拡張性の高い遷移系を記述可能な領域特化言語

馬谷 誠二

遷移系は，言語処理系やモデル検査器など様々なソフトウェアシステムの中核と言える．小規模な遷移系の形式的定義は，記述・読解・保守等を比較的容易に行うことが可能であるが，一般に，現実世界のシステムに内包されている遷移系は大規模であることが多い．そのため，遷移系記述の拡張性や実行効率を向上させることが重要である．本論文では，実用的なシステムに埋め込むのに十分な実行効率と拡張性を備えた遷移系を，簡潔に記述可能な領域特化言語（DSL）について提案する．まず，一連のプログラミング言語群の操作的意味論を我々の DSL を使って段階的に定義することにより，その拡張性を例示する．さらに，先行研究の一つである Redex との性能比較を行い，我々の DSL が実行時間で大幅な改善を達成できていることを示す．

1 はじめに

遷移系（transition system）は，様々なコンピュータシステムの動作モデルを簡潔に記述するために広く用いられている．例えば，プログラミング言語処理系の構文解析や意味解析，プログラム検証，モデル検査，ネットワークプロトコルの実装など，多くのシステムやライブラリに内在している．

可読性と保守性の観点から，遷移系は可能な限りその仕様に近い形で記述されるのが望ましい．いくつかの既存ツール [8] [9] [11] では，遷移系の実行可能な仕様を記述する方法を提供している．Redex [8] は，S 式で表現された項に対する強力なパターンマッチ機能を提供している．それによって，複雑な内部構造を持つ状態間の遷移規則（transition rule）を，形式的な定義と同程度に簡潔かつ理解しやすい形で記述することを可能にしている．典型的な応用例として，プログラミング言語の非決定的操作意味論に基づく抽象解釈（意味解析手法の一種）の実装 [7] などが挙げられる．

しかしながら，既存のツールにはいくつかの問題があり，より大規模で実用的な遷移系の構築には向いていない．まず，記述の簡略化を優先するが故に，記述された遷移系の実行速度を犠牲にしがちである．例えば，Redex で記述された Racket のマクロシステムの実行モデル [4] は，対象となるベンチマークプログラムのサイズが少しでも大きくなると，実行時間が大幅に増大する．また，既存のツールで記述された遷移規則は，その再利用性や拡張性が限定的である．再利用性が不十分であると，同じ遷移規則を重複して記述する必要が生じる．例えば，実世界のアプリケーション開発においては，わずかに異なるいくつかのバリエーション（例えば，ネットワークプロトコルの複数のバージョン）を含む，複数の遷移系を段階的に開発することがあり得る．あるいは，プログラミング言語の意味解析フレームワークにおいては，推論したい性質に応じて，微妙に性質の異なる複数の遷移系を同時に使用することも考えられる．

一般に，遷移規則の記述において，効率性と柔軟性はトレードオフの関係にある．我々は，実システムに組み込むことを目的とした遷移系を記述するために，拡張性と効率性の両方の利点を備えた遷移系を記述可能な領域特化言語である LwRed（LightWeight REDucer）を提案する．実際にいくつかの性能評価を行うことにより，LwRed が既存のツールに比べ実行速度を大幅に向上していることを示す．一方，LwRed

---

a DSL for Describing Extensible Transition Systems.

Seiji Umatani, 神奈川大学情報学部, Faculty of Informatics, Kanagawa University.

は柔軟性を少し犠牲にして設計されているが，ホスト言語 Racket [6] の表現力を上手く活用することで，実行速度を犠牲にすることなく，実用上，ほとんどの規則を簡潔に記述することを可能にしている．これについても，いくつかの例を用いて詳しく説明する．

本論文の構成は次のとおりである．2 節では，LwRed の設計および特徴について説明する．3 節では，LwRed が Racket の埋め込み DSL としてどのように実装されているかを詳しく説明し，4 節ではその性能を先行研究ツールと比較する．5 節では関連研究について触れ，最後に 6 節で結論を述べる．

2 設計

本節では，LwRed が提供する機能について，純粋型なしラムダ計算 [1] の拡張である PCF 言語 [10] の簡約関係を段階的に構築するプロセスを示すことにより説明する．

2.1 基本機能

遷移規則の集合（簡約関係と呼ぶことにする）は，次の `define-reduction` 構文によって定義される：

```
(define-reduction reduction-name
  [pattern body ... rule-name] ...)
```

各節は Racket の `match`構文とほぼ同じである．実際，*pattern* には任意の `match`パターンを指定できる．各節の末尾にある *rule-name* は，簡約関係を拡張する際，上書き対象を指定するために用いられる（2.2 節参照）．例えば：

```
(define-reduction -->0-red
  [(cons x y) x "car"])
```

は簡約関係`-->0-red` を定義しており，cons セルを状態として受け取り，その car 部を 1 ステップの簡約を行った後の状態として返す．

`define-reduction` によって定義された簡約関係から，$State \to \mathcal{P}(State)$ の遷移関数（transfer function）を取り出すには，次のようにする：

```
(define -->0-gen (reducer-of -->0-red))
(define -->0 (-->0-gen))
(-->0 (cons 1 2)) ; ⇒ (set 1)
```

簡約関係は Racket ユニットシステムにおける 1 ユニットである．`reducer-of` は，簡約関係を表すユニットから（パラメータ化された）遷移関数を取り出す．また上のコードは該当しないが，簡約関係は任意個の識別子をパラメータとして抽象化することも可能である（2.4 節参照）．パラメータ化されていない簡約関係を使う場合には，その遷移関数を無引数で呼び出すことによりインスタンス化しなければならない．

`apply-reduction-relation*`を用いると，複数ステップの遷移を繰り返し，遷移不可能な状態（つまり，どの遷移規則にもマッチしない状態）の集合を得られる．例えば，次の簡約関係`-->1-red` は，入力 cons セルの car 部または cdr 部を非決定的に抽出し，すべてのアトミックな要素の集合 $\{1, 2, 3\}$ を返す．

```
(define-reduction -->1-red
  [(cons x y) x "car"]
  [(cons x y) y "cdr"])
(define -->1 ((reducer-of -->1-red)))
(apply-reduction-relation* -->1
  (cons (cons 1 2) 3)) ; ⇒ (set 1 2 3)
```

ここまでに説明した LwRed の機能を用いて，型なし純粋 λ 計算（LAM）[1] の簡約関係を定義してみる．構文は以下のとおり：

$$e ::= x \mid (\lambda\ (x)\ e) \mid (e\ e)$$

ただし，変数 x は Racket のシンボルで表現することにする．唯一の簡約規則は次のように定義される：

$$((\lambda\ (x)\ e_1)\ e_2) \longrightarrow_\beta e_1[e_2/x]$$

ここで $e_1[e_2/x]$ は代入を表し，Racket による標準的な定義は次のとおり：

```
(define (subst e x e*) ; : e x e → e
  (match e
    [(? symbol? y) (if (eq? x y) e* e)]
    [`(λ (,(? symbol? y)) ,e1)
     (if (eq? x y)
         `(λ (,y) ,e1)
         (let ([y-new (gensym y)])
           `(λ (,y-new)
              ,(subst (subst e1 y y-new)
                      x e*))))]
    [`(,e1 ,e2) `(,(subst e1 x e*)
                  ,(subst e2 x e*))]))
```

