立する[*8]．

$$S_1 = \{\!|\emptyset, \langle \texttt{flet}\ (p\ M_1) \quad \{\!|\emptyset, \langle \texttt{flet}\ (q\ M_2) \{\!|\emptyset, \langle (+\ p\ q)\rangle|\!\}\rangle|\!\}\rangle|\!\}$$

$$S_2 = \{\!|\emptyset,\ \langle \texttt{flet}\ (p\ 3)\ \{\!|\emptyset,\ \langle \texttt{flet}\ (q\ 5)\ \{\!|\emptyset,\ \langle (+\ p\ q)\rangle|\!\}\rangle|\!\}\rangle|\!\}$$

$$S_3 = \{\!|\emptyset,\ (+\ 3\ 5)|\!\}$$

最後のステップで，入れ子になった `flet` 項が 2 つ同時に簡約されている．ただし，本論文での並列遷移は．Takahashi の定義とは異なり，`flet` 項の外にある部分は．$E[R]$ の形で指定される部分項 R（レデックス[*9]）のみを計算する．たとえば．$\{\!|\emptyset,\ (+\ M_1\ M_2)|\!\} \Rightarrow \{\!|\emptyset,\ (+\ 3\ 5)|\!\}$ は成立しない．

4.2.1 並列遷移の定義

並列遷移 $\Rightarrow$ と，その補助となる遷移 $\Rightarrow_r$ と $\Rightarrow_s$ を，**図 6** で定義する．$\Rightarrow$ は状態，$\Rightarrow_r$ は特定の形をした項，$\Rightarrow_s$ は項・評価文脈・ストアに対する二項関係である．直感的には，$R \Rightarrow_r M$ は，R がレデックスでその簡約の結果として M が得られること，また，$S_1 \Rightarrow S_2$ は，S_1 の計算可能な部分項のいくつかを簡約して S_2 が得られること，$S_1 \Rightarrow_s S_2$ は，S_1 において，必須遷移に対応する部分項以外の計算可能な部分項[*10]をいくつか簡約して S_2 が得られることをそれぞれ意味する．

スペースの節約のため，$\Rightarrow_s$ の定義は，項と評価文脈に対してオーバロードして定義している．$\Rightarrow_s$ の左辺が □ を含まないとき，右辺も □ を含まず，項に対する二項関係となり，左辺が評価文脈のとき，右辺も評価文脈の形となり，評価文脈に対する二項関係となる．上記の定義は，項でも

[*8] 加算項 `(+ e1 e2)` は λ_{sr}^{ft} で定義されていないが，ここでは便宜上用いている．
[*9] 以降では，レデックスを表すメタ変数として R を用いる．
[*10] そのような部分項は，$\langle \texttt{flet}\ (p\ M)\ S\rangle$ の形の項の S の中に出現する．

© 2013 Information Processing Society of Japan

並列遷移 $\Rightarrow$ の定義（状態に対して）：

$$\frac{R \Rightarrow_r R' \quad E \Rightarrow_s E' \quad \theta \Rightarrow_s \theta'}{\langle\!\langle \theta,\ E[R] \rangle\!\rangle \Rightarrow \langle\!\langle \theta',\ E'[R'] \rangle\!\rangle}\ (\mathit{Redex}) \qquad \frac{V \Rightarrow_s V' \quad E \Rightarrow_s E' \quad \theta \Rightarrow_s \theta' \quad (b \text{ fresh})}{\langle\!\langle \theta,\ E[(\mathtt{make}\ V)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta' \cup \{b \mapsto V'\},\ E'[b] \rangle\!\rangle}\ (\mathit{Make})$$

$$\frac{V \Rightarrow_s V' \quad E \Rightarrow_s E' \quad \theta \Rightarrow_s \theta'}{\langle\!\langle \theta \cup \{b \mapsto V_1\},\ E[((\mathtt{set!}\ b)\ V)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta' \cup \{b \mapsto V'\},\ E'[\mathbf{void}] \rangle\!\rangle}\ (\mathit{Set}) \qquad \frac{V \Rightarrow_s V' \quad E \Rightarrow_s E' \quad \theta \Rightarrow_s \theta'}{\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\mathtt{deref}\ b)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta' \cup \{b \mapsto V'\},\ E'[V'] \rangle\!\rangle}\ (\mathit{Deref})$$

$$\frac{\theta \Rightarrow_s \theta' \quad E \Rightarrow_s E' \quad V \Rightarrow_s V' \quad \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2,\ M_2 \rangle\!\rangle \quad Dom(\theta') \cap Dom(\theta_2) = \emptyset}{\langle\!\langle \theta,\ E[\langle \mathtt{flet}\ (p\ V)\ \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \rangle] \rangle\!\rangle \Rightarrow \langle\!\langle \theta' \cup (\theta_2\{p := V'\}),\ E'[M_2\{p := V'\}] \rangle\!\rangle}\ (\mathit{Join}) \quad \frac{M \Rightarrow_s M' \quad \theta \Rightarrow_s \theta'}{\langle\!\langle \theta,\ M \rangle\!\rangle \Rightarrow \langle\!\langle \theta',\ M' \rangle\!\rangle}\ (\mathit{Spec})$$

補助的遷移 $\Rightarrow_r$ の定義（以下の左辺の形の項に対して）：

$$\frac{M \Rightarrow_s M' \quad V \Rightarrow_s V'}{((\lambda x.M)\ V) \Rightarrow_r M'\{x := V'\}}\ (\mathit{App}) \qquad \frac{M \Rightarrow_s M' \quad F \Rightarrow_s F' \quad (v \text{ fresh})}{F[(\mathcal{S}k.M)] \Rightarrow_r \langle M'\{k := (\lambda v.F'[v])\} \rangle}\ (\mathit{RtSt}) \qquad \frac{V \Rightarrow_s V'}{\langle V \rangle \Rightarrow_r V'}\ (\mathit{RtVl})$$

$$\frac{M \Rightarrow_s M' \quad F \Rightarrow_s F' \quad (p \text{ fresh})}{F[(\mathtt{future}\ M)] \Rightarrow_r \langle \mathtt{flet}\ (p\ M')\ \langle\!\langle \emptyset,\ F'[p] \rangle\!\rangle \rangle}\ (\mathit{Fork})$$

補助的遷移 $\Rightarrow_s$ の定義（項・評価文脈に対して）：

$$\frac{v = x, c, b, p, \Box}{v \Rightarrow_s v}\ (\mathit{S\text{-}Val}) \qquad \frac{M \Rightarrow_s M'}{(\lambda x.M) \Rightarrow_s (\lambda x.M')}\ (\mathit{S\text{-}Lam}) \qquad \frac{M_1 \Rightarrow_s M_1' \quad M_2 \Rightarrow_s M_2'}{(M_1\ M_2) \Rightarrow_s (M_1'\ M_2')}\ (\mathit{S\text{-}App}) \qquad \frac{M \Rightarrow_s M'}{\langle M \rangle \Rightarrow_s \langle M' \rangle}\ (\mathit{S\text{-}Rt})$$

$$\frac{M \Rightarrow_s M'}{(\mathcal{S}k.M) \Rightarrow_s (\mathcal{S}k.M')}\ (\mathit{S\text{-}St}) \qquad \frac{M \Rightarrow_s M'}{(\mathtt{future}\ M) \Rightarrow_s (\mathtt{future}\ M')}\ (\mathit{S\text{-}Ft}) \qquad \frac{M \Rightarrow_s M' \quad S \Rightarrow S'}{\langle \mathtt{flet}\ (p\ M)\ S \rangle \Rightarrow_s \langle \mathtt{flet}\ (p\ M')\ S' \rangle}\ (\mathit{S\text{-}Flet})$$

補助的遷移 $\Rightarrow_s$ の定義（ストアに対して）：

$$\frac{V_i \Rightarrow_s V_i' \quad (1 \le i \le n)}{\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\} \Rightarrow_s \{b_1 \mapsto V_1', \cdots, b_n \mapsto V_n'\}}\ (\mathit{S\text{-}Store})$$

図 6 並列遷移
Fig. 6 Definition of parallel reduction rules.

