Stable Model Semantics Extension of miniKanren

XIANGYU GUO, JAMES SMITH, and AJAY BANSAL, Arizona State University, USA

This paper presents a miniKanren extension with negation support under stable model semantics called stableKanren. By utilizing macros and continuations in the functional programming language Scheme, miniKanren shows an innovative approach to achieving resolution and unification, the essence of Prolog. Also, miniKanren is designed to be easily modified and extended with new features. Extending miniKanren with negation that enables non-monotonic reasoning (NMR) becomes an intriguing topic. We choose stable model semantics as the underlying semantics since it is well-defined and has gained popularity in logic programming. Moreover, implementing a solver with stable model semantics is a challenging task due to the NP-hard nature of the problem. The logic programming community has shifted from top-down resolution and unification to the bottom-up grounding and constraint-propagation approach because of the uneasy modification to the underlying resolution and unification algorithms. We design our algorithms to evolve resolution and unification that requires dynamic code modification and generation. With direct access to resolution and unification in miniKanren and easy code generation in functional programming, we have implemented our stable model solving algorithms using these features.

CCS Concepts: • **Software and its engineering** → *Functional languages*; • **Computing methodologies** → *Logic programming and answer set programming*.

Additional Key Words and Phrases: logic programming, miniKanren, stable model semantics

1 INTRODUCTION

In the 1960s, John McCarthy, one of the first to propose a declarative programming paradigm in computer science, proposed that the most natural language for specifying problems and solutions would be logic and, in particular, predicate logic [21]. Logic programming turns a unidirectional function into a bidirectional relation so that an expression can produce more than one result. Prolog is an example language in this paradigm that combines a relational vision of programming with a symbolic pattern matching mechanism named *unification* (definition 2.19). Over time, many logic programming solvers were developed supporting the classical Prolog style as the input syntax format [16], [17]. However, researchers from the functional programming community believed that logic programming should be incorporated into the functional programming syntax instead of a new Prolog-based syntax so that the user can quickly adapt to this new paradigm. In the 1980s, John Alan Robinson, the inventor of the *resolution* (definition 2.16) algorithm, developed LOGLISP as an alternative to Prolog but using LISP syntax [25]. Later, Daniel Friedman et al. built miniKanren, a system that focuses on pure relation and finite failure [8], [9]. MiniKanren replicated the essence of Prolog using macros to create a static search stream connected through continuations. The resolution and unification were an elegant solution to *monotonic reasoning*, and the declarative semantics underneath monotonic reasoning is the *minimal Herbrand model semantics* [28].

In the 1980s, the emergence of negation in *normal logic programs* (definition 2.4) leads to the *non-monotonic reasoning (NMR)*. Van Emden et al. showed a normal program of the winning position in a two-person game is defined as there is a move that will make it so the opponent has no move [27]. For such a program, good semantics should be a generalization of the minimal Herbrand model semantics in a simple way. Many logic programming researchers were trying to define proper semantics for negation in normal logic programs. Michael Gelfond and Vladimir Lifschitz proposed well-defined and widely accepted *stable model semantics* (definition 2.15) as a generalization of minimal model semantics [14]. The stable model semantics give more power to logic programming so that we can do non-monotonic reasoning, and a new paradigm named

Authors' address: Xiangyu Guo, Xiangyu.Guo@asu.edu; James Smith, jsmit106@asu.edu; Ajay Bansal, Ajay.Bansal@asu.edu, Arizona State University, 699 S Mill Ave, Tempe, Arizona, USA, 85281.

answer set programming (ASP) emerged. Many applications have been generated since then, like nurse scheduling problems [7], modeling living systems [13], train table arrangement [1], and various other hybrid problem-solving approaches [18]. Recently, we have seen there is interest in having negation in miniKanren to increase its expressiveness of solving more problems. William Byrd et al. developed mediKanren for biomedical reasoning [4]. Adding negation to miniKanren will allow mediKanren to have the non-monotonic reasoning ability to handle the contradiction in the knowledge graph. Evgenii Moiseenko's constructive negation [23] can solve stratified negation, but not non-stratified negation under stable model semantics.

