遷移系記述システム LwRed の拡張パターンを使った関数型言語の評価文脈の記述

佐藤 美帆

2023 年 1 月 30 日

1 はじめに

本研究は関数型言語の意味論で広く用いられる評価文脈を簡潔に表現できるパターンを考察することが目的であり、今回は一例として Redex の構文定義の例 [1] に出る Box モダリティを持つ単純型λ微分積分である BoxyL 言語のコンテキストを LwRed 上で実際に表現する。

BoxyL 言語の定義を以下に示す。

```
(require redex/reduction-semantics)
(define-language BoxyL
 (e ::= x natural (+ e e) (cons e e)
 (car e) (cdr e) (λ (x : A) e) (e e)
          (box e) (let ((box y) e) e))
 (A B ::= Nat (A × B) (A → B) (□ A))
 (x y ::= variable-not-otherwise-mentioned)
 (v ::= natural (box e) (λ (x : A) e)
          (cons v v))

  #:binding-forms
   (λ (x : A) e #:refers-to x)
   (let ((box y) e_1) e_2 #:refers-to y))
```

本論文の残りの部分は以下のように構成されている。2 章では LwRed について述べる。3 章では LwRed の文脈パターン、コンテキストについて述べる。4 章では実際に BoxyL を LwRed 上に書いたコードについて説明する。

2 LwRed

プログラミング言語処理系など、さまざまなソフトウェアシステムの中核をなすのが「遷移系」であり、遷移システムにおいて拡張性や実行効率は重要な関心事である。遷移系は様々なコンピュータシステムの簡略化されたモデルを記述するために広く用いられており、可読性と保守性の観点からできるだけ仕様に近い形で記述される必要がある。

既存のいくつかのツールでは実行可能な遷移システムしようの記述方法を提供しているが、いくつかの問題がある。１つ目は記述の簡略化を優先するため、記述された遷移システムの実行速度が犠牲になっている。

２つ目は、既存のツールで記述された遷移ルールの再利用性・拡張性に問題があること。再利用性が不十分な場合、遷移ルールが重複し、現実のアプリケーションに考えられる微妙に異なるバリエーションで複数のシステムを段階的に開発する場合など、より大規模で実用的な遷移システムの構築には不向きである。

そこで、実世界のシステムに埋め込まれた遷移システムを記述するために、拡張性と効率性の両方の利点を持つドメイン固有の遷移システム記述言語 LwRed を使用することで既存のツールを比べ実行速度を大幅に向上させることが可能になった。LwRed は既存のツールに比べ柔軟性を犠牲にしているが、ホスト言語 Racket の表現をうまく利用することで、実行速度を犠牲にすることなく、ほとんどのルールを簡単に記述することが出来る。

3 文脈パターン

LwRed の設計原理と特徴について、遷移規則は (define-reduction) 構文で定義される。また Racket の match 構文とほとんど同じで、実際 pattern で任意のパターンの match を指定できる。

以下のような例を考える。

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

pattern body で任意のパターンマッチを指定し、rule-name は拡張する際に対象を指定するために使われる。

LwRED ではオブジェクト指向言語のクラスメンバの継承のように、既存の reduction にいくつかのルールを追加することで新しい reduction 関係を定義することが出来る。ルールの中には#:when や#:with という通常の Racket の構文にはないものがある。これらは 1 つのルールの中で任意の数を順序で記述することができる。

#:with 構文で使われるもう一つの代入演算子は右辺に値の集合として評価される式を書くことができ、左辺の変数はそのうちの 1 つに非決定的に束縛される。

そして、LwRed では difine-match-expander を使ってユーザが定義するコンテキストパターンを定義することが出来る。

```
(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))
```

c はこのパターンにマッチする文脈を再構成する関数に束縛され、p は穴に収まるパターンを指定する。

cxt の定義を以下に示す

```
(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) ...)]))
```

4 実装

上記で定義されたコンテキストを利用し、BoxyL 用の EC を定義したコードを以下に示す

```
(define-match-expander EC
  (syntax-parser
    [(EC e)
     #'(cxt EC e
            `(□ e)
            `(v □)
            `(cons v ... □ e ...)
            `(+ v ... □ e ...)
            `(car □)
            `(cdr □)
            `(let ((box x) □) e))]))

(define-reduction BoxyL/EC
  [(EC e)
   #:with e* <-(lift (-->BoxyL e))
   (EC e*)"EC"])
```

5 まとめ

本研究では Box モダリティを持つ単純型λ微分積分である BoxyL 言語のコンテキストを LwRed 上で実際に書くことを行った。

LwRed で使われていたコンテキストを使って BoxyL 用の EC を定義することが出来た。

参考文献

[1] William J. Bowman,Experimenting with Languages in Redex.
https://www.williamjbowman.com/doc/experimenting-with-redex/index.html