評価文脈でもない表現（□ を 2 つ以上含むなど）に対する二項関係にもなっているが，そのようなケースは使わない．

二項関係 $\Rightarrow_s$ は，flet の中でのみ $\Rightarrow$ による遷移が起こることを意味している．(*S-Flet*) では，前提の 1 つが $S \Rightarrow_s S'$ でなく $S \Rightarrow S'$ であることに注意されたい．

$\langle\!\langle \theta,\ M \rangle\!\rangle \Rightarrow \langle\!\langle \theta',\ M' \rangle\!\rangle$ が成立するとき，それを示すのに使った最後の規則が (*Spec*) 以外であるとき，この遷移を必須遷移と呼び，最後の規則が (*Spec*) の場合，投機的遷移と呼ぶ．

4.2.2 極大並列遷移の定義

この項では，極大並列遷移 $(-)^\star$ を状態に対して，補助的遷移 $(-)^\#$ を項・評価文脈・ストアに対して，補助的遷移 $(-)^\circ$ を（特定の形をした）項に対して定義する．また，以下の定義では，複数の定義節にマッチする場合は，先に現れる定義節を優先的に適用することとし，極大並列遷移および補助遷移の結果を一意的に定める．

極大並列遷移 $(-)^\star$ の定義（状態に対して）：

$\langle\!\langle \theta,\ E[R] \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\#,\ E^\#[R^\circ] \rangle\!\rangle$ (R° が定義されているとき)
$\langle\!\langle \theta,\ E[(\mathtt{make}\ V)] \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[b] \rangle\!\rangle \quad (b \text{ fresh})$
$\langle\!\langle \theta \cup \{b \mapsto V_1\},\ E[((\mathtt{set!}\ b)\ V)] \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[\mathbf{void}] \rangle\!\rangle$
$\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\mathtt{deref}\ b)] \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#] \rangle\!\rangle$
$\langle\!\langle \theta,\ E[\langle \mathtt{flet}\ (p\ V)\ \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \rangle] \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\# \cup (\theta_2\{p := V^\#\}),\ E^\#[M_2\{p := V^\#\}] \rangle\!\rangle$
($\langle\!\langle \theta_2,\ M_2 \rangle\!\rangle = \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle^\star$ かつ
$Dom(\theta^\#) \cap Dom(\theta_2) = \emptyset$ のとき)
$\langle\!\langle \theta,\ M \rangle\!\rangle^\star$
$= \langle\!\langle \theta^\#,\ M^\# \rangle\!\rangle$ (上記以外の形のとき)

特に，flet に対する定義の中で，$(-)^\star$ を使っていることに注意されたい．

© 2013 Information Processing Society of Japan

補助的遷移 $(-)^\circ$ の定義（以下の左辺の形の項に対して）：

$$((\lambda x.M)\ V)^\circ = M^\#\{x := V^\#\}$$
$$\langle V\rangle^\circ = V^\#$$
$$F[(\mathcal{S}k.M)]^\circ = \langle M^\#\{k := (\lambda v.F^\#[v])\}\rangle \quad (v \text{ fresh})$$
$$F[(\texttt{future}\ M)]^\circ = \langle \texttt{flet}\ (p\ M^\#)\ \{\!\{\emptyset,\ F^\#[p]\}\!\}\rangle \ (p \text{ fresh})$$

補助的遷移 $(-)^\#$ の定義（項，文脈に対して）：

$$v^\# = v\ (v = x, c, b, p, \Box \text{ のとき})$$
$$(\lambda x.M)^\# = (\lambda x.M^\#)$$
$$(M\ N)^\# = (M^\#\ N^\#)$$
$$\langle M\rangle^\# = \langle M^\#\rangle$$
$$(\mathcal{S}k.M)^\# = (\mathcal{S}k.M^\#)$$
$$\langle \texttt{flet}\ (p\ M)\ S\rangle^\# = \langle \texttt{flet}\ (p\ M^\#)\ S^\star\rangle$$
$$(\texttt{future}\ M)^\# = (\texttt{future}\ M^\#)$$

補助的遷移 $(-)^\#$ の定義（ストアに対して）：

$$\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}^\# = \{b_1 \mapsto {V_1}^\#, \cdots, b_n \mapsto {V_n}^\#\}$$

$M^\#$ の定義では，$\Rightarrow_s$ の定義と同様に，flet に対してのみ実質的な遷移が起きる．flet に対する定義の右辺で，$S^\#$ でなく $S^\star$ を使っていることに注意されたい．

4.2.3 並列遷移の性質と定理 4 の証明

並列遷移と極大並列遷移に対する鍵となる補題は以下のものである．

補題 3 *AM-PFSR* の状態 S_i，ストア θ_i，項 M_i，評価文脈 E_i（$i = 1, 2$）に対して，以下の性質が成立する．なお，ストア・項・評価文脈のいずれかを表すメタ変数として X_i を使う．(*0*) $S_1 \Rightarrow S_1$，(*0'*) $X_1 \Rightarrow_s X_1$，(*1*) $S_1 \to S_2$ ならば $S_1 \Rightarrow S_2$．また，$S_1 \to S_2$ が必須遷移ならば，$S_1 \Rightarrow S_2$ は必須遷移である，(*2*) $S_1 \Rightarrow S_2$ ならば $S_1 \to^* S_2$，(*2'*) $\theta_1 \Rightarrow_s \theta_2$，$E_1 \Rightarrow_s E_2$，$M_1 \Rightarrow_s M_2$ ならば $\{\!\{\theta_1,\ E_1[M_1]\}\!\} \to^* \{\!\{\theta_2,\ E_2[M_2]\}\!\}$，(*3*) $S_1 \Rightarrow {S_1}^\star$，(*3'*) $X_1 \Rightarrow_s {X_1}^\#$，(*4*) $S_1 \Rightarrow S_2$ ならば $S_2 \Rightarrow {S_1}^\star$，(*4'*) $X_1 \Rightarrow_s X_2$ ならば $X_2 \Rightarrow_s {X_1}^\#$，(*5*) $S_1 \Rightarrow S_2$ が *(Join)* 以外の必須遷移ならば，$S_2 \Rightarrow {S_1}^\star$ は投機的遷移である，(*5'*) $S_1 \Rightarrow S_2$ が投機的遷移で，$S_1 \Rightarrow {S_1}^\star$ が必須遷移ならば，$S_2 \Rightarrow {S_1}^\star$ は必須遷移である． □

補題 3 の証明は付録に掲載する．

定理 4 の証明．$S_1 \to^* S_2$ かつ $S_1 \to^* S_3$ とすると，補題 3 の（*1*）から $S_1 \Rightarrow^* S_2$ かつ $S_1 \Rightarrow^* S_3$ である．これらの $\Rightarrow$ による遷移の回数をそれぞれ m，n とし，m と n の最大値を k とする．S_1 に極大並列遷移 $(-)^*$ を k 回適用して得られる状態を S_4 とすると，補題 3 の（*4*）より，$S_2 \Rightarrow^* S_4$ かつ $S_3 \Rightarrow^* S_4$ である．よって，補題 3 の（*2*）より $\to$ は合流性を満たす． ■

