Self-aware Program Analysis in stableKanren

No Author Given

No Institute Given

Abstract. This paper presents two improvements to stableKanren that improve its compatibility and performance. miniKanren supports monotonic reasoning, in which one program produces exactly one model. stableKanren extends miniKanren to nonmonotonic reasoning, where a program can have zero to multiple models. The current overhead of supporting such nonmonotonicity in the stableKanren "run" interface creates two issues. First, a miniKanren program with infinite answers could not produce an answer under stableKanren. Second, a stableKanren program with the monotonicity property runs slower under stableKanren. These two types of programs are subject to monotonic reasoning and produce only one model. Therefore, the overhead for nonmonotonicity checking is unnecessary for these two types of programs. We construct a "run-partial" interface in stableKanren without nonmonotonicity checking. Using "run-partial" resolves the two issues; however, it requires expert knowledge to identify whether the program is monotonic or nonmonotonic and choose the corresponding interface. We further introduce two program analyses to the stableKanren "run" interface to identify the monotonic program implicitly. We use a macro to handle the first issue at the syntax level, so the "run" interface does not apply nonmonotonicity checking on the diverge miniKanren program. Furthermore, we present four syntax-level macros to create two smaller twins of the input program. We run these two twins to resolve the second issue at the syntax and semantics level. The twins of the monotonic programs produce identical outcomes, but the nonmonotonic programs do not. The updated "run" interface can identify monotonic programs at the syntax level; there is still more work to identify such programs during execution.

Keywords: logic programming · miniKanren · stableKanren · macros.

1 Introduction

Friedman et al. build miniKanren to show a natural way to extend functional programming to relational programming [1]. The core miniKanren implementation introduces only a few operators to users: == for *unification*, *fresh* for *existential quantification*, *conde* for *disjunction*, and a *run* interface. The reasoning capability provided by miniKanren is monotonic reasoning. Therefore, miniKanren programs always have and only have one model. To support nonmonotonic reasoning, Guo et al. extend miniKanren to stableKanren [2], stableKanren adds two more operators *noto* for *negation* and *defineo* for *goal function* definition. In

contrast to the programs under monotonic reasoning, nonmonotonic reasoning programs can have zero to multiple models. For example, the following stableKanren program has no model because of its contradiction.

Listing 1.1. A contradiction program in stableKanren

```
(defineo (cut_hair) (noto (cut_hair)))
```

The program defines one simple goal function *cut_hair*. To make such a goal succeed, the body of the goal function requires *cut_hair* to fail, which is a contradiction. It is impossible to make *cut_hair* succeed and fail simultaneously, so there is no model for such a program.

The missing *optimal substructure property* in nonmonotonic reasoning means the partial result obtained during the resolution may not be part of the final result. In stableKanren, the resolution in the *run* interface continues to check other goal functions using the partial result to ensure nonmonotonicity [2]. However, this process is unnecessary for monotonic programs and creates two issues. We present these issues in detail in Section 3. Firstly, monotonic programs with infinite answers could not produce an answer under stableKanren (Listing 1.4). Secondly, monotonic programs run slower under stableKanren (Listing 1.5). We introduce "run-partial" (Listing 1.6) to bypass the additional check on monotonic programs. The "run-partial" interface resolves the issues but requires expert knowledge to identify the monotonic programs to choose the corresponding interface. We want the "run" interface in stableKanren to detect monotonic programs automatically and decide to check nonmonotonicity implicitly.

In Section 2.1, we show that the input programs can be categorized as definite programs (Definition 2), stratified programs (Definition 6) and normal programs (Definition 4). The definite programs and the stratified programs are monotonic; hence, there is no need for nonmonotonicity checking. This paper adds two program syntax analyses to the "run" interface to identify definite programs and stratified programs. We focus on the syntax elements in stableKanren when creating macros. The definite programs have no negation (*noto*) in them. In Section 4.1, we design a new macro *has-negation?* (Listing 1.7) to distinguish definite programs. The stratified programs have negations, but the negations do not show up in the loop (Definition 5). In Section 4.2, we construct smaller executable twins (Listing 1.9, 1.10, 1.11) from the input program to analyze the relationship between negations and loops. We run the executable twins internally and compare the outcomes. The twins of the monotonic programs produce identical outcomes, but the twins of the nonmonotonic programs do not. As a result, we can identify stratified programs. In Section 4.3, we show that the updated "run" interface can identify monotonic programs to avoid nonmonotonic checking. In Section 5, we propose future work to identify monotonicity during execution.