ここでは `gensym` を素朴に用いることで，名前の誤っ

た補足を回避している．subst を用い，β 規則は次のように単純に定義できる：

```
(define-reduction -->LAM-red
  [`((λ (,x) ,e1) ,e2)
   (subst e1 x e2) "β"])
```

次に，LAM を拡張して，以下の構文を持つ PCF 言語 [10] に対する簡約関係を定義してみる（簡約規則の数学的定義は省略する）．

$$e ::= \cdots \mid n \mid (\texttt{+}\ e\ e) \mid (\texttt{<=}\ e\ e) \mid (\texttt{if}\ e\ e\ e) \mid (\texttt{fix}\ e)$$

LwRed では，オブジェクト指向言語におけるクラスメンバの継承のように既存の簡約関係にいくつかの規則を追加することにより，新しい簡約関係を定義できる．親の簡約関係から規則を継承するには，define-reduction の#:superキーワードで親の名前を指定する †1．

```
(define-reduction -->PCF0-red
  #:super -->LAM-red
  [`(+ ,(? num? n1) ,(? num? n2))
   (+ n1 n2) "+"]
  [`(<= ,(? num? n1) ,(? num? n2))
   #:with n := (<= n1 n2)
   n "le"]
  [`(if #f ,e2 ,e3) e3 "if-false"]
  [`(if ,e1 ,e2 ,e3)
   #:when (not (false? e1))
   e2 "if-true"]
  [`(fix (λ (,x) ,e))
   #:with y := (gensym 'y)
   `((λ (,x) ,e)
     (λ (,y) ((fix (λ (,x) ,e)) ,y)))
   "fix"])
(define -->PCF0 ((reducer-of -->PCF0-red)))
```

上のいくつかの規則の本体では，#:whenと#:withが使われているが，これらは通常の Racket の構文ではない．「#:when e」はパターンマッチの追加の条件を指定しており，e が#f と評価された場合，当該規則はマッチしないものとみなされる．「#:with x := e」は，Racket の let構文同様，新しい変数束縛を導入する．1 つの規則中に任意個の#:when，#:withを自由な順序で書くことができる．

2.2 文脈パターン

#:with構文と一緒に用いるもう一つの代入演算子 <-を用いると，値の集合へと評価される式をその右辺に書くことができ，左辺に書かれた変数は非決定的にその集合の要素の一つに束縛される．この束縛演算子と非決定的モナド [12] に値の集合を注入する lift を用いることで，例えばある簡約を別に定義された簡約の中に埋め込むことができる．

これを用いることで，評価文脈（evaluation context）の記述が可能となる．例えば，前節の PCF0 では (if (<= 10 8) 3 2) のような項は簡約できないが，上記の束縛演算子を用いると，条件式における評価文脈を PCF0 に追加できる：

```
(define-reduction -->PCF1-red
  [`(if ,e1 ,e2 ,e3)
   #:with e1* <- (lift (-->PCF0 e1))
   `(if ,e1* ,e2 ,e3) "ECif"])
(define -->PCF1 ((reducer-of -->PCF1-red)))
(apply-reduction-relation* -->PCF1
  '(if (<= 10 8) 3 2))
```

e1 の遷移結果を埋め込むことにより，条件節の簡約を実現できている．

しかしながら，(if (if (<= 10 8) 3 2) ...) のような入れ子の評価文脈はまだ表現できていない．PCF0 から継承した簡約関係を，規則の一部に自分自身による遷移を埋め込むよう定義することにより，評価文脈の再帰的構造を表現することができる．そのようにして，さらに条件式以外の評価文脈も加えると，次の定義が得られる †2：

```
(define (val? x)
  (match x
    [(? num?)  #t]
    [(? boolean?) #t]
    [`(λ (,x) ,e) #t]
    [_            #f]))
```

†1 num?は number?の別名である．以降，特に断わることなく同様の別名を使用することにする．

†2 本論文では，「....」はコードの省略の意味で使われ，「...」は構文パターン／テンプレートにおける繰り返しを意味する．

```
(define-reduction -->PCF2-red
  #:super -->PCF0-red
  [`((λ (,x) ,e) ,(? val? v)) ; override
   (subst e x v) "β"]
  [`(if ,e1 ,e2 ,e3) ; override
   #:when (and (val? e1) (not (false? e1)))
   e2 "if-true"]
  [`(,e1 ,e2)
   #:with e1* <- (lift (-->PCF2 e1))
   `(,e1* ,e2) "EC-app1"]
  ....
  [`(if ,e1 ,e2 ,e3)
   #:with e1* <- (lift (-->PCF2 e1))
   `(if ,e1* ,e2 ,e3) "EC-if"]
  [`(fix ,e)
   #:with e* <- (lift (-->PCF2 e))
   `(fix ,e*) "EC-fix"])
(define -->PCF2 ((reducer-of -->PCF2-red)))
```

"β"規則と"if-true"規則を上書きして，値呼びの評価戦略に従うようにしている．

上記の定義では，遷移する可能性のあるすべての部分項に対し，それぞれの評価文脈を指定するという類似した規則が繰り返されており，冗長なコードとなっている．LwRed が Racket 標準の `match`を利用していることを上手く用いることにより，この冗長性を排除することができる．具体的には，以下の説明のように `define-match-expander`を使ってユーザ定義の文脈パターンを定義する．

まず，簡単な例を使って，補助的なパターンエクスパンダ (cxt C p c ...) について説明する．cxt の完全な定義は付録 A に載せている．「c ...」というフォームの列は，パターン中の複数の可能な文脈フォームを意味する．それぞれの文脈フォームは，ホールを表す特別なシンボル□を一つだけ含んでいる．これら複数の文脈フォームは，`match`の単一の `or`パターンに展開される．識別子 C は，このパターンにマッチする文脈を再構築する関数に束縛される．パターン p はホールに埋め込まれるパターンを指定する．例えば，`if`式の条件節に対する文脈だけからなる単純な評価文脈を表現する cxt パターンとその展開後パターンは次の通りである．

```
(cxt EC (? val? v) `(if □ ,p1 ,p2))
⟹ ; expands to
(and `(if ,(? val? v) ,_ ,_)
     (app (match-lambda
            [`(if ,_ ,p1 ,p2)
             (λ (e) `(if ,e ,p1* ,p2*))])
          EC))
```