4.3 並列機械の遷移の一意性

定理 2 を証明する準備として，補題 4 と定理 5 を示す．

補題 4 並列機械 *AM-PFSR* で，$S_1 \to S_1'$ かつ無限遷移列 $S_1 \to S_2 \to \cdots$ が無限個の必須遷移を含めば，$S_1' \to S_2' \to \cdots$ となる遷移列で無限個の必須遷移を含むものが存在する． □

この補題の証明は付録に掲載する．

定理 5 $A_1, A_2 \in \mathit{eval}_p(P)$ ならば $A_1 = A_2$ □

定理 5 の証明．$A_1, A_2 \in \mathit{eval}_p(P)$ とする．

(Case 1) $A_1 = A_2 = \bot$ あるいは $A_1 = A_2 = \mathbf{error}$ のときは $A_1 = A_2$ である．

(Case 2) A_1，A_2 がともに $\bot$ でも **error** でもなければ，合流性より $A_1 = A_2$ である．(Case 3) 一方が **error** で他方が $\bot$ でも **error** でもないとする．S_0 から，必須遷移について stuck した状態 S_1 へ行く遷移列と，S_0 から $S_2 = \{\!\{\theta,\ V\}\!\}$ へ行く遷移列がある．合流性から，$S_1 \to^* S_3$，$S_2 \to^* S_3$ なる状態 S_3 が存在する．$S_1 \to^* S_3$ より，S_3 も stuck した状態であるが，これは $S_2 \to^* S_3$ と矛盾する．

(Case 4) $A_1, \bot \in \mathit{eval}_p(P)$ かつ $A_1 \neq \bot, \mathbf{error}$ と仮定する．

仮定より，$\{\!\{\emptyset,\ P\}\!\} = S_1 \to S_2 \to \cdots$ となる遷移列で，必須遷移を無限個含むものがある．また，$S_1 = S_1' \to S_2' \to \cdots \to S_{n+1}' = \{\!\{\theta,\ V\}\!\}$ かつ $\mathit{Unload}(V) = A_1$ となる遷移列が存在する．後者の遷移列の長さ n に関する帰納法で矛盾を導く．$n = 0$ のとき，補題 1 より $S_1 = S_1' = \{\!\{\emptyset,\ V\}\!\}$ であり，S_1 から始まる必須遷移はないので矛盾する．$n > 0$ のとき，補題 4 より，S_2' から始まる遷移列で，必須遷移を無限個含むものがある．$S_2' \to^* S_{n+1}' = \{\!\{\theta, V\}\!\}$ の長さは $n - 1$ であるので，帰納法の仮定により矛盾する．

(Case 5) $\mathbf{error}, \bot \in \mathit{eval}_p(P)$ のとき，(Case 4) と同様にして矛盾する． ■

これまでの議論によって，定理 2 が証明される．

定理 2 の証明．定理 1 から，$\mathit{eval}_s(P)$ は，ただ 1 つの要素 $A \in \mathbb{A}$ を持つ集合 $\{A\}$ であることが分かる．また，補題 1 と定理 5 より，$\mathit{eval}_p(P)$ は $\{A'\}$ と表せる．ここで，定理 3 より，$\{A\} \subseteq \{A'\}$ であるので，$A = A'$ である． ■

4.4 先行研究の証明方法との比較

Flanagan ら [15] は，コントロール・オペレータがない future を持つ体系の合流性を示した．Moreau [6], [7] は，彼らの証明を拡張して call/cc のある体系の合流性を証明したが，単純な拡張では済まなかった．例として，遷移列 $S_1 \to S_2 \to S_3$ を考えよう[*11]．

[*11] ここでは，Moreau の体系を本論文の記法に合わせて書き直している．

© 2013 Information Processing Society of Japan

$S_1 = \{\!|\theta,\ \langle \texttt{flet}\ (p\ (\texttt{call/cc}\ (\lambda k.M)))\ \{\!|\emptyset,\ N|\!\}\rangle|\!\}$

$S_2 = \{\!|\theta,\ \langle \texttt{flet}\ (p\ (\texttt{call/cc}\ (\lambda k.M)))\ \{\!|\emptyset,\ N'|\!\}\rangle|\!\}$

（ただし，$\{\!|\emptyset,\ N|\!\} \to \{\!|\emptyset,\ N'|\!\}$）

$S_3 = \{\!|\theta,\ M\{k := K\}|\!\}$

（ただし，$K = (\lambda v.(\texttt{abort}\ \langle \texttt{flet}\ (p\ v)\ \{\!|\emptyset,\ N|\!\}\rangle))$）

ここで，call/cc は shift と類似した計算規則を持つが，捕捉する継続に abort を含む点が異なる[*12]．彼の体系では，関数抽象の内部の計算は進まないため，K の中の N の計算を進めて N' にすることができず，S_2 と S_3 を合流させることができない．この困難に対処するため，彼は，「捕捉された継続の内部の n ステップ以下の計算を除いて，2 つの項が一致すること」を意味する二項関係 $\overset{\sim}{\to}_n$ を導入し，この $\overset{\sim}{\to}_n$ の差を除いて合流性が成立することを示し，最終的に透過性を証明した．

しかし，Moreau の手法は，shift/reset を含む体系では使えない．call/cc で捕捉された継続は abort をともなっているため関数合成ができない（その継続を使うと，現在の継続を捨ててしまう）が，shift で捕捉される継続は合成可能である．すると，合流性の証明において，2 つの継続を合成する場合に，$\overset{\sim}{\to}_n$ が推移的でなければならないが，「n ステップ以下の計算で一致する」という関係は推移律が成立しないからである．

そこで，我々は新たな手法を考案した．分析の結果，上記の合流性の問題を解決するためには，体系の遷移の定義を変更し，flet 項の計算はどのような文脈の下でも適用可能とすることが必要であると判断した．驚くべきことに，この新しい遷移の下では，Takahashi の並行簡約の証明技法を素直に適用することにより合流性が証明できることが分かった．我々の証明は遷移のステップ数に関する議論をいっさい使っておらず，Moreau の証明のように $\overset{\sim}{\to}_n$ による「誤差」を意識せずに済むため，見通しが良く簡潔である．

本論文では，Moreau スタイルの定式化より遷移を増やして合流性・透過性を証明したので，真に強い結果を証明したことになる．また，flet 項が並列に動作するプロセスを表すと考えれば，上記の S_3 における K の中の flet 項の計算が shift の実行後も動作することは自然である．むしろ，あるプロセスにおけるコントロール・オペレータの実行によって他のプロセスを完全に停止させる，という Moreau の意味論は，バリア同期などの仕組みを必要とし，並列実行環境では好ましくない可能性がある．本論文は，証明の都合で生じた遷移の追加が，処理系実装上も意義を持つ可能性がある，という意味で，理論と実践の新たな関係を示唆するものと考えている．

[*12] abort は $\{\!|\theta,\ E[(\texttt{abort}\ V)]|\!\} \to \{\!|\theta,\ V|\!\}$ を満たすコントロール・オペレータである．