2 Preliminaries

In this section, we review a few definitions, including definite programs (Definition 2), normal programs (Definition 4), and stratified programs (Definition

6). Then, we show a high-level overview of stableKanren and its nonmonotonic checking. Lastly, we present the issues in stableKanren's nonmonotonic checking.

2.1 Normal Programs

Let us define normal programs. To begin with, we use the definition of *definite program* and *normal program* from Lloyd [3].

Definition 1 (definite program clause). *A definite program clause is a clause of the form,*

$$A \leftarrow B_1, \cdots, B_n$$

where $A, B_1, \ldots, B_n$ *are atoms*[1]*.*

A definite program clause contains precisely one atom A in its consequent. A is called the *head* and $B_1, \ldots, B_n$ is called the *body* of the program clause.

Definition 2 (definite program). *A definite program is a finite set of definite program clauses.*

Based on the definition of the definite program clause, we have the definition of *normal program clause* and *normal program.*

Definition 3 (normal program clause). *A normal program clause is a clause of the form,*

$$A \leftarrow B_1, \cdots, B_n, not\ B_{n+1}, \cdots, not\ B_m$$

For a normal program clause, the body of a program clause is a conjunction of literals instead of atoms; $B_1, \cdots, B_n$ are *positive literals* and $not\ B_{n+1}, \cdots, not\ B_m$ are *negative literals.*

Definition 4 (normal program). *A normal program is a finite set of normal program clauses.*

According to Gelder et al., there are sets of atoms named *unfounded sets* in a normal program that can help us categorize the normal programs [5].

Definition 5 (unfounded set). *Given a normal program, the unfounded set is a set of atoms that only cyclically support each other, forming a loop.*

Considering the combinations of negations and unfounded sets (loops) in normal programs, we have informal definitions of *stratified program.*

Definition 6 (stratified program). *Given a normal program, it is stratified if all unfounded sets (loops) do not contain any negation.*

[1] Atom is evaluated to be true or false.

2.2 stableKanren

stableKanren extends miniKanren to support nonmonotonic reasoning [2]. It handles the negation and loop in normal programs. To support negation, stableKanren defines a set of internal macros and functions that implicitly transform the positive goal from the user's input into a negative counterpart. Then, the positive and negative goals are unified under one goal function using *defineo* macro. The *negation counter (n)* counts the number of *noto* and decides to use a positive or negative goal during resolution. To resolve the loop, stableKanren wraps *local tabling (P)* and *call stack frame (cfs)* around the unified goal function. The *ext-p* and *expand-cfs* extend the local tabling (P) and call stack frame (cfs) respectively.

As a result, stableKanren can solve normal programs (Definition 4). For example, Emden et al. introduce a two-person game [4]. The game is given a directed connected graph, a peg on a starting node, and two players. Each player takes a turn to move the peg to the adjacency node; the winning move in this game is defined as there is a move that will make it so that the opponent has no move. We give an example graph in Figure 1 and the stableKanren program to

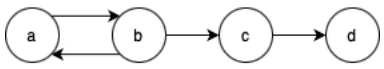

Fig. 1. A playboard in two-person game

find the winning positions as follows.