ホールにフィットする項は，述語 val?によって値だけに限定されている．また，`match`の `and`パターンと app パターンを組み合わせることにより，再構築関数 EC を構成することができる．ここで，p1*と p2*は，ユーザの記述した（複雑かも知れない）パターン p1 と p2 から変数束縛のためだけに導出されたパターンである．

cxt の助けを借りることにより，PCF の値呼び評価文脈 EC を次のように定義できる：

```
(define-match-expander EC
  (syntax-parser
    [(EC p)
     #'(cxt EC p
         `(fix □)
         `(+ ,(? val? v) □)
         `(+ □ ,e1)
         `(<= ,(? val? v) □)
         `(<= □ ,e1)
         `(if □ ,e1 ,e2)
         `(,(? val? v) □)
         `(□ ,e1))]))
```

例えば，次の式：

```
(match '(if (+ 1 2) (* 3 4) 5)
  [(EC e) (EC `(<= ,e ,e))])
```

は (if (<= (+ 1 2) (+ 1 2)) (* 3 4) 5) へと評価される．

最終的に，評価文脈をサポートする PCF3 は，EC を用いて次のように定義することができる：

```
(define-reduction -->PCF3-red
  #:super -->PCF0-red
  [`((λ (,x) ,e) ,(? val? v))
   (subst e x v) "β"]
  [`(if ,e1 ,e2 ,e3)
   #:when (and (val? e1) (not (false? e1)))
   e2 "if-true"]
```

```
[(EC e)
 #:with e* <- (lift (-->PCF3 e))
 (EC e*) "EC"])
```

様々な種類の評価文脈中の遷移を単一の"EC"規則にまとめることにより，コード量を削減できている．さらに，"EC"規則だけを上書きすることで他の評価戦略に切り替えることも簡単にできる．何より，EC を使って書かれたコードは，それに対応する形式的な定義に近く，読みやすくコンパクトである．

2.3 レキシカル・スコープからの逸脱

前節の PCF3 は，LAM 用の subst 関数の定義を参照しているため不完全である．例えば：

```
(->PCF3 '((λ (x) (+ x 1)) 2))
```

は，式 (+ x 1) に対する代入が未定義なため失敗する．

PCF 用の subst 関数を再定義することは必要だが，それに加え，親簡約関係（つまり LAM）中の規則に含まれるすべての subst 関数への参照を置き換える必要がある．subst 関数を参照しているすべての規則を一つ一つ単純に上書きすれば問題は解決するが，それでは継承の利点が損われてしまう．一方，形式的な記述，特にプログラミング言語の意味論の形式的な定義においては，言語を拡張する際，付随する補助定義も暗黙のうちに拡張され，既存の規則中に含まれる同名の定義への参照は自動的にリダイレクトされると仮定されることが多い．今回，新しい定義を自動的かつ暗黙のうちに生成することは我々のツールの目的ではないが，既存の参照を新しく書かれた定義にリダイレクトする機能は，自然な記述のために不可欠であると言える．

そこで，LwRed の define-reduction は，通常の Racket のレキシカル・スコープ規則とは異なるスコープ規則に従うことにする．具体的には，親簡約関係から継承されたすべての規則は，あたかも子簡約関係の中に「字面上」埋め込まれているかのように振る舞うことにする（コード量が増大してしまうため，実装で本当に埋め込んだりはしていない．詳細は 3.4 節参照）．

簡単な例として，以下の定義：

```
(define (f x) `(+ ,x 1))
(define-reduction -->2-red
  [`(f ,x) (f x) "f"])
```

の下，(((reducer-of -->2-red)) '(f a)) は'(+ a 1) となるが，異なるスコープ中（例えば，異なるモジュール中）で：

```
(define (f x) `(* ,x ,x))
(define-reduction -->3-red
  #:super -->2-red)
```

と定義すれば，規則"f"中の変数参照 f は新しい定義を指すことになり，(((reducer-of -->3-red)) '(f a)) は'(* a a) となる．

上記のスコープ規則に従って subst 関数を再定義した PCF の定義は次のとおり：

```
;; subst for PCF
(define (subst e x e*) (match e ...))
(define-reduction -->PCF4-red
  #:super -->PCF3-red
  [(EC e)
   #:with e* <- (lift (-->PCF4 e))
   (EC e*) "EC"])
```

自分自身による簡約（つまり-->PCF4）を参照する必要があるため，"EC"規則だけは上書きする必要があるが，それ以外の全ての規則は継承によって再利用できる．

実際の LwRed では，メタ関数を定義するための専用の define-metafunction フォームを使用して subst 関数を定義することもできる．define-metafunction では，define-reduction と同様に，継承，上書き，レキシカル・スコープ規則からの逸脱を利用でき，親メタ関数との差分だけを記述することで新しい定義を書くことができる．

2.4 パラメータ化された簡約

簡約関係定義を取り巻く文脈が，定義中で同時に使用される識別子の集合を成している場合，2.3 節のレキシカル・スコープからの逸脱を用いることができる．しかしながら，(例えば:=と<-のように）切り替えたい複数の名前が文脈中に同時に見えており，かつ，それぞれが異なる名前を持っている場合，それらを切り替えながら使用することはできない．そのよう

な場合のため，LwRed は，識別子の列 *param* ...でパラメータ化された簡約関係を定義する次の形式を提供している：

```
(define-reduction (reduction-name param ...)
  ...)
```

パラメータ化された簡約関係の例として，非決定的な選択構文 amb を PCF に追加してみる．まず，amb は確率的に選択を行うという規則を含んだベース言語-->PCF5 を定義する：

```
(define select random-ref)
(define-reduction
  (-->PCF5-red --> :=<1> :=<2>)
  [`(<= ,(? num? n1) ,(? num? n2))
   #:with n :=<1> (<= n1 n2)
   n "le"]
  ....
  [`(amb ,es ...)
   #:with e* :=<2> (select es)
   e* "amb"]
  [(EC e)
   #:with e* <- (lift
                 ((--> --> :=<1> :=<2>) e))
   (EC e*) "EC"])
```

これまでの:=の出現は:=<1>と:=<2>に分類され，-->PCF5-red のパラメータとして抽象化されている．es 間の確率的な選択は racket/random ライブラリを用いて実装している．また，"EC"規則を子簡約関係で再利用できるよう，この簡約関係は-->でもパラメータ化されている（後述）．

PCF5 のインスタンスは次のように生成する．ここで，:=は:=<1>と:=<2>の両方に使用される．

```
(define -->PCF5* (reducer-of -->PCF5-red))
(define -->PCF5 (-->PCF5* -->PCF5* := :=))
(define e5 '(+ (amb 1 2 3 4)
               (amb 10 20 30 40)))
(apply-reduction-relation* -->PCF5 e5)
  ; ⇒ {14}
(apply-reduction-relation* -->PCF5 e5)
  ; ⇒ {43}
```

PCF5 の子簡約関係を定義し，amb の意味をランダム選択から非決定的計算へ変えてみる．まず，レキシカル・スコープからの逸脱を利用して select 関数の定義を変更する：

```
  (define select lift)
```