5. まとめと今後の課題

本研究では，並列計算オペレータ future と，限定継続オペレータ shift/reset が同時に存在する言語 λ_{sr}^{ft} を提案した．そして，λ_{sr}^{ft} に対して，2 つの抽象機械を定義し，両者の結果が一致することを示すことで，future が意味的透過性を有することを明らかにした．

今後の課題としては，(1) shift/reset と異なるコントロール・オペレータ（階層化 shift/reset や control/prompt など）について，透過性の検証すること，(2) CEK 機械のように，より現実のマシンに近い抽象機械を構築することで，並列性の有無を検証すること，などがあげられる．

謝辞 きわめて詳細にわたる建設的コメントを多数いただいた査読者の方々に感謝する．

参考文献

[1] Plotkin, G.: Call-by-Name, Call-by-Value and the Lambda-Calculus, *Theoretical Computer Science*, Vol.1, No.2, pp.125–159 (1975).
[2] Danvy, O. and Filinski, A.: Abstracting Control, *Proc. 1990 ACM Conference on LISP and Functional Programming*, pp.151–160, ACM (1990).
[3] Asai, K.: Offline Partial Evaluation for Shift and Reset, *Proc. 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp.3–14, ACM (2004).
[4] 田中麻峰，亀山幸義：限定継続に基づくスケーラブルなウェブアプリケーション構築手法，電子情報通信学会技術研究報告，ソフトウェアサイエンス，Vol.109, No.456, pp.163–168 (2010).
[5] Katz, M. and Weise, D.: Continuing Into the Future: On the Interaction of Futures and First-Class Continuations, *Proc. 1990 ACM Conference on LISP and Functional Programming*, pp.176–184, ACM (1990).
[6] Moreau, L.: A Parallel Functional Language with First-Class Continuations. Programming Style and Semantics, *Computers and Artificial Intelligence*, Vol.14, No.2, p.173 (1995).
[7] Moreau, L.: The Semantics of Scheme with Future, *Proc. 1st ACM SIGPLAN International Conference on Functional Programming*, pp.146–156, ACM (1996).
[8] Takahashi, M: Parallel Reductions in λ-Calculus, *J. Symbolic Computation*, Vol.7, pp.113–123 (1989).
[9] Danvy, O. and Filinski, A.: Representing Control: A Study of the CPS Transformation, *Mathematical Structures in Computer Science*, Vol.2, No.4, pp.361–391 (1992).
[10] Filinski, A.: Representing Monads, *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.446–457, ACM (1994).
[11] Niehren, J., Schwinghammer, J. and Smolka, G.: A Concurrent Lambda-Calculus with Futures, *Theoretical Computer Science*, Vol.364, No.3, pp.338–356 (2006).
[12] Niehren, J., Sabel, D., et al.: Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures, *Electronic Notes in Theoretical Computer Science*, Vol.173, pp.313–337 (2007).
[13] Halstead, Jr, R.: Multilisp: A Language for Concurrent

© 2013 Information Processing Society of Japan

Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.4, pp.501–538 (1985).
[14] Gasbichler, M. and Sperber, M.: Final Shift for Call/cc:: Direct Implementation of Shift and Reset, *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, pp.271–282, ACM (2002).
[15] Flanagan, C. and Felleisen, M.: The Semantics of Future and Its Use in Program Optimization, *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.209–220, ACM (1995).

付　　録

A.1 定理 3 の証明

本節を通じて，$(-)^s$ は AM-FSR の要素を表し，$(-)^p$ は AM-PFSR の要素を表す．この区別は重要であり，たとえば，E^s は AM-FSR の評価文脈であり，AM-PFSR でも文脈であるが，`future` を含みうるため，AM-PFSR の評価文脈になるとは限らない．

二項関係 $X^s \sim Y^p$（X^s は AM-FSR の状態，項，ストア，文脈のいずれか，Y^p は AM-PFSR の対応する構文要素），$F^s \approx_f C^p$（F^s は AM-FSR の純粋評価文脈），$E^s \approx_e C^p$（E^s は AM-FSR の一般評価文脈）を同時に帰納的に定義する．

$X^s \sim Y^p$ の定義（X^s は項か文脈）：

$$\frac{(d = \Box, x, b, c)}{d \sim d} \qquad \frac{M^s \sim M^p}{(\lambda x.M^s) \sim (\lambda x.M^p)}$$

$$\frac{M_1^s \sim M_1^p \quad M_2^s \sim M_2^p}{(M_1^s\ M_2^s) \sim (M_1^p\ M_2^p)} \qquad \frac{F^s \approx_f C^p \quad M^s \sim M^p}{F^s[M^s] \sim C^p[M^p]}\ (*)$$

$$\frac{M^s \sim M^p}{(\mathcal{S}k.M^s) \sim (\mathcal{S}k.M^p)} \qquad \frac{M^s \sim M^p}{(\texttt{future}\ M^s) \sim (\texttt{future}\ M^p)}$$

ルール (*) が，本節の証明で本質的なものである．後述するように，$\langle\Box\rangle \approx_f \langle\Box\rangle$ が成立するので，$M^s \sim M^p$ ならば $\langle M^s\rangle \sim \langle M^p\rangle$ であり，AM-FSR の任意の項 M^s に対して $M^s \sim M^s$ が成立する．

$X^s \sim Y^p$ の定義（X^s は状態かストア）：

$$\frac{\theta^s \sim \theta^p \quad M^s \sim M^p}{[\![\theta^s,\ M^s]\!] \sim [\![\theta^p,\ M^p]\!]}$$

$$\frac{V_i^s \sim V_i^p\ \ (\text{for } i = 1, \cdots, n)}{\{b_1 \mapsto V_1^s, \cdots, b_n \mapsto V_n^s\} \sim \{b_1 \mapsto V_1^p, \cdots, b_n \mapsto V_n^p\}}$$

$F^s \approx_f C^p$ の定義：

AM-FSR の純粋評価文脈 F^s の構文は，$F^s ::= \langle\Box\rangle \mid F^s[(\Box\ M^s)] \mid F^s[(V^s\ \Box)] \mid F^s[\texttt{future}\ \Box]$ と定義できるので，この構成法に関して帰納的に $\approx_f$ を定義する．

$$\frac{}{\langle\Box\rangle \approx_f \langle\Box\rangle} \qquad \frac{F^s \approx_f C^p \quad M^s \sim M^p}{F^s[(\Box\ M^s)] \approx_f C^p[(\Box\ M^p)]}$$

$$\frac{F^s \approx_f C^p \quad V^s \sim V^p}{F^s[(V^s\ \Box)] \approx_f C^p[(V^p\ \Box)]}$$

$$\frac{F^s \approx_f C^p}{F^s[(\texttt{future}\ \Box)] \approx_f \langle \texttt{flet}\ (p\ \Box)\ [\![\emptyset,\ C^p[p]]\!]\rangle\ (\text{p fresh})}$$

$E^s \approx_e C^p$ の定義：

$$\frac{G^s \sim C^p}{G^s \approx_e C^p} \qquad \frac{E^s \approx_e C_1^p \quad F^s \approx_f C_2^p}{E^s[F^s] \approx_e C_1^p[C_2^p]}$$

$F^s \approx_f C^p$ の直感的な意味は，F^s における $\Box$ の周りの `future` をすべて，(*fork*) 規則で遷移させて C^p が得られるということである．たとえば，$F^s \equiv \langle((\texttt{future}\ ((\texttt{future}\ \Box)\ x))\ y)\rangle$ のとき，$F^s \approx_f \langle \texttt{flet}\ (p_2\ \Box)\ \langle \texttt{flet}\ (p_1\ (p_2\ x))\ \langle(p_1\ y)\rangle\rangle\rangle$ となる．