In the late 1990s, the difficulty in solving the NMR program drove the logic programming community to turn away from top-down resolution and unification and embrace the bottom-up grounding and constraint-propagation approach. There were some who started to believe that a "fully top-down" solving procedure would be impossible even in the propositional case since, for some programs, the truth of a literal w.r.t. stable model semantics cannot be decided when considering only the atom dependency graph below it [2]. Moreover, some researchers pointed out that ASP and the underlying stable model semantics lacked a relevance property; the truth value of an atom can depend on other, totally unrelated rules and atoms [6]. Later, in 2017, Kyle Marple et al. developed s(ASP), a top-down solver that tried to supplement the missing information of the input program dependency graph during compilation time [20]. However, our recent research discovered that solving such an NMR program using resolution and unification requires a dynamic search stream created at the runtime. Also, miniKanren made it easy to modify the resolution and unification in an accessible way, and we realized that functional programming is well-suited to achieve our goal.

As we can see, the advantage of stable model semantics is valuable; solving normal logic programs in a top-down manner is feasible, and miniKanren is also a top-down solver that can be modified easily. In this paper, we present stableKanren, a stable model semantics extension to the miniKanren system that combines the top-down normal logic program solving idea with functional programming. We also realize that some constructs of the functional programming language Scheme, like macros and continuations, can help solve normal logic programs. MiniKanren uses macros to create a static search stream connected through continuations in monotonic reasoning. Our stableKanren is going further with macros and continuations to create a dynamic search stream in non-monotonic reasoning. We believe our solver can benefit both communities in the sense that it brings stable model semantics to miniKanren, and it shows the advantages of using functional language to implement normal program solver to logic programming.

2 BACKGROUND AND RELATED WORK

In this section, we present the background concepts and some related work that has been done. We introduce some background information on functional programming (FP) and miniKanren, logic programming (LP) and stable model semantics. We then walk through the two different stable model solving solutions developed by the LP community, namely the top-down versus the bottom-up, and eventually, we show some connections between miniKanren and LP.

2.1 Functional programming and miniKanren

We have seen many benefits from using FP over the years. Firstly, doing meta-programming through macros makes the code look simple and concise. Secondly, using immutable variables leads to fewer side-effects, hence easier to test, and fewer bugs. Thirdly, even though a pure function may look simple, it can be easily combined with other functions to create complex features.

MiniKanren is a system that shows a way to build features of logic programming atop a functional programming language [3]. The original implementation was hosted on Scheme [10]. The core

implementation introduced only a few operators: "==" for unification, "fresh" for existential, "conde" for disjunction, and a "run" interface. Unlike Prolog, miniKanren uses a complete interleaving search strategy on a stream to simulate backtracking [30]. Also, the interleaved stream resolved the issue that the sequence of substitutions may be infinite to guarantee fairness when producing the result [8].

MiniKanren uses a special syntax notation, *lambdag@*, to represent an internal goal definition function that maps a substitution S to an ordered sequence of zero or more substitutions. In this way, we enforced the contract (signature) among all goal functions. For the reader using non-Scheme-based miniKanren implementation, find the corresponding *lambdag@* similar to listing 1 in your project, as our algorithms are relying on it heavily.

Listing 1. A macro to define goal function's internal continuation

```
(define-syntax lambdag@
  (syntax-rules (:) ((_ (S) e ...) (lambda (S) e ...))))
```

The *lambdag@* is a simple macro definition that transforms "lambdag@" to "lambda" so that the Scheme runtime can define an anonymous lambda expression.

2.2 Logic programming and stable model semantics

Llyod presented the definition for *definite program* and *normal program* [19].

Definition 2.1 (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.

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

Based on the definite program clause's definition, Llyod defines the *normal program clause* and *normal program*.

Definition 2.3 (normal program clause)**.** For a *normal program clause*, the body of a program clause is a conjunction of literals instead of atoms,

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

Definition 2.4 (normal program)**.** A *normal program* is a finite set of normal program clauses.

Just like a function in functional programming can have variables, similarly in logic programming, we can add variables to each atom. Therefore, we have two types of variables in logic programming, *head variable* and *body variable*.

Definition 2.5 (head variable)**.** A head variable is a variable that shows up in both the clause's head and body.

Definition 2.6 (body variable)**.** A body variable is a variable that shows up only in the clause's body.

We use quantifiers like $\exists$ and $\forall$ over variables in logic programming so that each *free variable* can be *grounded* to a value. Furthermore, we can define two types of logic rules (statements, clauses), *propositional rule* and *predicate rule*.