Listing 1.2. Two person game in stableKanren

```
(defineo (move x y)
  (conde [(== x 'b) (== y 'c)] [(== x 'a) (== y 'b)]
         [(== x 'b) (== y 'a)] [(== x 'c) (== y 'd)]))
(defineo (win x) (fresh (y) (move x y) (noto (win y))))
```

In this example, the variable x of *win* unifies with values representing the winning positions. A set of queries and outputs through *run* is shown as follows.

```
> (run 3 (q) (win q))
(c b a)
> (run 1 (q) (win 'c) (win 'a))
(_.0)
> (run 1 (q) (win 'b) (win 'a))
()
```

The *run* interface takes three parameters. The first parameter is the number of answers we expect; the query returns answers no more than this number. The second parameter is the query variable, which stores the answers found by the query. The third parameter is the actual query. The first query generates possible winning positions. The second query asks "Can node c and node a both be the winning position?" It returns a list containing one element "_.0", a representation of anything in miniKanren and stableKanren. Anything can let our queries succeed. So, a and c can both be the winning position. The last

query asks: "Can node b and node a both be the winning position?" It returns an empty list, representing nothing in miniKanren and stableKanren. In this case, nothing can let our queries succeed. Therefore, a and b can not be the winning position simultaneously.

3 Problems

As Guo et al. mentioned, the partial result of the normal program (Definition 4) may not be the final result since the missing *optimal substructure property* in nonmonotonic reasoning [2]. Inside *defineo* (Listing 1.3), the goal function adds to a checking set (*program-rules*) for future nonmonotonicity checking.

Listing 1.3. A stableKanren defineo macro

```
(define-syntax defineo
  (syntax-rules ()
    ((_ (name params ...) exp ...)
      (begin
      (set! program-rules
            (adjoin-set (make-record 'name
                          (length (list 'params ...)))
                        program-rules))
      ;;; Omitted other details.
      ))))
```

The resolution in the *run* interface continues to check other goal functions using the partial result to ensure nonmonotonicity.

We notice that not all programs are required to check for nonmonotonicity. In particular, nonmonotonic checking on monotonic programs creates two issues. Firstly, definite programs that do not involve any negation are considered monotonic reasoning. Applying nonmonotonic checking on definite programs with infinite answers does not terminate. For example, consider the following program without using *noto* written in stableKanren in Listing 1.4.

Listing 1.4. A "revo" relation to determine two lists are reversed to each other

```
(defineo (revo xs sx)
  (fresh (empty) (nullo empty) (rev-acco xs empty sx)))
(defineo (rev-acco xs acc sx)
  (conde [(nullo xs) (== sx acc)]
         [(fresh (h t acc1)
            (conso h t xs)
            (conso h acc acc1)
            (rev-acco t acc1 sx))]))
```

The *revo* determines if the two lists are reversed to each other. A simple query on the two empty lists does not terminate.

```
> (run 1 (q) (revo '() '()))
; infinity loop
```

Secondly, stratified programs in which the loop does not contain any negation are considered monotonic reasoning. Applying nonmonotonic checking on stratified programs slows down the running speed. For instance, consider the strongly connected components (SCCs) of Figure 2. We will call an SCC a final SCC if the only nodes reachable from nodes in that SCC are others in that SCC.

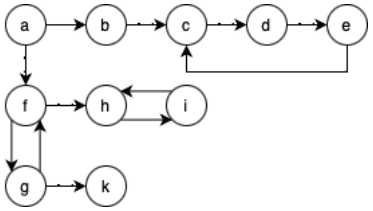

Fig. 2. A directed graph contains multiple SCCs

Now, given a node, we want to find all nodes that are reachable from that node in the final SCCs. The solution in stableKanren is as follows.

Listing 1.5. An encoding of final-SCC problem in stableKanren

```
(defineo (edge x y)
  (conde [(== x 'a) (== y 'b)] [(== x 'b) (== y 'c)]
         [(== x 'c) (== y 'd)] [(== x 'd) (== y 'e)]
         [(== x 'e) (== y 'c)] [(== x 'a) (== y 'f)]
         [(== x 'f) (== y 'h)] [(== x 'f) (== y 'g)]
         [(== x 'g) (== y 'f)] [(== x 'g) (== y 'k)]
         [(== x 'h) (== y 'i)] [(== x 'i) (== y 'h)]))

(defineo (reachable x y)
  (conde [(edge x y)]
         [(fresh (z) (edge x z) (reachable z y))]))

(defineo (reducible x)
  (fresh (y) (reachable x y) (noto (reachable y x))))

(defineo (fullyReduce x y)
  (reachable x y) (noto (reducible y)))
```

To run a query on *fullyReduce* will produce all final-SCC pairs for us.

```
> (time (length (run* (q) (fresh (x y)
                        (fullyReduce x y) (== q `(,x ,y))))))
100
```

Getting 100 final-SCC pairs takes 32.81 seconds. We will see this is relatively slower than running a query using "run-partial" (Listing 1.6).