AM-FSR において，$\langle C^s\rangle$ の形をしている一般評価文脈 E^s（一番外側に reset がある一般評価文脈）を，プログラム評価文脈と呼ぶ．AM-PFSR でも同様である．

$F^s \sim C^p$ が成立しても，C^p は AM-PFSR の純粋評価文脈とは限らない．なぜなら，$\langle(\texttt{future}\ \Box)\rangle \sim \langle(\texttt{future}\ \Box)\rangle$ が成立するので．一方，次の補題で示すとおり，$F^s \approx_f C^p$ が成立すれば，C^p は AM-PFSR の純粋評価文脈になる．

補題 5　X^s が *AM-FSR* の項，値，文脈，状態，ストアであり，$X^s \sim Y^p$ ならば，Y^p は *AM-PFSR* の対応する構文要素である．$F^s \approx_f C^p$ ならば，C^p は *AM-PFSR* の純粋評価文脈である．プログラム評価文脈 E^s について，$E^s \approx_e C^p$ であるならば，C^p は *AM-PFSR* のプログラム評価文脈である．□

証明．二項関係 $\sim$，$\approx_f$，$\approx_e$ の定義に関する帰納法で証明できる．■

補題 6（分解補題）(*1*) 項か文脈である X^s に対して，$E^s[\langle X^s\rangle] \sim Y^p$ ならば，$Y^p \equiv C^p[\langle Z^p\rangle]$ かつ $E^s \sim C^p$ かつ $X^s \sim Z^p$ となる C^p と Z^p がある．

(*2*) M^s が値 V^s か $(V_1^s\ V_2^s)$ の形の項とすると，$F^s[M^s] \sim N^p$ ならば，$N^p \equiv F^p[M^p]$ かつ $F^s \sim F^p$ かつ $M^s \sim M^p$ となる F^p と M^p がある．

(*3*) $G^s[M^s] \sim N^p$ ならば，$N^p \equiv C^p[M^p]$ かつ $G^s \sim C^p$ かつ $M^s \sim M^p$ となる C^p と M^p がある．□

証明．(*1*)，(*3*) は容易である．(*2*) で $M^s \equiv (V_1^s\ V_2^s)$ のときは，$F^s[M^s] \equiv F_1^s[X^s]$ の形になるすべてのケースについて補題が成立することを示せばよい．■

補題 7　(*1*) $X^s \sim Y^p$ かつ $V^s \sim V^p$ ならば $X^s\{x := V^s\} \sim Y^p\{x := V^p\}$ である．また，文脈 C^s に対して，$C^s \sim C^p$ かつ $M^s \sim M^p$ ならば $C^s[M^s] \sim C^p[M^p]$ である．

(*2*) $F^s \approx_f C^p$ ならば $F^s \sim C^p$ である．また，$E^s \approx_e C^p$ ならば $E^s \sim C^p$ である．□

証明．(*1*) は X^s および C^s に関する帰納法で証明できる．

© 2013 Information Processing Society of Japan

(2) の前半は (*) ルールで $M^s = M^p = \Box$ とおけば得られ，後半は E^s の構成に関する帰納法で得られる． ■

次に，AM-PFSR において「文脈に対する遷移」という概念を次のように定義する．すべての項 M^p とストア θ^p およびプログラム評価文脈 E^p に対して，(*fork*) 遷移のみで，$\langle\!\langle \theta^p,\ E^p[C_1^p[M^p]]\rangle\!\rangle \to^* \langle\!\langle \theta^p,\ E^p[C_2^p[M^p]]\rangle\!\rangle$ となるとき，および，そのときに限り，$C_1^p \to_f^* C_2^p$ と定義する．文脈に対する遷移は代入に関して閉じている．すなわち，$C_1^p \to_f^* C_2^p$ であるならば，任意の C_3^p について，$C_1^p[C_3^p] \to_f^* C_2^p[C_3^p]$ である．

次の補題 8 が本節の証明の鍵となる補題である．

補題 8 (1) $F^s \approx_f F^p$ かつ $G^s \sim C^p$ ならば，$F^s[G^s] \approx_f F_2^p$ かつ $F^p[C^p] \to_f^* F_2^p$ となる F_2^p がある．

(2) $F^s \sim C^p$ ならば，$F^s \approx_f F^p$ かつ $C^p \to_f^* F^p$ となる F^p がある．

(3) E^s をプログラム評価文脈とする．$E^s \sim C^p$ ならば，$E^s \approx_e E^p$ かつ $C^p \to_f^* E^p$ となる E^p がある． □

(1) の証明．G^s について，$G^s ::= \Box \mid G^s[(\Box\ M^s)] \mid G^s[(V^s\ \Box)] \mid G^s[(\texttt{future}\ \Box)]$ という構成に関する帰納法で証明する．

(Case 1：$G^s \equiv \Box$ のとき) $C^p \equiv \Box$ である．よって，$F_2^p \equiv F^p$ とおけばよい．

(Case 2：$G^s \equiv G_1^s[(\texttt{future}\ \Box)]$ のとき) 補題 6 より，$C^p \equiv C_1^p[(\texttt{future}\ \Box)]$ かつ，$G_1^s \sim C_1^p$ となる C_1^p がある．G_1^s に対する帰納法の仮定より，$F^s[G_1^s] \approx_f F_1^p$ かつ $F^p[C_1^p] \to_f^* F_1^p$ となる F_1^p が存在する．ここで，$F_2^p \equiv \langle \texttt{flet}\ (p\ \Box)\ \langle\!\langle \emptyset,\ F_1^p[p]\rangle\!\rangle\rangle$ とおけば，$\approx_f$ の定義より，$F^s[G^s] \equiv F^s[G_1^s[(\texttt{future}\ \Box)]] \approx_f F_2^p$ がいえる．また，次の遷移列があるので補題の結論が導けた．

$$\begin{aligned} F^p[C^p] &\equiv F^p[C_1^p[(\texttt{future}\ \Box)]] \\ &\to_f^* F_1^p[(\texttt{future}\ \Box)] \\ &\to_f \langle \texttt{flet}\ (p\ \Box)\ \langle\!\langle \emptyset,\ F_1^p[p]\rangle\!\rangle\rangle \equiv F_2^p \end{aligned}$$

(Case 3：$G^s \equiv G_1^s[(\Box\ M^s)]$ のとき) 補題 6 より，$M^s \sim M^p$ かつ $C^p \equiv C_1^p[(\Box\ M^p)]$ かつ $G_1^s \sim C_1^p$ となる M^p と C^p がある．G_1^s に対する帰納法の仮定より，$F^s[G_1^s] \approx_f F_1^p$ かつ $F^p[C_1^p] \to_f^* F_1^p$ となる F_1^p が存在する．ここで，$F_2^p \equiv F_1^p[(\Box\ M^p)]$ とおくと，$\approx_f$ の定義より，$F^s[G^s] \equiv F^s[G_1^s[(\Box\ M^s)]] \approx_f F_2^p$ がいえる．また，次の遷移列があるので補題の結論が導けた．

$$F^p[C^p] \equiv F^p[C_1^p[(\Box\ M^p)]] \to_f^* F_1^p[(\Box\ M^p)] \equiv F_2^p$$

