Context-Free Grammars Including Left Recursion using Recursive miniKanren

Hirotaka Niitsuma

Graduate School of Natural Science and Technology, Okayama University
Okayama, 700-0530 Japan
niitsuma@de.cs.okayama-u.ac.jp

Abstract. recursive miniKanren is logic programming language which can deal infinite recursive data structure and a subset of the Scheme language. We define a pattern match macro which can use the same syntax of the match macro of the Scheme language using recursive miniKanren. The macro enables to write searching sub-list with a given pattern by only few line code. Using this property, we introduce techniques writing context-free grammar with our match macro. Unlike other specific paraphrasing tools, our technique can combine logical relations of miniKanren with a context-free grammar. We show the logical relations resolves the ambiguity of a grammar.

1 Introduction

miniKanren[5] is one of the major relational logic programming languages. Prolog is also a well-known relational programming language. A typical Prolog implementation consists of thousands of lines of C code. The main advantage of miniKanren is that it consists of less than 1000 lines of Scheme code. With this advantage, miniKanren can be easily modified. Therefore, many dialects of miniKanren are made[3, 4, 7].

Since miniKanren is not only a subset of Prolog but also a subset of the Scheme language, miniKanren has properties like Lisp. For example, miniKanren can use pattern match macros of the Scheme language[8]. However, the pattern match macro of miniKanren cannot use a pattern including "..." which denotes sequence. This research introduces the extension of the match macro which enables using a pattern including "...". Let us call the extended match macro as match^{ee} .

Due to the simplicity of miniKanren, miniKanren can not handle structures containing infinite recursion. For example, miniKanren can not handle Scheme code like `#0 = ( #0# 3 )`. Such recursions appear in many applications, e.g. type inference of delayed stream, Fourier analysis of signal processing, and context-free grammar with left-recursion. This research shows we can handle such infinite recursions by modifying some functions of miniKanren. Let us call the modified miniKanren and the original miniKanren *recursive miniKanren* [1] and *normal miniKanren* [2], respectively.

[1] `https://github.com/niitsuma/Racket-miniKanren/blob/recursive2/`
[2] `https://github.com/miniKanren/Racket-miniKanren`

2 Notation

The following abbreviation is used in this paper. The symbol $\equiv$ is == in our program code. The symbol $\triangleright$ is ==> in our program code. Table 1 shows the list of the abbreviations.

Table 1. Notation

abbreviation	program code
$\equiv$	`==`
$\triangleright$	`==>`
run*	`run*`
$_0$ $_1$ $_2$ ...	`_.0 _.1 _.2` ...
match^e	`matche`
match^{ee}	`matchee`

3 recursive miniKanren

Consider the following execution result of the normal miniKanren:

(run* (q) ($\equiv$ q ‘(,q 3)))
$>$ ()

where $>$ denotes execution result. The normal miniKanren estimates that there is no result satisfying the recursive relation:

$q = (q\ \ 3)$

However, the above recursive relation represents the following circular list.

$q =$ #0 $=$ (#0# 3)

Expanding this recursive relation gives

$q = (((((\ldots)\ 3)\ 3)\ 3)\ 3)$.

We introduce symbol $\triangleright$ to represent such infinite recursion. The expression

($\triangleright$ x y)

represents the expression y. The symbol $\triangleright$ anotates the expression y has subexpression x which has self recursive structure. The “left hand side” x should be a single logical variable. We do not consider the case the “left hand side” x is an expression including multiple logical variables. This research does not consider such complicated infinite structure. However we cannot find the case this notation cannot represent relations in our experiments. It is seem to be sufficient for many cases that the combination of this notation based on a single logical variable. Note that this notation is alomost same to #0# of the circular list.

Let us show example usage of the symbol $\triangleright$. The expression

$(\triangleright z\ \ (1\ z\ 2))$

represents

$(1\ (1\ (1\ \ (\ldots)\ 2)\ 2)\ 2)$

where z is a logical variable. The symbol $\triangleright$ anotates the subexpression z is self recursive.