次に，:=<1>は:=のまま，:=<2>を<-でインスタンス化する：

```
(define-reduction
  (-->PCF6-red --> :=<1> :=<2>)
  #:super (-->PCF5-red --> :=<1> :=<2>))
(define -->PCF6* (reducer-of -->PCF6-red))
(define -->PCF6 (-->PCF6* -->PCF6* := <-))
(apply-reduction-relation* -->PCF6 e5)
  ; ⇒ {32 33 34 21 22 ... 31 11 43 12 44}
```

パラメータ-->を子簡約関係-->PCF6*にすることで，"EC"規則を含む全ての親簡約規則をそのまま再利用できている．

2.5 ユニットシステムとの統合

最後に，LwRed の拡張性を向上させるもう一つの機能を紹介する．

レキシカル・スコープからの逸脱により，親簡約関係中からの名前の参照が，子簡約関係の定義を囲むコードで定義されている名前を参照できるようになる．しかし，大規模なソフトウェア開発においては，Racket のユニットシステム [3] などのモジュール・システムを用いて，複数のコンポーネントが互いのスコープの外に分離されてしまうことがよくある．補助定義の数が膨大であり，それらがいくつかの独立したコンポーネントを構成している場合，ユニットシステムは便利である．さらに，2 つのコンポーネントが相互に依存しているような場合，そういった依存関係は Racket ユニットを使わなければ構築できない．また，ユニットはコンパイル時ではなく実行時に置き換えることもできる．例えば，様々な状況や文脈に応じ，単一システム内で同じ言語ファミリーの多くのバリエーションが必要とされるような場合にユニットは有益である．

ユニットシステムの一部として簡約関係を適切に操作するため，LwRed は簡約関係そのものをユニットとして実現している．簡約関係の定義が依存しているシグネチャ（Racket ユニットの importシグネチャに対応）は以下のように指定する：

```
(define-reduction ....
  #:within-signatures [sig ...] ....)
```

簡約関係ユニットは，その遷移関数を意味する名前 reducer だけからなる red^シグネチャをエクスポートする．

例として，まず PCF6 を複数のユニットに分割してみる．組み込み演算子に関連する補助定義の名前は delta^シグネチャに，その他の雑多な補助関数は misc^シグネチャに集約する：

```
(define-signature delta^ (prim? δ))
(define-signature misc^  (val? subst))
```

これらのシグネチャを実装するユニット定義は次のとおり：

```
(define-unit delta1@
  (import) (export delta^)
  (define (prim? op) (member op '(+ <=)))
  (define (δ op vs) ....) ; as before
(define-unit misc1@
  (import delta^) (export misc^)
  (define (val? x) ....) ; as before
  (define (subst e x e*) ....)); as before
```

これらの定義の下，簡約関係は以下のように定義できる．各規則の定義はこれまでと同じである：

```
(define-reduction (-->PCF7-red -->)
  #:within-signatures (delta^ misc^)
  [`(,(? prim? op) ,(? val? vs) ...)
   (δ op vs) "prim"]
  [`((λ (,x) ,e) ,(? val? v))
   (subst e x v) "β"]
  ....)
```

Racket のユニットシステムが提供するユニットを合成・インスタンス化する通常の方法を用いることで，次のように遷移関数を生成できる：

```
(define-values/invoke-unit/infer
  (export red^)
  (link delta1@ misc1@ -->PCF7-red@))
(define -->PCF7 (reducer reducer))
```

最後に，簡約関係と補助関数の間の相互依存の例として，eval プリミティブを PCF7 に追加してみる．delta2@ユニットでは，インポートされている red^ の reducer を介して遷移関数がインスタンス化され，再帰的に呼び出されている．

```
(define-unit delta2@
  (import red^) (export delta^)
  (define (prim? op)
    (member op '(+ <= eval)))
  (define (δ op vs)
    (define --> (reducer reducer))
    (case op
      [(eval)
       (set-first
       (apply-reduction-relation*
         --> (cadar vs)))]
      .... ; as before
    )))
(define-unit misc2@
  (import delta^) (export misc^)
  (define (val? x)
    (match x
      [`(quote ,form) #t]
      .... ; as before
    ))
  (define (subst e x e*) ....)); as before
(define-values/invoke-unit/infer
  (export red^)
  (link delta2@ misc2@ -->PCF7-red@))
```

上記の拡張では，簡約関係-->PCF7-red@を継承や上書きによって修正する必要がまったくなく，合成する 2 つの補助ユニットを置き換えるだけで済んでいる．

3 実装

本節では，LwRed の実装について説明する．分かりやすさのため，本節で紹介するソースコードは，実際の実装を若干簡略化したものである．

3.1 非決定的モナド

2.2 節で説明したとおり，define-reduction の規則本体では非決定的モナドを使用することができる．非決定的モナドは，Racket の集合データ型 set を使って，次のように実装されている．ここで，pure と lift はそれぞれ，単一の値または値の集合をモナドに注入する操作である．

```
(define (pure x) (set x))
(define (lift xs) xs)
```

```
  (define (bind r k)
    (apply set-union (set) (set-map r k)))
```

LwRed ユーザが規則本体を記述するのに用いる do記法は，単純な Racket マクロとして定義されている：

```
(define := (gensym)) (define <- (gensym))
(define (with op r k)
  (cond [(eq? op :=) (k r)]
        [(eq? op <-) (bind r k)]))
(define-syntax (do stx)
  (syntax-case stx ()
    [(do s) #'s]
    [(do #:with pat op e s1 s ...)
     #'(with op e
         (match-lambda
           [pat (do s1 s ...)]))]
    [(do #:when t s ...)
     #'(if t (do s ...) (set))]
    [(do s1 s ...)
     #'(begin s1 (do s ...))]))
```

#:with構文の op が:=の場合は letとして機能し，<-の場合は bind として機能する．これらの演算子は，実行時に交換可能なよう，ファーストクラスなシンボルとして定義されている．

3.2 遷移関数

簡約操作の第一近似として，まずは継承とユニットを考慮しない実装について考える．簡単のため，常にパラメータ化された簡約関係を定義する define-reduction について説明する．パラメータを指定しないものは，パラメータがゼロ個の特殊バージョンに過ぎず，define-reduction マクロの追加の展開規則で簡単に扱うことができる．

```
(define-syntax (define-reduction stx)
  (syntax-case stx ()
    [(_ (rid param ...)
        [pat b ... name] ...)
     #`(define ((rid param ...) s)
         #,(make-reducer-body #'s
            (mk-hash
             (syntax->datum #'(name ...))
             (syntax->list
              #'([pat b ...] ...)))))]))
```

「param ...」によりパラメータ化された簡約関係 rid は，「parm ...」をパラメータとして受け取る同名の関数として実現される．この関数は「状態 s を受け取り，可能な 1 ステップの簡約により生成される次状態の集合を返す遷移関数」を返す．

make-reducer-body 関数は，遷移関数本体のコードを生成する．補助関数 mk-hash は，キーと値のリストを受け取り，ハッシュテーブルを生成する．遷移規則 [pat b ... name] のリストから mk-hash を使用して構築された規則集合を表すハッシュテーブルが，make-reducer-body に渡される．次の make-reducer-body では規則名（テーブルのキー）は使用されないが，3.3 節の継承において重要な役割を果たす．

```
(define-for-syntax
  (make-reducer-body s cmap)
  (for/fold ([body #'(set)])
            ([c (in-hash-values cmap)])
    (syntax-case c ()
      [(pat b ... bn)
       #`(let ([nexts #,body])
           (match #,s
             [pat (set-union
                   nexts
                   (do b ... (pure bn)))]
             [_ nexts]))])))
```