(Case 4：$G^s \equiv G_1^s[(V^s\ \Box)]$ のとき) Case 3 と同様に証明できる．

(2) の証明．$F^s \sim C^p$ を導いた規則は (*) であるので，ある F_1^s, G^s, F_1^p, C_2^p に対して，$F^s \equiv F_1^s[G^s]$, $C^p \equiv F_1^p[C_2^p]$, $F_1^s \approx_f F_1^p$, $G^s \sim C_2^p$ となる．そこで (1) を使えばよい．

(3) の証明．プログラム評価文脈 E^s の構文は，$E^s ::= F^s \mid E^s[F^s]$ と定義できる．この構文に関する帰納法で (3) は証明できる． ■

定理 6 (シミュレーション) S_1^s をプログラム状態 ($\langle\!\langle \theta,\ P\rangle\!\rangle$ の形の状態) とする．$S_1^s \sim S_1^p$ かつ $S_1^s \to S_2^s$ ならば，$S_2^s \sim S_2^p$ かつ $S_1^p \to^* S_2^p$ となる状態 S_2^p が存在する．また，後者の遷移列には必須遷移を *1* つ以上含む． □

証明．$S_1^s \to S_2^s$ の遷移の種類による場合分けで証明する．ここでは，(*app*) と (*fid*) の遷移について証明を述べる．

(Case：(*app*) のとき) $S_1^s \equiv \langle\!\langle \theta^s,\ E^s[((\lambda x.M^s)\ V^s)]\rangle\!\rangle$, $S_2^s \equiv \langle\!\langle \theta^s,\ E^s[M^s\{x := V^s\}]\rangle\!\rangle$ である．補題 6 より，$S_1^p \equiv \langle\!\langle \theta^p,\ C^p[((\lambda x.M^p)\ V^p)]\rangle\!\rangle$, $\theta^s \sim \theta^p$, $E^s \sim C^p$, $M^s \sim M^p$, $V^s \sim V^p$ となる θ^p, C^p, M^p, V^p がある．補題 8 より，$E^s \approx_e E^p$ かつ $C^p \to_f^* E^p$ となる E^p がある．よって，

$$S_1^p \to^* \langle\!\langle \theta^p,\ E^p[((\lambda x.M^p)\ V^p)]\rangle\!\rangle$$

となる．E^p はプログラム評価文脈なので，遷移規則 (*app*) (必須遷移) により，

$$\langle\!\langle \theta^p,\ E^p[((\lambda x.M^p)\ V^p)]\rangle\!\rangle \to \langle\!\langle \theta^p,\ E^p[M^p\{x := V^p\}]\rangle\!\rangle$$

となる．$S_2^p \equiv \langle\!\langle \theta^p,\ E^p[M^p\{x := V^p\}]\rangle\!\rangle$ とおけば，$S_1^p \to^* S_2^p$ (必須遷移を含む遷移列) であり，また，補題 7 より，$\langle\!\langle \theta^s,\ E^s[M^s\{x := V^s\}]\rangle\!\rangle \sim S_2^p$ である．

(Case：(*fid*) のとき) $S_1^s \equiv \langle\!\langle \theta^s,\ E^s[(\texttt{future}\ V^s)]\rangle\!\rangle$, $S_2^s \equiv \langle\!\langle \theta^s,\ E^s[V^s]\rangle\!\rangle$ である．

S_1^s はプログラム状態なので，E^s はプログラム評価文脈であり，$E^s \equiv E_1^s[F^s]$ の形である．よって，ある θ^p, C_1^p, C_2^p, V^p に対して，$S_1^p \equiv \langle\!\langle \theta^p,\ C_1^p[C_2^p[V^p]]\rangle\!\rangle$, $\theta^s \sim \theta^p$, $E_1^s \sim C_1^p$, $F^s[(\texttt{future}\ \Box)] \sim C_2^p$, $V^s \sim V^p$ である．補題 8 より，$E_1^s \approx_e E_1^p$ かつ $C_1^p \to_f^* E_1^p$ となる E_1^p がある．ここで，$F^s[(\texttt{future}\ \Box)] \sim C_2^p$ をルール (*) で導出する際，純粋評価文脈 $F^s[(\texttt{future}\ \Box)]$ の全体が $\approx_f$ で関係付けられるか，その真部分 (つまり，F^s の一部) が $\approx_f$ で関係付けられるかで場合分けする．

(Case 1：$F^s[(\texttt{future}\ \Box)] \approx_f C_2^p$ のとき) 二項関係 $\approx_f$ の定義より，$C_2^p \equiv \langle \texttt{flet}\ (p\ \Box)\ F^p[p]\rangle$ (p はフレッシュ) かつ $F^s \approx_f F^p$ となる F^p がある．$S_2^p \equiv \langle\!\langle \theta^p,\ E_1^p[F^p[V^p]]\rangle\!\rangle$ とおくと，$S_1^s \sim S_2^p$ となる．また，以下の遷移列がある．

$$\begin{aligned} S_1^p &\equiv \langle\!\langle \theta^p,\ C_1^p[\langle \texttt{flet}\ (p\ V^p)\ F^p[p]\rangle]\rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p,\ E_1^p[\langle \texttt{flet}\ (p\ V^p)\ F^p[p]\rangle]\rangle\!\rangle \\ &\to \langle\!\langle \theta^p,\ E_1^p[F^p[V^p]]\rangle\!\rangle \equiv S_2^p \text{ (必須遷移を含む遷移列)} \end{aligned}$$

(Case 2：ある F_1^s, G^s, F^p, C_3^p に対して $F^s \equiv F_1^s[G^s]$, $F_1^s \approx_f F^p$, $C_2^p \equiv F^p[C_3^p]$, $G^s[(\texttt{future}\ \Box)] \sim C_3^p$ となる

© 2013 Information Processing Society of Japan

とき）

$G^s[(\texttt{future}\ \square)] \sim C_3^p$ に補題 6 を適用して，ある C_4^p に対して，$C_3^p \equiv C_4^p[(\texttt{future}\ \square)]$ かつ $G^s \sim C_4^p$ となる．補題 8 の（*1*）を $F_1^s \approx_f F^p$ と $G^s \sim C_4^p$ に適用して，ある F_2^p に対して，$F_1^s[G^s] \approx_f F_2^p$ かつ $F^p[C_4^p] \to_f^* F_2^p$ となる．$S_2^p \equiv \langle\!\langle \theta^p,\ E_1^p[F_2^p[V^p]] \rangle\!\rangle$ とおくと，$S_2^s \sim S_2^p$ である．また，

$$\begin{aligned} S_1^p &\equiv \langle\!\langle \theta^p,\ C_1^p[F^p[C_4^p[(\texttt{future}\ V^p)]]] \rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p,\ E_1^p[F^p[C_4^p[(\texttt{future}\ V^p)]]] \rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p,\ E_1^p[F_2^p[(\texttt{future}\ V^p)]] \rangle\!\rangle \\ &\to \langle\!\langle \theta^p,\ E_1^p[\langle \texttt{flet}\ (p\ V^p)\ \langle\!\langle \emptyset,\ F_2^p[p] \rangle\!\rangle \rangle] \rangle\!\rangle \\ &\to \langle\!\langle \theta^p,\ E_1^p[F_2^p[V^p]] \rangle\!\rangle \equiv S_2^p \end{aligned}$$