This research also consider expressions have multiple annotations using the symbol $\triangleright$. Let us show another example. The expression

$((\triangleright u\ ((\triangleright v\ \ (1\ .\ v))\ u)))$

represents

$(v\ (v\ (v\ (\ldots)))) = ((1\ \ 1\ \ 1 \ldots)\quad ((1\ \ 1\ \ 1 \ldots)\quad ((1\ \ 1\ \ 1 \ldots)\quad (\ldots))))$

where u and v are logical variables. Using multiple $\triangleright$ can represent nested recursive structure.

recursive miniKanren has the mechanism of finding the self recursive structures that the symbol $\triangleright$ can represent. Let us show the execution result of the recursive miniKanren:

$(\,\text{run}^*\ (\,q\,)\ (\equiv q\ \text{`}(\,,q\ 3\,)\,)\,)$

$> (\triangleright\ _{-0}\ \ (\ _{-0}\ \ 3))$

where $_{-0}$ is a logical variable. Expanding this self recursive structures gives

$q =\ _{-0} = (((((\ldots)\ 3)\ 3)\ 3)\ 3)$

Comparing the expression using #0# with this result, recursive miniKanren can be regarded as an automatic detector of circular list. Let us call the self recursive relation based on $\triangleright$ as circular like relation (CLR).

3.1 Extended Triangular Substitutions

Triangular substitution [1] is a fundamental mechanism in logical programming. Extending triangular substitution enables finding CLR.

Normal miniKanren has preprocessing *occurs-check* which excludes infinite recursive relation before the main proces of the triangular substitution. Algorithm 1 shows deltails of *occurs-check*. The function *occurs-check*$(x\ v\ s)$ checks if it is self-recursive in all subtrees in the expression tree v. When a self-recursive subtree fund, normal miniKanren regards current relation invalid. recursive miniKanren assigns the annotation $\triangleright$ about the self-recursive. Algorithm 2 shows the difference between normal miniKanren and recursive miniKanren. Infinite recursions are exceptions which the occurs-check function causes. Algorithm 2 shows that $\triangleright$ can handle any exceptions which the occurs-check function causes.

Triangular substitution also has other processes traversing all subtree in a given expression. Traversing all subtrees including the infinite annotation ▷ is a complicated process. It might be infinite loop. To avoid infinite loop, recursive miniKanren uses *on-trees* macro [6] for traversing all subtrees.

Algorithm 1 occurs-check(x v s)

```
Require: $x$ is a logical variable we want to check if it is self-recursive in the expression $v$.
  $s$ is a dictionary (a set of definitions) of variables in current scope.
  function OCCURS-CHECK(x v s)
      expand $v$ by substituting definitions in $s$
      if $x = v$ then
          return true
      else
          for all $u \in \{$all subexpressions of $v\}$ do
              if OCCURS-CHECK(x u s) then
                  return true
          return false
```

Algorithm 2 normal and recursive miniKanren differecne

```
  if OCCURS-CHECK(x v s) then
      infinite recursion
      if normal miniKanren then
          $v$ is regarded invalid expression
      else if recursive miniKanren then
          $v \leftarrow (\triangleright\ x\ v)$
  else
      finite recursion
```

4 matchee Macro

match^{e} macro [8] is a pattern match macro which can describe the same pattern of the match macro of the Scheme language inside miniKanren. However, the match^{e} macro can not describe an iterative pattern including "..." . match^{ee} macro [3] is an extension of the match^{e} macro so that the iterative pattern can use. This macro uses "___" instead of "..." to describe the iterate patterns. Let us show example usage of this macro:

```
(run* (q)
    (match^{ee}
```

[3] https://github.com/niitsuma/Racket-miniKanren/blob/recursive2/matchee.scm

```
'((1 (2 3)) (10 (2 30)) (100 (2 300)))
[( (,a (2 ,b)) ___ )
 (≡ q `(,a ,b))]  ))
```

⇒

(((1 10 100) (3 30 300)))

In the above example, the match pattern describes an iteration of the pattern (,*a* (2 ,*b*)). In this case, the logical variables *a* is matched to (1,10,100) and the logical variables *b* is matched to (3,30,300). Like this example, match^{ee} macro can describe complicated iterative pattern by using "___".

match^{ee} is especially useful finding certain patterns from given list data. Let us consider the following example.

```
(run* (q)
 (match^ee
  '(1 2 3)
  [(,x ___ . ,r)
   (≡ q  `(,x ,r) )]))
```

⇒

'((() (1 2 3)) ((1) (2 3)) ((1 2) (3)) ((1 2 3) ()))