As we have shown in Section 2.1, the definite programs and stratified programs have and only have one model since they have *optimal substructure properties*. So, the partial result of these programs is also a part of the final result. We can use a "run-partial" interface in Listing 1.6 to bypass nonmonotonic checking.

Listing 1.6. A run-partial interface without unavoidable contradictions checking

```
(define-syntax run-partial
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (lambdaf@ ()
         ((fresh (x) g0 g ...
          (lambdag@ (negation-counter cfs c : S P)
              (cons (reify x S) '())))
          negation-counter call-frame-stack empty-c))))))
```

The *run-partial* has the extended lambdag@ (adding negation counter and call stack frame) introduced by stableKanren but has no nonmonotonicity checking. Using "run-partial" to query on Listing 1.4, the query can return a result. Also, using "run-partial" to query *fullyReduce* on Listing 1.5, it produces 100 final-SCC pairs in 0.01 seconds.

Although the *run-partial* resolves those two issues, it requires expert knowledge to identify whether a program is monotonic or not. The user has to make the right decision on which interface to run the query. We want to incorporate expert knowledge into the "run" interface so that it can identify the monotonic program implicitly.

4 Identify Monotonic Programs in stableKanren

The key idea is identifying negations and loops in stableKanren programs. The syntax of stableKanren consists of only five elements, ==, *fresh*, *conde*, *noto*, *goal functions*. Identifying negations is easy, a syntax analysis using macro can achieve this. A goal function is definite as long as there is no negation (*noto*) in it. Identifying loops requires more analysis of the relation between goal functions. Instead of creating a dependency graph representation to the input program, then have a stand-alone algorithm to analyze the graph. We use four macros to construct smaller executable twins (Listing 1.9, 1.10, 1.11) from the input program. These smaller twins are variants of stableKanren programs with different semantics on the loops. Therefore, we can run the twins internally and compare the outcomes. The twins of the monotonic programs produce identical outcomes, but the twins of the nonmonotonic programs do not.

4.1 No Negation, No Problem

The stableKanren programs without using any *noto* (negation) operator, like the one we see in Listing 1.4, are definite programs. This section introduces a compilation time syntax analysis macro *has-negation?* to identify definite programs.

Listing 1.7. A syntax analysis to identify negation in the program

```
(define-syntax has-negation?
  (syntax-rules (noto conde fresh)
    ((_ (noto g)) #t)
    ((_ (conde (g0 g ...) (g1 g^ ...) ...))
        (or (has-negation? g0 g ...)
            (has-negation? g1 g^ ...)
            ...))
    ((_ (fresh (x ...) g0 g ...))
        (has-negation? g0 g ...))
    ((_ g) #f)
    ((_ g0 g1 ...)
        (or (has-negation? g0)
            (has-negation? g1)
            ...))))
```

The *has-negation?* iterates through nested *conde* and *fresh* until it reaches the goal function level. If there is a negative goal function, it returns true, and vice versa. We apply this syntax analysis to improve the *defineo* macro in Listing 1.8.

Listing 1.8. Improved defineo macro

```
(define-syntax defineo
  (syntax-rules ()
    ((_ (name params ...) exp ...)
      (begin
        (if (has-negation? exp ...)
          (set! program-rules
                (adjoin-set (make-record 'name
                              (length (list 'params ...)))
                            program-rules)))
      ;;; Omitted other details.
      ))))
```

In the improved *defineo*, only a goal function that has negation will be added to the nonmonotonic checking set. Goal functions with negations may not be checked for nonmonotonicity if these negations are stratified.