これは必須遷移を含む遷移列であるので，証明できた．■

定理 3 の証明．P を AM-FSR のプログラムとする．

（Case 1：$eval_s(P) \neq \{\bot\}, \{\textbf{error}\}$ のとき）AM-FSR で $\langle\!\langle \emptyset,\ P \rangle\!\rangle \to^* \langle\!\langle \theta^s,\ V^s \rangle\!\rangle$ である．定理 6 より，AM-PFSR で，$\langle\!\langle \emptyset,\ P \rangle\!\rangle \to^* \langle\!\langle \theta^p,\ V^p \rangle\!\rangle$ かつ $V^s \sim V^p$ となる θ^p と V^p が存在する．一方，$V^s \sim V^p$ ならば $Unload(V^s) = Unload(V^p)$ なので，$eval_s(P) \subseteq eval_p(P)$ である．

（Case 2：$eval_s(P) = \{\textbf{error}\}$ のとき）AM-FSR で $\langle\!\langle \emptyset,\ P \rangle\!\rangle \to^* S_1^s$ であり，状態 S_1^s は `stuck` する．定理 6 より，AM-PFSR で，$\langle\!\langle \emptyset,\ P \rangle\!\rangle \to^* S_1^p$ かつ，$S_1^s \sim S_1^p$ となる状態 S_1^p がある．プログラム状態 S_1^s が `stuck` することと補題 2 から，$S_1^s \equiv \langle\!\langle \theta^s,\ E^s[(V_1^s\ V_2^s)] \rangle\!\rangle$ となる θ^s，E^s，V_1^s，V_2^s がある．$S_1^s \sim S_1^p$ と補題 6 より，$S_1^p \equiv \langle\!\langle \theta^p,\ C^p[(V_1^p\ V_2^p)] \rangle\!\rangle$ かつ $\theta^s \sim \theta^p$，$E^s \sim C^p$，$V_1^s \sim V_1^p$，$V_2^s \sim V_2^p$ となる θ^p，C^p，V_1^p，V_2^p が存在する．補題 8 より，$C^p \to_f^* E^p$ となる E^p が存在する．$S_2^p \equiv \langle\!\langle \theta^p,\ E^p[(V_1^p\ V_2^p)] \rangle\!\rangle$ とおくと，$S_1^p \to^* S_2^p$ である．ここで，S_1^s が `stuck` 状態である原因による場合分けにより，S_2^p が必須遷移について `stuck` 状態であることを示す．一例として，V_1^s が `make`，`set!`，`deref` 以外の定数のときを考える．$V_1^s \sim V_1^p$ より，V_1^p も同じ定数であるため，補題 1 より，S_2^p から始まる必須遷移は存在せず，S_2^p は必須遷移に関して `stuck` する．他の場合も同様である．よって，$eval_s(P) = \{\textbf{error}\} \subseteq eval_p(P)$ である．

（Case 3：$eval_s(P) = \{\bot\}$ のとき）AM-FSR で $\langle\!\langle \emptyset,\ P \rangle\!\rangle \to \cdots$ となる無限の遷移列がある．定理 6 より，$\langle\!\langle \emptyset,\ P \rangle\!\rangle \to \cdots$ となる AM-PFSR の無限遷移列（必須遷移を無限個含むもの）がある．よって，$eval_s(P) \subseteq eval_p(P)$ となる． ■

A.2 補題 3 と補題 4 の証明

4 章の補題 3 と補題 4 を証明する．まず，以下の補題を示す．

補題 9 $\langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2,\ M_2 \rangle\!\rangle$ とする．（*1*）$E_1 \Rightarrow_s E_2$ ならば $\langle\!\langle \theta_1,\ E_1[M_1] \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2,\ E_2[M_2] \rangle\!\rangle$．（*2*）$V_1 \Rightarrow_s V_2$ ならば $\langle\!\langle \theta_1\{x := V_1\},\ M_1\{x := V_1\} \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2\{x := V_2\},\ M_2\{x := V_2\} \rangle\!\rangle$．（*3*）$V_1 \Rightarrow_s V_2$ ならば $\langle\!\langle \theta_1\{p := V_1\},\ M_1\{p := V_1\} \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2\{p := V_2\},\ M_2\{p := V_2\} \rangle\!\rangle$．（*4*）$\theta_1' \Rightarrow_s \theta_2'$ かつ $Dom(\theta_2) \cap Dom(\theta_2') = \emptyset$ ならば，$\langle\!\langle \theta_1 \cup \theta_1',\ M_1 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2 \cup \theta_2',\ M_2 \rangle\!\rangle$． □

この補題は，二項関係 $\Rightarrow$ または $\Rightarrow_s$ と項の構成に関する帰納法で証明できる．

補題 3 の証明．（*0*）と（*0'*）は同時帰納法で証明できる．同様に，（*1*），（*2*）と（*2'*），（*3*）と（*3'*）は，順に帰納法で証明できる．証明は容易なので省略する．

（*4*）と（*4'*）は，$\Rightarrow$ 関係および $\Rightarrow_s$ 関係の導出の大きさに関する同時帰納法で証明できる．ここでは，主要なケースのみ述べる．以下では，$S_1 \Rightarrow S_2$ を導くのに使った最後の規則により場合分けをする．

（Case：$S_1 \Rightarrow S_2$ が（*App*）規則による）$S_1 = \langle\!\langle \theta,\ E[((\lambda x.M)\ V)] \rangle\!\rangle$，$S_2 = \langle\!\langle \theta',\ E'[M'\{x := V'\}] \rangle\!\rangle$，$M \Rightarrow_s M'$，$V \Rightarrow_s V'$，$E \Rightarrow_s E'$，$\theta \Rightarrow_s \theta'$ のとき，（*4'*）に関する帰納法の仮定より，$M' \Rightarrow_s M^\#$，$V' \Rightarrow_s V^\#$，$E' \Rightarrow_s E^\#$，$\theta' \Rightarrow_s \theta^\#$ を得る．また，$S_1{}^\star = \langle\!\langle \theta^\#,\ E^\#[M^\#\{x := V^\#\}] \rangle\!\rangle$ であるので，補題 9 を使って，$S_2 \Rightarrow S_1{}^\star$ を得る．

（Case：$S_1 \Rightarrow S_2$ が（*RtSt*）による）$S_1 = \langle\!\langle \theta,\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle$，$S_2 = \langle\!\langle \theta',\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle] \rangle\!\rangle$，$M \Rightarrow_s M'$，$E \Rightarrow_s E'$，$F \Rightarrow_s F'$，$\theta \Rightarrow_s \theta'$ のとき，（*4'*）に関する帰納法の仮定より，$M' \Rightarrow_s M^\#$，$E' \Rightarrow_s E^\#$，$F' \Rightarrow_s F^\#$，$\theta' \Rightarrow_s \theta^\#$ を得る．また，$S_1{}^\star = \langle\!\langle \theta^\#,\ E^\#[\langle M^\#\{k := (\lambda v.F^\#[v])\}\rangle] \rangle\!\rangle$ であるので，補題 9 を使って，$S_2 \Rightarrow S_1{}^\star$ を得る．

（Case：$S_1 \Rightarrow S_2$ が（*Join*）による）$S_1 = \langle\!\langle \theta,\ E[\langle \texttt{flet}\ (p\ V)\ \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \rangle] \rangle\!\rangle$，$S_2 = \langle\!\langle \theta' \cup \theta_2\{p := V'\},\ E'[M_2\{p := V'\}] \rangle\!\rangle$，$\theta \Rightarrow_s \theta'$，$E \Rightarrow_s E'$，$V \Rightarrow_s V'$，$\langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2,\ M_2 \rangle\!\rangle$ のとき，（*4*）の帰納法の仮定より，$\langle\!\langle \theta_2,\ M_2 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle^\#$（これを $\langle\!\langle \theta_3,\ M_3 \rangle\!\rangle$ とおく），（*4'*）の帰納法の仮定より，$\theta' \Rightarrow_s \theta^\#$，$E' \Rightarrow_s E^\#$，$V' \Rightarrow_s V^\#$ を得る．