マッチしたすべての節を非決定的に実行するため，make-reducer-body マクロは define-reduction で並列に書かれたパターンマッチを，入れ子の match式へ変換する．すべての可能な遷移は，nexts に蓄積された内側の結果と外側のマッチ結果の set-union により集められる．外側の 1 ステップの遷移は，非決定的モナド (do b ... (pure bn)) として実行される．

最後に，遷移関数を使用して複数の遷移を実行する apply-reduction-relation*関数は，標準的なワークリストアルゴリズムを使用して実装されている：

```
(define (apply-reduction-relation* --> s)
  (let ([all-states (mutable-set)]
        [normal-forms (mutable-set)]
        [worklist (make-queue)])
    (define (loop)
```

```
    (unless (or (queue-empty? worklist))
      (let* ([s (dequeue! worklist)]
             [nexts (--> s)])
        (if (set-empty? nexts)
            (set-add! normal-forms s)
            (for ([next (in-set nexts)]
                  #:when (not
                          (set-member?
                           all-states
                           next)))
              (set-add! all-states next)
              (enqueue! worklist next))))
      (loop)))
  (set-add! all-states s)
  (enqueue! worklist s)
  (loop)
  normal-forms))
```

3.3 継承

継承による拡張をサポートするには，ある簡約関係の define-reduction をマクロ展開する際，その親簡約関係（および再帰的にさらにその祖先）の define-reduction により指定された様々なプロパティを維持する必要がある．現在の実装では，定義：

```
(define-reduction (rid param ...)
  #:super (sup-id sup-arg ...)
  [pat b ... name] ...)
```

に対応し，次の rdesc 構造体を構築する．なお簡単化のため，親簡約関係が存在しない場合には，#:super (#f) と書くものとする．

```
(struct rdesc (entity params
               sup-id sup-args cmap))
```

entity フィールドには，対応する遷移関数（実行時に使用されるファーストクラスのオブジェクト）が格納されている実行時変数名を参照する構文オブジェクトが含まれる．params，sup-id，sup-args は，それぞれ define-reduction 構文中の対応する構文オブジェクトである．また，前節で mk-hash を使用して構築されたのと同じハッシュテーブルがマクロ展開時に構築され，cmap に格納される．

継承をサポートするよう更新された define-reduction の定義を図 1 に示す．主に構文

```
(define-syntax (define-reduction stx)
  (syntax-case stx ()
    [(_ (rid param ...)
        #:super (sup-rid arg ...)
        [pat b ... name] ...)
     (with-syntax
       ([eid (fmt-id #'rid "~a@" #'rid)])
       #`(begin
           (define-syntax rid
             (rdesc #'eid #'(param ...)
              #'sup-rid #'(arg ...)
              (make-hash
               (list (cons name
                           #'((... ...)
                              [pat b ...]))
                     ...))))
           (define-syntax (#%reducer stx)
             #`(λ (param ...)
                 (λ (s)
                   #,(make-reducer-body
                      #'s
                      (syntax-local-value #'rid)
                      #f '()))))
           (define eid (#%reducer))))]))
```

図 1　継承をサポートする define-reduction

変換子（つまりマクロ）を定義するために用いられる Racket の define-syntaxは，構文変換子以外の任意の値を束縛するためにも使用できる．更新後の define-reduction により生成されるコードでは，定義している簡約関係の rdesc 構造体を格納するために名前 rid を使用している．生成された遷移関数を格納するために使用される実行時変数の名前（eid）は，rid の末尾に@を付加することでつくられるフレッシュな識別子である †3．

make-reducer-body が生成するコードは，param 達のスコープ内部に置かなければならない．また，make-reducer-body は，rdesc が rid に格納された**後に**コードを生成しなければならない．仮に，eid を定義するコードを：

```
(define ((eid param ....) s)
  .... #,(make-reducer-body ....) ....)
```

のように生成すると，make-reducer-body の

†3　末尾の@は，慣例的にユニット名を表す．後述の拡張により，簡約県警の実体は単なる関数ではなくユニットとなる．

呼出しが早すぎることになる．この問題を解決するため，make-reducer-body の呼出しを「define-reduction のマクロ展開時」から「define-reduction によって生成されるコードのマクロ展開時」まで遅延させる．eid とは別に定義されている#%reducer は，そのためのマクロである．

rid から遷移関数を取り出す reducer-of の定義は次の通り：

```
(define-syntax (reducer-of stx)
  (syntax-case stx ()
    [(_ rid)
     (rdesc-entity
      (syntax-local-value #'rid))]))
```

syntax-local-value は，define-syntaxによって格納されているデータへマクロ展開中にアクセスするための Racket の関数であり，reducer-of の呼出しが eid への参照に置き換えられる．

継承をサポートするよう更新された make-reducer-body の定義を図 2 に示す．更新後の定義では，祖先簡約関係の rdesc 達を参照することにより，それらに含まれる簡約規則も再帰的に生成する．引数 maybe-args は，親簡約関係に対し再帰的に呼び出された場合にはその実引数を保持し，define-reduction から直接呼び出された場合には #f を保持している．引数 subrules には，祖先簡約関係の処理中にコードの生成を抑制するため，子孫簡約関係中に存在する簡約規則の名前の集合が含まれている．親簡約関係に対する make-reducer-body の再帰呼出しによって生成された本体コード sup-body に対し，子簡約関係の規則のためのコードが追加される．本体コード中のすべてのパラメータ参照 param が，Racket の make-rename-transformer によって生成されるリネームマクロによって，子簡約関係から渡された名前 arg への参照に置き換えられていることに注意して欲しい．

3.4 レキシカル・スコープの調整

2.3 節で説明したレキシカル・スコープからの逸脱は，構文オブジェクト中のレキシカル文脈データを操作するための Racket 関数 syntax->datum，

```
(define-for-syntax (make-reducer-body
                    s rd maybe-args subrules)
  (match-define
    (rdesc _ params sup-id sup-args cmap) rd)
  (define sup-body
    (if (syntax->datum sup-id)
        (make-reducer-body
         s (syntax-local-value sup-id)
         sup-args
         (append subrules (hash-keys cmap)))
        #'(set)))
  (let ([args (or maybe-args #'())]
        [params (if maybe-args params #'())])
    (with-syntax ([(param ...) params]
                  [(arg ...)   args])
      #`(let-syntax
          ([param (make-rename-transformer #'arg)]
           ...)
          #,(for/fold ([body sup-body])
              ([(nam c) (in-hash cmap)]
               #:unless (member nam subrules))
            (syntax-case c ()
              [(pat b ... bn)
               #`(let ([nexts #,body])
                   (match #,s
                     [pat (set-union nexts
                            (do b ... (pure bn)))]
                     [_ nexts]))]))))))
```