4.2 Positive and Negative Twins

Identifying stratified programs requires more analysis of the relation between goal functions. This section presents a new approach to constructing two smaller executable twins, positive and negative, from the input program. Also, we reuse the infrastructure in stableKanren to find loops, but we give the positive and negative twins different semantics on the loops. Lastly, we identify the stratified programs by running positive and negative twins and comparing the outcomes.

A smaller twin only captures the dependency between goals, so it does not have any variables in the program. Also, it appends a suffix (+ or −) to each goal function. Listing 1.9 shows how to create a negative twin.

Listing 1.9. Macro to transform the input program to a negative twin

```
(define-syntax negative-twin
  (syntax-rules (conde fresh noto == succeed fail)
    ((_ (conde [(== a ...) ...] [(== b ...) ...] ...))
        succeed)
    ((_ (fresh (x ...) g g0 ...))
        (fresh () (negative-twin g) (negative-twin g0) ...))
    ((_ (conde [g g0 ...] [g1 g^ ...] ...))
        (conde
          [(negative-twin g) (negative-twin g0) ...]
          [(negative-twin g1) (negative-twin g^) ...] ...))
    ((_ (noto g))
        (noto (negative-twin g)))
    ((_ (== u v))
        succeed)
    ((_ succeed) succeed)
    ((_ fail) fail)
    ((_ (p ...))
        (eval `(,(sym-append-str (car `(p ...)) "-"))))
    ((_ g g0 ...)
        (fresh ()
          (negative-twin g) (negative-twin g0) ...))))
```

The *negative-twin* iterates through nested *conde* and *fresh* until it reaches the goal function level. It appends a − to each goal function. Since it removes all variables in the program, all unifications (== operators) are converted to *succeed*. The *positive-twin* does the same transformation except it appends a + to each goal function.

We wrap the *negative-twin* and *positive-twin* with loop handling under different semantics. Also, a suffix (+ or −) is appended to each goal function. Listing 1.10 shows the semantics for loops under negative twin.

Listing 1.10. Macro to create an executable negative twin

```
(define-syntax define-negative-twin
(syntax-rules ()
((_ name exp ...)
  (eval
    `(define (,(sym-append-str `name "-"))
       (lambdag@ (n cfs c : S P L)
       (let* ([name- (sym-append-str `name "-")]
              [signature (list name- `())]
              [result (element-of-set? signature P)]
              [record (element-of-set? signature cfs)])
         (if (or (and result #t) (and record #t))
           (unit c)
           ((fresh ()
              (negative-twin exp ...) (ext-p name- `()))
            n (expand-cfs signature n cfs) c)))))))))
```

The local tabling, and loop handling are infrastructures in stableKanren. The local tabling is recorded on P and extended by *ext-p*, and the loop is tracked on *cfs* and updated by *expand-cfs*. The *define-negative-twin* always succeeds on the loops. Similarly, Listing 1.11 shows the semantics for loops under positive twin.

Listing 1.11. Macro to create an executable positive twin

```
(define-syntax define-positive-twin
(syntax-rules ()
((_ name exp ...)
  (eval
    `(define (,(sym-append-str `name "+"))
       (lambdag@ (n cfs c : S P L)
       (let* ([name+ (sym-append-str `name "+")]
              [signature (list name+ `())]
              [result (element-of-set? signature P)]
              [record (element-of-set? signature cfs)])
         (if (and result #t)
           (unit c)
           (if (and record #t)
             (let ([diff (- n (get-value record))])
               (if (= 0 diff)
                 ; Positive loop (stratified negation)
                 (unit c)
                 ; Negative loop (normal program)
                 (mzero)))
              ((fresh ()
                 (positive-twin exp ...) (ext-p name+ `()))
               n (expand-cfs signature n cfs) c)))))))))))
```

The only semantics difference between *define-negative-twin* and *define-positive-twin* is that the *define-positive-twin* fails on the negative loops. Both *define-*

negative-twin and *define-positive-twin* are created in *defineo* as shown in Listing 1.12.