補題 9 の（*3*）から，$\langle\!\langle \theta_2\{p := V'\},\ M_2\{p := V'\} \rangle\!\rangle \Rightarrow \langle\!\langle \theta_3\{p := V^\#\},\ M_3\{p := V^\#\} \rangle\!\rangle$ を得る．同様に，補題 9 の（*4*），（*1*）から，$\langle\!\langle \theta' \cup \theta_2\{p := V'\},\ E'[M_2\{p := V'\}] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \theta_3\{p := V^\#\},\ E^\#[M_3\{p := V^\#\}] \rangle\!\rangle$，すなわち $S_2 \Rightarrow S_1{}^\star$ を得る．

（Case：$S_1 \Rightarrow S_2$ が（*Spec*）による）S_1 により場合分けする．ここでは，2 つの場合を示す．

（Subcase：$S_1 = \langle\!\langle \theta,\ E[((\lambda x.M)\ V)] \rangle\!\rangle$）$S_1 \Rightarrow S_2$ が投機的遷移であるので，$S_2 = \langle\!\langle \theta',\ E'[((\lambda x.M')\ V')] \rangle\!\rangle$，$M \Rightarrow_s M'$，$V \Rightarrow_s V'$，$E \Rightarrow_s E'$，$\theta \Rightarrow_s \theta'$ である．このとき，（*App*）の場合と同様に，$S_2 \Rightarrow S_1{}^\star$ を得る．

（Subcase：$S_1 = \langle\!\langle \theta,\ V \rangle\!\rangle$）$S_2 = \langle\!\langle \theta',\ V' \rangle\!\rangle$，$V \Rightarrow_s V'$，$\theta \Rightarrow_s \theta'$ のとき，（*4'*）に関する帰納法の仮定より，

© 2013 Information Processing Society of Japan

$V' \Rightarrow_s V^{\#}$, $\theta' \Rightarrow_s \theta^{\#}$ である．また，$S_1{}^{\star} = \{\!|\theta^{\#}, V^{\#}|\!\}$ であるので，補題 9 を使って，$S_2 \Rightarrow S_1{}^{\star}$ を得る．

以上で，(*4*)，(*4'*) の証明を終える．(*5*)，(*5'*) は，(*4*) の証明を精密に見直すことにより証明できる． ■

補題 4 の証明には，以下の補題が必要である．

補題 10 並列機械 *AM-PFSR* において以下の性質が成立する．

(*1*) $S_1 \to S_2$（投機的遷移）かつ $S_2 \to S_3$（必須遷移）ならば，$S_1 \to^* S_3$ となる遷移列で，先頭の遷移が必須遷移のものが存在する．

(*2*) S_1 から始まる無限遷移列で無限個の必須遷移を含むものがあれば，S_1 から始まる無限遷移列で必須遷移のみから構成されるものがある．

(*3*) (*join*) 遷移のみからなる無限遷移列は存在しない．□

証明（概要）．

(*1*) $S_1 \to S_2$（投機的遷移）かつ $S_2 \to S_3$（必須遷移）ならば，S_1 に後者と同じ必須遷移が適用可能である．その遷移の種類により場合分けすることで容易に証明できる．

(*2*) S_1 から始まる無限遷移列に投機的遷移があるとき，$S_k \to S_{k+1}$（投機的遷移）かつ $S_{k+1} \to S_{k+2}$（必須遷移）が成立する最小の k をとる．これに (*1*) の性質を適用して遷移をつなぎかえると，S_1 から始まる無限遷移列で先頭から k 個以上の遷移が必須遷移のものが得られる．これを繰返すことにより，各 $k \geq 1$ に対して，S_1 から始まる無限遷移列 σ^k で，最初の k 個以上の連続した遷移が必須遷移であり，かつ，σ^{k+1} の最初の k 個までの有限遷移列は，σ^k の最初の k 個までの有限遷移列と同じである，というものが得られる．遷移列 σ^k の n 番目の状態を σ_n^k とおくと，$\sigma_1^1 \to \sigma_2^2 \to \cdots \to \sigma_k^k \to \cdots$ は AM-PFSR の無限遷移列となり，これに含まれるすべての遷移は必須遷移である．

(*3*) 状態や項に対するメタ変数を X とする．表現 X に対して $|X|$ を以下のように定義する．$|p| = 0$，$|V| = 0$，$|(M\ N)| = |M| + |N|$，$|\langle \texttt{flet}\ (p\ M)\ S\rangle| = |M| + |S| + 1$（他のコンストラクタは $|(M\ N)|$ と同様）．

(*join*) 遷移により $S_1 \to S_2$ となるとき，$|S_1| > |S_2|$ であることが容易に示せる．$|X|$ は自然数であるので，(*join*) 遷移のみからなる無限遷移列は存在しない．

補題 4 の証明（概要）．

$S_1 \to S_1'$ かつ S_1 から始まる無限個の必須遷移を含む無限遷移列があるとする．補題 10 の (*2*) より，$S_1 \to S_2 \to S_3 \to \cdots$ となる必須遷移のみからなる無限遷移列がある．補題 10 の (*3*) より，この遷移列には，(*join*) 以外の必須遷移を無限個含む．

ここで，補題 3 の (*1*) と (*4*) より，$S_1{}^{\star} \Rightarrow S_2{}^{\star} \Rightarrow S_3{}^{\star} \Rightarrow \cdots$ となる無限遷移列がある．また，$S_{i+1} \Rightarrow S_i{}^{\star}$ が $i \geq 1$ に対して成立する．

遷移列 $S_1 \to S_2 \to S_3 \cdots$ で初めて (*join*) 以外の必須遷移が現れる場所を $S_i \to S_{i+1}$ とする．補題 3 の (*5*) より，$S_{i+1} \Rightarrow S_i{}^{\star}$ は投機的遷移である（$S_1 \to S_2 \to \cdots$ も並列遷移列と見なしている．以下同様）．$S_{i+1} \to S_{i+2}$ は必須遷移であるため，$S_{i+1} \Rightarrow S_{i+1}{}^{\star}$ も必須遷移であり，補題 3 の (*5'*) を使って，$S_i{}^{\star} \Rightarrow S_{i+1}{}^{\star}$ は必須遷移である．

以上より，$S_1 \to S_2 \to \cdots$ の i 番目の遷移が (*join*) 以外の必須遷移であれば，$S_1{}^{\star} \Rightarrow S_2{}^{\star} \Rightarrow \cdots$ の i 番目の遷移が必須遷移である．よって，後者は必須遷移を無限に含む．$S_1' \Rightarrow S_1{}^{\star}$ と補題 3 の (*2*) より，S_1' から始まり，必須遷移を無限個含む遷移列が存在する． ■

田中 麻峰 （正会員）

2008 年静岡大学情報学部情報科学科卒業．2010 年筑波大学大学院システム情報工学研究科博士前期課程コンピュータサイエンス専攻修了．2013 年同博士後期課程単位取得退学．現在，ニッセイ情報テクノロジー株式会社在籍．

亀山 幸義 （正会員）

筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻教授．プログラム言語論とプログラム検証論に興味を持つ．ACM，日本ソフトウェア科学会，電子情報通信学会各会員．

© 2013 Information Processing Society of Japan