In this example, all possible sub-lists which can match to the given pattern are enumerated. The match pattern describes all possible sub-lists which can divide given input '(1 2 3). The possible devisions are
() and (1 2 3),   (1) and (2 3),   (1 2) and (3),   (1 2 3) and (). The macro successfully finds all possible patterns. This example shows the match pattern (,*x* ___ . ,*r*) can use to search all possible sub-list. This teqnique is useful search sub-sentence with some given patterns like context-free grammar.

matchee macro can use in both of normal miniKanren and recursive miniKanren. And *matchee* macro can use as original *matche* macro[8] . *matchee* gives the same results of original *matche* macro[8] as like the following example.

```
(run* (q)
 (match^ee
  '(1 2 3)
  [(,x . ,r)  (≡ q  `(,x ,r) )]))
```

⇒ '((1 (2 3)))

5 Context-Free Grammar with Left Recursion

Let us consider the following sentence example [2] for context-free grammar analysis.

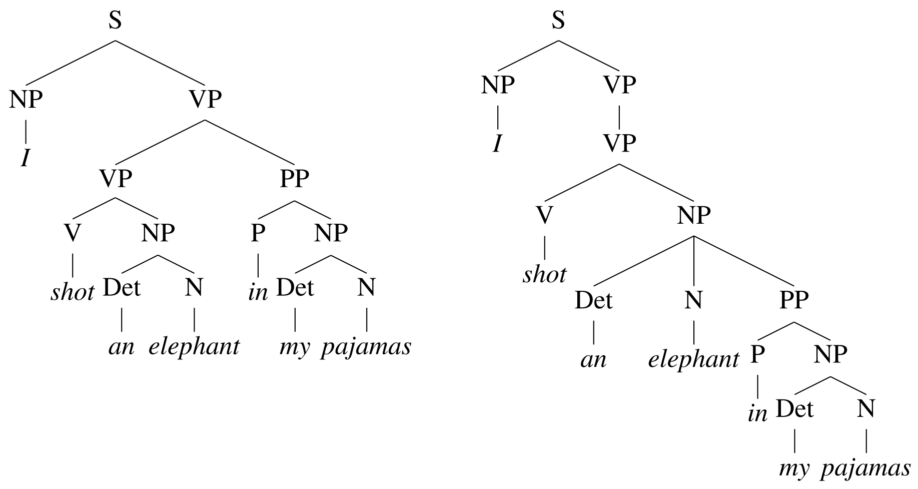
"I shot an elephant in my pajamas."

This sentence can be analyzed by using the flowing context-free grammar.

$$\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{PP} &\rightarrow \text{P NP} \\
\text{NP} &\rightarrow \text{Det N} \\
\text{NP} &\rightarrow \text{Det N PP} \\
\text{VP} &\rightarrow \text{V NP} \\
\text{VP} &\rightarrow \text{VP PP} \\
\text{N} &\rightarrow \text{'elephant'}|\text{'pajamas'} \\
\text{V} &\rightarrow \text{'shot'} \\
\text{P} &\rightarrow \text{'in'}
\end{aligned}$$

Here, the nonterminal S stands for sentence, NP for noun phrase, VP for verbphrase, Det for determiner, PP for prepositional phrase, N for Noun, V for Verb , and P for preposition. Note that the highlighted part has left recursion. This grammar is left recursive in the rules for NP. Usually, left recursion causes an infinite loop when analyzing a sentence. To avoid the infinite loop, a grammar including left recursion requires special treatments. For example, preprocessing to eliminate the left recursion can avoid the infinite loop [9].

The example sentence can be analyzed in two ways [2] as the following:



To find both two results, we need an algorithm with a backtracking mechanism based on top-down search.

recursive miniKanren can deal left recursion without special treatments. And can find both two results with a backtracking mechanism. Using match^{ee} macro, the grammar including left recursion can write as the following