Listing 1.12. Extended defineo macro to create twins

```
(define-syntax defineo
  (syntax-rules ()
    ((_ (name params ...) exp ...)
      (begin
        (if (has-negation? exp ...)
          (set! program-rules
                (adjoin-set (make-record 'name
                              (length (list 'params ...)))
                            program-rules)))
        (define-positive-twin name exp ...)
        (define-negative-twin name exp ...)
      ;;; Omitted other details.
      ))))
```

The extended *defineo* creates positive and negative twins from the user input during compilation. It also uses *has-negation?* to avoid nonmonotonic checking on definite programs. Lastly, we can run the twins internally using *run-partial* (Listing 1.6) and compare the outcomes as follows,

```
(define (stratified?)
  (fold-left
    (lambda (l r) (and l r)) #t
    (map
      (lambda (p)
        (= (length
             (run-partial #f (q)
               (apply (eval (sym-append-str (get-key p) "+"))
                      '())))
           (length
             (run-partial #f (q)
               (apply (eval (sym-append-str (get-key p) "-"))
                      '()))))))
      program-rules)))
```

As we have shown the semantics between positive twin and negative twin are slightly different. Therefore, the twins of the monotonic programs produce identical outcomes, but the twins of the nonmonotonic programs do not.

4.3 Enhanced Run

This section uses the improvements we made to make the *run* interface automatically identify monotonic programs. The enhanced *run* is as follows,

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (lambdaf@ ()
         ((fresh (x) g0 g ...
          (lambdag@ (negation-counter cfs c : S P L)
            (if (and (not (stratified?))
                     (null? (nonmonotonic-checking)))
              (mzero)
              (cons (reify x S) '()))))
          negation-counter call-frame-stack empty-c))))))
```

It uses *stratified?* to avoid nonmonotonic checking on stratified programs. As a result, using our updated *defineo* and *run*, the query on Listing 1.4 can produce an answer, and the query on Listing 1.5 produces 100 pairs of final SCCs in 0.03 seconds.

5 Conclusion and Future Work

This paper presents two improvements to stableKanren, so it will not apply nonmonotonic checking on monotonic programs. There are two types of programs, definite programs (Definition 2) and stratified programs (Definition 6), which are monotonic programs. We add a syntax analysis macro *has-negation?* (Listing 1.7) to stableKanren's *defineo* to identify definite programs and avoid checking nonmonotonicity. As a result, stableKanren can produce answers for definite programs with infinite answers. We also introduce four macros to stableKanren's *defineo* to transform the input program into two smaller positive and negative twins. These twins help us distinguish stratified programs so that stableKanren saves running time for stratified programs.

For future work, a normal program may behave like a stratified program during the execution time, hence the nonmonotonic checking can be bypassed. For example, from the syntax perspective the two-person game in Listing 1.2 is a normal program as the negation appears in the loop. However, if the game is played on a directed acyclic graph (DAG), the answer has *optimal substructure property* and it is stratified negation. Currently, our smaller twins cannot capture this runtime behavior as the variables are discarded during the twin creation. Another issue is when a contradiction in Listing 1.1 coexists with other programs in the same environment, the *run* interface cannot produce an answer even if the query did not touch the contradiction part. We can still use *run-partial* (Listing 1.6) to produce the answers, but ideally, we want to enhance *run* to isolate the contradiction implicitly.

References

1. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. The MIT Press, MIT Press (2005). https://doi.org/10.7551/mitpress/5801.001.0001
2. Guo, X., Smith, J., Bansal, A.: Stablekanren: Integrating stable model semantics with minikanren. In: Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming. PPDP '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3610612.3610617
3. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987). https://doi.org/10.1007/978-3-642-83189-8, https://doi.org/10.1007/978-3-642-83189-8
4. Van Emden, M.H., Clark, K.L.: The Logic of Two-Person Games, chap. 12, pp. 320–340. Prentice-Hall, Inc., USA (1984)
5. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 619–649 (Jul 1991). https://doi.org/10.1145/116825.116838, https://doi.org/10.1145/116825.116838