図 2 make-reducer-body の定義

datum->syntax を用いて実現する．syntax->datum は構文オブジェクトから純粋なコード（S 式）を抽出し，datum->syntax はある構文オブジェクトのレキシカル文脈データを S 式に付加することにより，新たな構文オブジェクトを生成する．したがって，以下の rescope 関数は，stx1 のレキシカル文脈データを stx2 に移植する：

```
(define-for-syntax (rescope stx1 stx2)
  (datum->syntax stx1
                 (syntax->datum stx2)))
```

これにより，make-reducer-body によって生成されるコードを任意のレキシカル文脈中に埋め込むことができる．具体的には，make-reducer-body を次のように修正する．生成されるコード中のすべての構文オブジェクトは，追加の引数 ctx によって渡されるレキシカル文脈データによって rescope される．

```
(define-for-syntax
  (make-reducer-body
```

```
 ctx s rd maybe-args subrules)
....
(with-syntax ([(param ...)
               (rescope ctx params)]
              [(arg ...)
               (rescope ctx args)])
  ....
  (syntax-case (rescope ctx c) ()
    [(pat b ... bn)
     ....])))
```

修正後の define-reduction を以下に示す．簡約関係名 rid を make-reducer-body の引数 ctx として渡している．この rid は，さらに make-reducer-body のすべての再帰呼び出しにも渡されており，define-reduction を囲む文脈中のあらゆる名前が，祖先簡約関係中で定義された簡約規則からも見えるようになる．

```
(define-syntax (define-reduction stx)
 (syntax-case stx ()
  [(_ (rid param ...) ....)
   (with-syntax ....
    #`(begin
        ....
        (define-syntax (#%reducer stx)
         #`(λ (param ...)
             (λ (s)
               #,(make-reducer-body
                  #'rid #'s
                  (syntax-local-value #'rid)
                  #f '()))))
        ....))]))
```

3.5 Unitization

2.5 節で述べたように，define-reduction は実際には遷移関数を直接生成せず，次の red^シグネチャをエクスポートするユニット定義を生成する：

```
(define-signature red^ (reducer))
```

reducer がその遷移関数を表している．また，簡約関係を実装するユニット中の最後の式は，その遷移関数を参照する式となっている．したがって，reducer-of は，invoke-unitを使って実体ユニットをインスタンス化するコードとして次のように定義できる：

```
(define-syntax (define-reduction stx)
 (syntax-case stx ()
  [(_ (rid param ...)
      #:super (sup-rid arg ...)
      #:within-signatures (sig ...)
      [pat b ... name] ...)
   (with-syntax
    ([eid (format-id #'rid "~a@" #'rid)])
    #`(begin
       (define-syntax rid ....) ; as before
       (define-unit eid
        (import sig ...) (export red^)
        (define-signature M^
          ((define-values (reducer) (#%reducer))
           (define-syntaxes (#%reducer)
             (λ (stx) ....))))
        (define-unit M@ (import) (export M^))
        (define reducer
          (invoke-unit
           (compound-unit (import) (export)
            (link (([m : M^]) M@)
                  (() (unit (import M^) (export)
                        reducer) m)))))
        reducer)))]))
```

図 3 define-reduction の定義（最終版）

```
(define-syntax (reducer-of stx)
  (syntax-case stx ()
    [(_ rid)
     #`(invoke-unit
        #,(rdesc-entity
           (syntax-local-value #'rid)))]))
```

define-reduction は，define-unitを用いてユニットを定義するコードを生成する．生成されるコード中において，define-unitは#:within-signaturesで指定されたインポートされているシグネチャのレキシカル・スコープ内部に配置されなければならない．一方，これまで説明した実装において重要な役割を果たしている#%reducer は，マクロとして定義されているため Racket のユニットシステムで若干問題を生じる．すなわち，ユニットシステムでは，マクロ定義（define-syntax）を define-unit本体に書くことができず，代わりに，define-signatures 中に記述せざるを得なくなる．

上記の問題を解決すると同時に，前節で述べた LwRed の設計方針を満たすため，define-reduction

を図 3 に示すように修正し，入れ子になった define-unitを生成するようにする．rid の定義はこれまでと同じである．簡約関係の実体ユニットである eid の定義は，#:within-signaturesで指定されたすべてのシグネチャをインポートし，red^をエクスポートする．また，生成コードには，ローカルシグネチャM^とローカルユニット M@の定義も含まれている．（M@ではなく）M^に#%reducer の定義が含まれており，M@は M^と eid の間の橋渡しの役割を果たしている．M^（およびその中に含まれる#%reducer）がシグネチャ「sig ...」からインポートされている名前のスコープの内側に置かれていることに注意して欲しい．eid ユニットの遷移関係は，M@と匿名ユニットを combine-unit によって組み合わせることで作成されたユニットを呼び出すことにより生成される．後者の匿名ユニットが，M@からインポートした遷移関係をそのまま返している．

4 評価

本節では，LwRed と Redex [8] の実行時間を比較する．測定には 2 つの異なるプログラムセットを用いる．まず，2 節の PCF 言語で書かれたいくつかの小さなプログラムを，両ツールで実装された簡約関係の上でそれぞれ実行し，その実行時間を測定する．次に，より現実的な規模の遷移系として，Racket の衛生的かつ手続き的なマクロシステムの実行モデル [4] を両ツール上に実装し，その成果物 [5] に含まれているベンチマークを用いて実行時間を測定する．どちらの場合においても，Redex 上の実装は LwRed 上の実装と本質的に同じように書かれている．

測定に用いた環境は次のとおり：

- ハードウェア：MacBook Pro (16-inch, M1 Max, 2021)，64GB メモリ
- ソフトウェア：macOS 12.6, Racket v8.6 [cs]

ベンチマークの実行時間の測定には，Racket 標準ライブラリの time-apply 関数を使用する．

4.1 PCF マイクロベンチマーク

測定の準備として，let構文と letrec構文を 2 節の PCF に変換する別の遷移系を定義する．そして，それら 2 つシンタックスシュガーを用いて，単純な再帰算術関数からなる 3 つの小さなベンチマーク・プログラム（add，mult，ack）を用意する：

```
(let ([add (λ (x) (+ (car x) (cdr x)))])
  (let ([curry (λ (f) (λ (x) (λ (y)
                         (f (cons x y)))))])
    (((curry add) 5) 7)))
(letrec
  ([mult
    (λ (p)
      (if (= (car p) 0)
          0
          (+ (cdr p)
             (mult (cons (- (car p) 1)
                         (cdr p))))))])
  (mult (cons 4 7)))
(letrec
  ([ack
    (λ (m)
      (λ (n)
        (if (= m 0)
            (+ n 1)
            (if (= n 0)
                ((ack (- m 1)) 1)
                ((ack (- m 1))
                 ((ack m) (- n 1)))))))])
  ((ack m) n))
```