(**define** (*cfg sent tree*)
 (match^{ee}

```
  sent
  [((NP . ,x)(VP . ,y) )
    (≡ tree `(S (NP . ,x)(VP . ,y)))]
  [(,pre ___ (P . ,x)(NP . ,y) . ,post)
   (fresh (t)
    (appendo pre `((PP (P . ,x)(NP . ,y)) . ,post)  t)
    (cfg t tree))]
  [(,pre ___ (Det . ,x)(N . ,y) . ,post)
   (fresh (t)
    (appendo pre `((NP (Det . ,x)(N . ,y)) . ,post)  t)
    (cfg t tree))]
  [(,pre ___ (Det . ,x)(N . ,y)(PP . ,z) . ,post)
   (fresh (t)
    (appendo pre `((NP (Det . ,x)(N . ,y)(PP . ,z)  ) . ,post)  t)
    (cfg t tree))]
  [(,pre ___ (V . ,x)(NP . ,y) . ,post)
   (fresh (t)
    (appendo pre `((VP (V . ,x)(NP . ,y)) . ,post)  t)
    (cfg t tree))]
  [(,pre ___ (VP . ,x)(PP . ,y) . ,post)
   (fresh (t)
    (appendo pre `((VP (VP . ,x)(PP . ,y)) . ,post) t )
    (cfg t tree))]
  ))
```

Note that the match^{ee} macro enables the grammar rule to almost directory write down the grammar rules as match patterns.

The grammar rule can apply to a sentence as the following way.

```
(remove-duplicates
    (run* (q) (cfg
        '((NP I) (V shot) (Det an) (N elephant)
            (P in) (Det my) (N pajamas)))'
        q))
```

```
⇒ '(

  (S (NP I)                                (S (NP I)
     (VP                                      (VP (V shot)
        (VP (V shot)                              (NP (Det an)
            (NP (Det an)                              (N elephant)
                (N elephant)))                        (PP (P in)
        (PP (P in)                                        (NP (Det my)
            (NP (Det my)                                      (N pajamas))))))
                (N pajamas)))))

  )
```

The desgired both two results are successfully extracted. Here, *remove-duplicates* is required because of this simple implementation extracts same results more than once. However, it is better than missing to find possible results. Recall the principal advantage of the miniKanren is it consists of under 1000 lines of Scheme code. recusive miniKanren is also consists of under 1000 lines of Scheme code. Using more than 1000 lines of Scheme code can remove *remove-duplicates*.

The match^{ee} macro also can use in normal miniKanren. The above code can run with normal miniKanren. However, normal miniKanren can not find these phased results, because of the left recursion.

5.1 Adding Logical Relation

recursive miniKanren is not a paraphrasing tool but computer language. It can add various relations as the language sentences. Let us consider adding rule; a single NP can not contain "elephant" and "pajamas" simultaneously. This rule can describe by adding few lines to the match pattern in the *cfg* function as the following

```
(define (cfg sent tree)
 (match^ee
  sent
  ...
  [(,pre ___ (Det . ,x)(N . ,y)(PP . ,z) . ,post)
   (fresh (t)
    (excludee
     (fresh ()
      (containo 'elephant `(,x ,y ,z))
      (containo 'pajamas `(,x ,y ,z)))
     (appendo pre `((NP (Det . ,x)(N . ,y)(PP . ,z)) . ,post)  t)
     (cfg t tree)))]
  ...
```

Using this rule successfully excludes the phased results represents "elephant wearing pajama". Here, *excludee* is a macro represents excluding a special case given in the first argument from current results:

```
(define-syntax excludee
  (syntax-rules ()
    ((_ pred b ...)
     (condu
       [pred fail]
       [alwayso b ...]))))
```

containo is a function giving a decision whether the list tree given in the secound argument contains the element given in the first argument:

```
(define containo
  (lambda (x l)
    (conde
((fresh (a)
        (caro l a)
        (≡ a x)))
((fresh (c d)
        (conso c d l)
        (conde
         [ (containo x c)]
         [ (containo x d)]
         ))))))
```

As like the above example, miniKanren can describe various conditions with a context-free grammar. The example code for this context-free grammar is in our repository. [4].

5.2 Expanding Context-Free Grammar

Our proposed technique can use to not only phase sentence also generate sentence. With the grammar rule *cfg* described with *mcathee* macro, just running following search program generates possible sentence which can keep the grammar rule.

```
(remove-duplicates
 (run 20 (q)
   (fresh (x)
     (cfg x  q)
     )))
```

⇒

```
(
  (S (NP . _0) (VP . _1))
  (S (NP (Det . _0) (N . _1)) (VP . _2))
  (S (NP (Det . _0) (N . _1) (PP . _2)) (VP . _3))
  (S (NP . _0) (VP (V . _1) (NP . _2)))
  (S (NP . _0) (VP (VP . _1) (PP . _2)))
  (S (NP (Det . _0) (N . _1) (PP (P . _2) (NP . _3))) (VP . _4))
  ...
)
```