図 4 に実行時間の測定結果を示す．図中，「lw」と「rx」はそれぞれ LwRed と Redex の実装を意味する．Redex で生成された遷移関数は，デフォルトでメモ化が有効になっている．まったくメモ化しない LwRed と公平に比較するため，Redex についてはメモ化機能をオフにした実行についても測定を行った．図 4 の「rx-no-cache」がその結果である．

棒グラフは，5 つのプログラムの実行時間（ミリ秒）を示している（ack は 3 つの異なるパラメータの組で測定を行った）．図には，letと letrecの前処理を含む簡約全体の間に辿った状態の数を示す折れ線グラフも含めている．apply-reduction-relation*を少し修正することで状態数を数えている †4．

†4 同じ PCF 実行モデルを実装しているため，rx と rx-no-cache の状態数は lw と同じである．

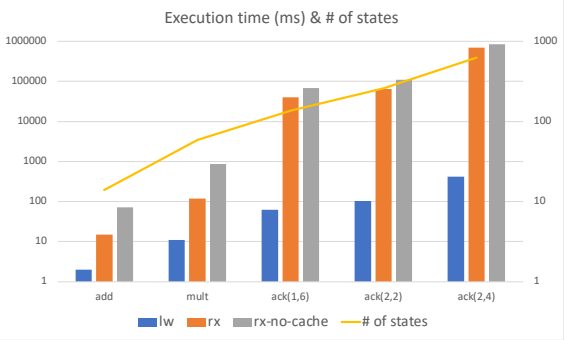

図 4　PCF プログラムの実行結果（全体）

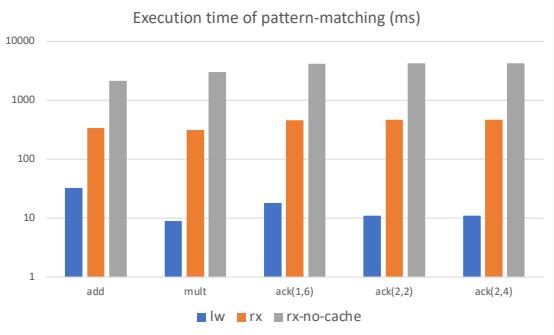

図 5　PCF プログラムの実行結果（パターンマッチのみ）

これらの測定結果から，`lw` は平均して `rx` より 584 倍，`rx-no-cache` より平均して 850 倍高速であることがわかる．状態数と実行時間はすべての実装において比例している．この結果は，PCF が単純な言語実行モデルであり，各状態で実行される操作の量が一定であることを示唆している．

`lw` については，Racket の `memoize` ライブラリを使ってすべての補助関数をメモ化することも試みたが，実行時間の改善は見られなかった．このことから，`rx` が `rx-no-cache` よりも高速である理由は，Redex が項のパターンマッチ処理をメモ化しているからだと推測できる．一方，`lw` は Racket の `match`をそのまま利用しているため，その実装にメモ化を追加するのは容易ではない．

上記の観察に基づいてより詳細に調査するため，パターンマッチングのみを行うプログラムを用意し，それ以外はまったく同じ設定で実行速度を測定した．その結果（図 5）は，メモ化を行わないにもかかわらず，LwRed のパターンマッチが `rx` より平均して 43 倍，`rx-no-cache` より平均して 386 倍高速であることを示している．

パターンマッチングで大きな性能差があるが，図 4 の実行時間全体と比べると，その差はそれほど顕著ではなく，他の要因によっても高速化されていることがわかる．Redex の実行時間には，例えば到達可能な各状態に対するアサーションチェックのような，現在の LwRed がサポートしていない機能を実現するのに必要なオーバヘッドが含まれているものと思われる．

4.2　スコープ集合モデル

より現実的なベンチマーク・プログラムとして，衛生的かつ手続き的なマクロをサポートするインタプリタとして表現された，Racket の形式的な実行モデル（スコープ集合モデルと呼ぶ）を用いることにする．スコープ集合モデルの実行可能コードは，ビッグステップ操作意味論に基づき Redex の `define-metafunction` の集合として記述されている [5]．今回の性能比較に用いるため，それを同等のスモールステップ簡約規則に系統的に書き換えている．

スコープ集合モデルには，前節の PCF にはないいくつかの特徴がある：

1. PCF と比較し，より広範な遷移系を形成している．具体的には，スコープ集合モデル全体で 56 個の簡約規則があるのに対し，PCF には 13 個しかない．
2. スコープ集合モデルにおける各状態は，PCF のような単純な項ではなく，ストア，環境，継続といったより複雑なデータ構造を含んでいる．
3. 2.5 節の最後の例のように，スコープ集合モデルは評価とマクロ展開の間の相互再帰的な遷移関係から成る．各状態はその一部として継続を含み，相互再帰に伴い継続が伸び続けるため，スコープ集合モデルにおける全状態空間はかなりの規模となる．

前節と同様，スコープ集合モデルの 3 つの異なる実装（`lw`，`rx`，`rx-no-cache`）を用意し，[5] に含まれる 22 個のサンプル Racket プログラムの実行時間を測定する．図 6 にその結果を示す．結論から述べる

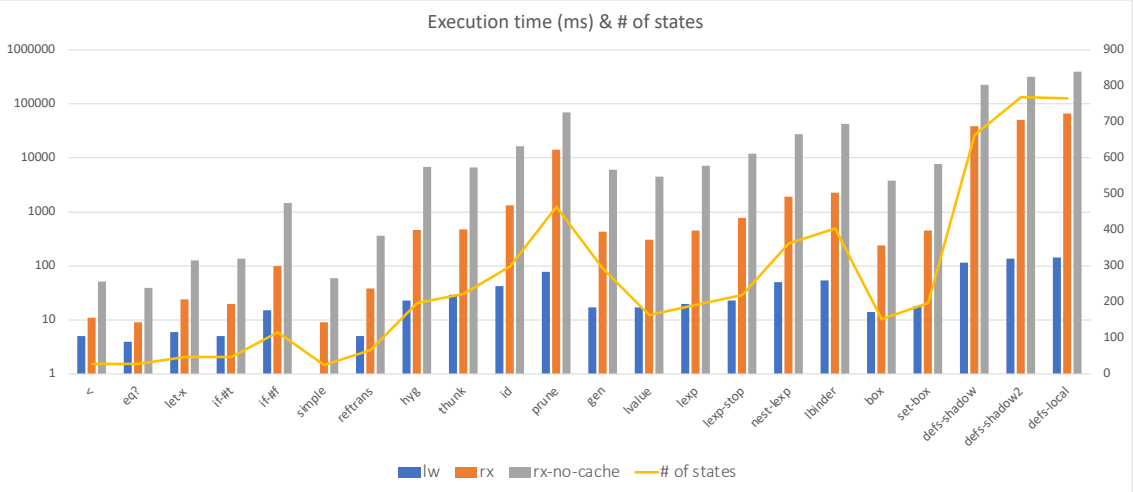

図 6　スコープ集合モデルの実行結果

と，`lw` は `rx` より平均して 76 倍，`rx-no-cache` より平均して 580 倍高速となっている．また図からわかるとおり，実行時間はすべての実装（特に `rx`）において状態数に対し指数的な傾向を示している．この結果は，各状態遷移で実行される操作にかかる時間が定数オーダではないことを意味する．Redex では，遷移中の複雑な操作は通常 `define-metafunction` によって定義された別の補助関数として記述される．この構文は簡約関係と同様に Redex のパターンマッチ機能の上に構築されているため，前節のパターンマッチ速度の測定結果からも分かるように，補助関数の実行に実行性能のボトルネックが生じている可能性がある．一方，LwRed では，Racket ライブラリ中の効率的なデータ型や関数を自由に使用することが可能である．生の Racket コードを直接 Redex に埋め込むことも原理的には可能であるが，Redex と Racket の記法が混在することで可読性が低下し，推奨されていない．