[4] `https://github.com/niitsuma/Racket-miniKanren/blob/recursive2/context-free-grammar.scm`

5.3 Indirect Left Recursion

Our proposed technique also works for indirect left recursion. Let us consider the following grammar including indirect left recursion[5] .

$$\begin{aligned} &\mathrm{A} \rightarrow \mathrm{C}\ d \\ &\mathrm{B} \rightarrow \mathrm{C}\ e \\ &\mathrm{C} \rightarrow \mathrm{A} \mid \mathrm{B} \mid f \end{aligned}$$

This grammar rule can be described as the follwoing match pattern.

```
(define (cfg sent tree)
   (match^ee
    sent
    [((A . ,x)) (≡ tree ‘(A . ,x))]
    [((B . ,x)) (≡ tree ‘(B . ,x))]
    [((C . ,x)) (≡ tree ‘(C . ,x))]
    [(,pre ___ (C . ,x) d  . ,post)
     (fresh (t1)
       (appendo pre ‘( (A (C . ,x) d) . ,post)  t1)  (cfg t1 tree))  ]
    [(,pre ___ (C . ,x) e . ,post)
     (fresh (t1)
       (appendo pre ‘( (B (C . ,x) e) . ,post)  t1) (cfg t1 tree))  ]
    [(,pre ___ (A . ,x)  . ,post)
     (fresh (t1)
       (appendo pre ‘( (C (A . ,x) ) . ,post)  t1)  (cfg t1 tree))  ]
    [(,pre ___ (B . ,x)  . ,post)
     (fresh (t1)
       (appendo pre ‘( (C (B . ,x) ) . ,post)  t1)  (cfg t1 tree))  ]
    [(,pre ___f  . ,post)
     (fresh (t1)
       (appendo pre ‘( (C  f) . ,post)  t1)       (cfg t1 tree))  ]
   )
  )
```

The phase result for ’(f d e) is given by running the following code

```
(remove-duplicates (run* (q) (cfg ’(f d e) q) )))
```

⇒

```
(    (B (C (A (C f) d)) e)    (C (B (C (A (C f) d)) e))   )
```

This result shows our technique works for the indirect left recursion. Note that run* sentence is used for this phase procedure. run* sentence causes an infinete loop when executes for infinite recursive structure. However recursive miniKanren find and remove

[5] This example grammar takes from http://stackoverflow.com/questions/15999916/step-by-step-elimination-of-this-indirect-left-recursion

the indirect infinite recursive structure automatically. Running this code finishes in finite time.

6 Conclusion

match^{ee} macro can describe a context-free grammar as just write down the grammar rules as match patterns in the match^{ee} sentence. recursive miniKanren can handle left recursions of the grammar without special treatments. This technique can easily combine various logical statements to a context-free grammar.

References

1. Baader, F., Snyder, W.: Unification theory (1999)
2. Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python. O'Reilly Media (2009)
3. Byrd, W.E.: Relational programming in minikanren: techniques, applications, and implementations. Ph.D. thesis, Indiana University (2010)
4. Byrd, W.E., Holk, E., Friedman, D.P.: minikanren, live and untagged quine generation via relational interpreters. In: Proceedings of the 2012 Workshop on Scheme and Functional Programming (2012)
5. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. MIT Press, Cambridge, MA (2005)
6. Graham, P.: On LISP: Advanced Techniques for Common LISP. Prentice Hall (September 1993)
7. Hemann, J., Friedman, D.P.: microkanren: A minimal functional core for relational programming. In: Proceedings of the 2013 Workshop on Scheme and Functional Programming (2013)
8. Keep, A.W., Adams, M.D., Kuper, L., Byrd, W.E., Friedman, D.P.: A pattern matcher for miniKanren or how to get into trouble with CPS macros. In: Scheme '09: Proceedings of the 2009 Scheme and Functional Programming Workshop. pp. 37–45. No. CPSLO-CSC-09-03 in California Polytechnic State University Technical Report (2009), `http://digitalcommons.calpoly.edu/csse_fac/83/`
9. Moore, R.C.: Removing left recursion from context-free grammars. In: Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference. pp. 249–255. NAACL 2000, Association for Computational Linguistics, Stroudsburg, PA, USA (2000), `http://dl.acm.org/citation.cfm?id=974305.974338`