5　関連研究

Redex [8] は，簡約関係だけでなく，型システム（型判断規則），遷移状態に対するアサーション，単体テストなど，意味論に関する様々な側面を表現するための機能を備えたプログラミング言語研究支援 DSL である．Redex の表現力は，主に S 式に対する強力なパターンマッチ機能によるものである．しかし，いくつかの他の構成要素（例えば，評価文脈）に関連するアドホックな記法など，いくつかの制限によりその柔軟性を損なっている．また，補助定義を記述する際，通常の Racket 言語であれば当然のように記述できる高階関数は許されない．

Redex は言語拡張のための 2 つの機能 `define-extended-language` と `extend-reduction-relation` を提供している．しかしながら，これらは LwRed の言語拡張の機能ほど柔軟ではなく，特に，親遷移規則中の名前参照を別の名前へリダイレクトすることはできない．Redex の拡張である [9] は何種類かの定義名への動的束縛を導入している．しかしながら，上記の問題に対処するための能力は限定的であり，実際，スコープ集合モデルを実装するには簡約関係の定義のいくつかの部分でコードの複製が必要であることがわかっている．

Ott [11] もまた，簡約関係を含むプログラミング言語に関する様々な研究を支援する機能を備えた DSL である．形式的に定義された遷移規則を他の言語上の実行可能なコードに変換することができる．しかしながら，Redex や LwRed ほど柔軟なパターンマッチを記述できず，大規模な遷移規則集合に対する拡張性も特にサポートしていない．

6 まとめと今後の課題

本論文では，実用的なシステムに組み込まれた遷移システムを記述するための領域特化言語である LwRed を提案した．LwRed は，レキシカル・スコープから逸脱した規則本体の記述，Racket ユニットシステムとの統合などの特徴を備えており，大規模な遷移系のモジュール化され拡張可能な記述を可能としている．また，LwRed の設計はホスト言語である Racket との互換性が高く，簡易かつ効率的な遷移規則を実現することが可能である．

Racket のパターンマッチやパターンエクスパンダを使用して簡潔な状態パターンを記述するには，Racket のマクロにある程度精通している必要がある．例えば，本論文で示したものよりさらに複雑な文脈（例えば，プロセス計算で使用されるような複数のホールを含む文脈）であっても，パターンエクスパンダとして実現することは可能である．とはいえ，このような複雑な形状の文脈をより直接的に表現するための上位 API を LwRed に追加することは，様々な遷移系を実装する労力を軽減するのに役立つものと思われ，今後の課題の一つと考えている．

謝辞 本研究は JSPS 科研費 JP20K11757 の助成を受けたものである．

参 考 文 献

[1] Barendregt, H.: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland Linguistic Series, North-Holland, 1984.
[2] Culpepper, R. and Felleisen, M.: Fortifying Macros, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, 2010, pp. 235–246.
[3] Findler, R. B. and Flatt, M.: Modular Object-Oriented Programming with Units and Mixins, *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, 1998, pp. 94–104.
[4] Flatt, M.: Binding as Sets of Scopes, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, 2016, pp. 705–717.
[5] Flatt, M.: Binding as Sets of Scopes – Notes on a new model of macro expansion for Racket, 2016.
[6] Flatt, M. and PLT: Reference: Racket, Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `https://racket-lang.org/tr1/`.
[7] Horn, D. V.: An Introduction to Redex with Abstracting Abstract Machines (v0.6), 2014.
[8] Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B.: Run Your Research: On the Effectiveness of Lightweight Mechanization, *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, 2012, pp. 285–296.
[9] Moy, C.: Redex Parameters, 2020.
[10] Plotkin, G.: LCF considered as a programming language, *Theoretical Computer Science*, Vol. 5, No. 3(1977), pp. 223–255.
[11] Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strniša, R.: Ott: Effective Tool Support for the Working Semanticist, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, 2007, pp. 1–12.
[12] Wadler, P.: The Essence of Functional Programming, *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, 1992, pp. 1–14.

A 付録: cxt パターンエクスパンダの定義

可読性のため，以下の cxt の定義では，syntax/parse ライブラリ [2] を使用している．

```
(begin-for-syntax
  (define (mask-unquote stx)
    (syntax-parse stx
      #:datum-literals [unquote ?]
      [(unquote (? p x)) #'(unquote (? p))]
      [(unquote e) #'(unquote _)]
      [(s ...) (stx-map mask-unquote #'(s ...))]
      [s #'s]))
  (define (pick-id stx)
    (syntax-parse stx
      #:datum-literals [unquote ?]
      [(unquote (? p x)) #'(unquote x)]
      [(s ...) (stx-map pick-id #'(s ...))]
      [s #'s]))
  (define-syntax-class hole
    #:attributes (nam pat upat body)
    (pattern
      (~datum □)
      #:with nam  #'□
      #:with pat  #'_
      #:attr upat (λ (pat) #`,#,pat)
      #:with body #',nam)
    (pattern
      (e1 ... h:hole e2 ...)
      #:with (u1 ...)
             (stx-map mask-unquote #'(e1 ...))
      #:with (u2 ...)
             (stx-map mask-unquote #'(e2 ...))
      #:with (b1 ...)
             (stx-map pick-id #'(e1 ...))
      #:with (b2 ...)
             (stx-map pick-id #'(e2 ...))
      #:with nam  #'h.nam
      #:with pat  #'(e1 ... h.pat e2 ...)

      #:attr upat
             (λ (pat)
               #`(u1 ...
                  #,((attribute h.upat) pat)
                  u2 ...))
      #:with body #'(b1 ... h.body b2 ...))))
(define-match-expander cxt
  (syntax-parser
    [(_ C:id pat h:hole)
     #`(and #,((attribute h.upat) #'pat)
            (app (match-lambda
                   [h.pat (λ (h.nam) h.body)])
                 C))]
    [(_ C:id pat h1:hole h2:hole ...)
     #'(or (cxt C pat h1)
           (cxt C pat h2) ...)]))
```