Definition 2.7 (propositional rule)**.** A propositional logic rule is a rule without any free variables in it, and it can be evaluated to get a truth value directly.

Definition 2.8 (predicate rule)**.** A predicate logic rule is a rule with free variables and quantifiers in it, and it needs to quantify its variables before evaluating a truth value.

Definition 2.9 (grounding)**.** Grounding is a process of assigning free variables with a value. Doing so turns a predicate rule into a propositional rule.

According to Allen Van Gelder et al., there are sets of atoms named *unfounded sets* in a normal program that can help us categorize the normal programs [29].

Definition 2.10 (unfounded set)**.** Given a normal program, the atoms inside the unfounded set are only cyclically supporting each other, forming a loop.

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

Definition 2.11 (tight program)**.** Given a normal program, it is tight if there is no unfounded set (loop) in the program.

Definition 2.12 (stratified program)**.** Given a normal program, it is stratified if all unfounded sets (loops) do not contain any negation.

Over the years, a few important semantics have been invented to tackle negation literals in normal programs, like *closed world assumption (CWA)* [24], *negation as failure (NAF)* [5], and *well-founded semantics* [29].

Definition 2.13 (closed world assumption)**.** Given a logic program, an atom that is not currently known to be true is false.

Definition 2.14 (negation as failure)**.** Given a normal program, *not B* succeeds iff *B* fails.

Clark's NAF and Clark's completion were an attempt at tight programs. The well-founded semantics by Allen Van Gelder et al. was for a non-tight but stratified program. Eventually, the *stable model semantics* by Michael Gelfond and Vladimir Lifschitz [14] can be seen as an attempt to generalize the semantics for non-stratified programs. Miroslaw Truszczynski [26] introduces an alternative reduct to the original definition.

Unlike the original definition that deletes rules during the reduct computation, the alternative reduct leaves rule heads intact and only reduces rule bodies. This feature suits our needs, hence, we are using the alternative reduct to describe the declarative semantics of stable model semantics in three steps.

Definition 2.15 (stable model semantics)**.** Given an input program P, the first step is getting a *propositional image* of P. A propositional image Π is obtained from grounding each variable in P. The second step is enumerating all interpretations I of Π. For a Π that has N atoms, we will have 2^N interpretations. The third step is using each model M from I to create a *reduct program* Π_M and verify M is the minimal model of Π_M. To create a reduct program, we are replacing a negative literal $\neg B_i$ in the rule with $\bot$ if $B_i \in M$; otherwise, we are replacing it with $\top$. Once completed, Π_M is negation-free and has a unique minimal model M'. If $M = M'$, we say M is a stable model of P.

For example, consider the program P

$$p(1, 2).$$
$$q(x) \leftarrow p(x, y), \neg q(y).$$

The domain of x, y is $\{1\} and \{2\}$ respectively. Let Π be P with the second rule replaced by its ground instance:

$$q(1) \leftarrow p(1,2), \neg q(2).$$

Let $M = \{q(2)\}$. Then Π_M is

$$p(1,2).$$
$$q(1) \leftarrow p(1,2), \bot.$$

The minimal Herbrand model M' of this program is $\{p(1,2)\}$. It is different from M, so that M is not a stable model of P. Now let us try $M = \{p(1,2), q(1)\}$. In this case Π_M is

$$p(1,2).$$
$$q(1) \leftarrow p(1,2), \top.$$

The minimal Herbrand model M' is the same as M, hence $\{p(1,2), q(1)\}$ is a stable model of P.

2.3 Two approaches to solve logic programs

Over the years, there have been two approaches to solving a logic program namely top-down and bottom-up. A top-down solver, such as Prolog and miniKanren, uses *resolution* and *unification* to obtain a model of the input program. We define resolution, goal's signature, resolution loop, and unification in normal programs as follows.

Definition 2.16 (resolution)**.** Resolution is selecting a sub-goal "g" that can be proven true. The resolution starts from the goal in the query and expands the call frame stack (CFS) by selecting a clause that the head of the clause unifies with the goal and recursively expands the sub-goals in the body of the clause.

Definition 2.17 (goal's signature)**.** A goal's signature consists of the goal's name and parameters bounded to the values. If the parameter has no bounded value, we use the parameter name as part of the signature.

Definition 2.18 (resolution loop)**.** Resolution records a goal's signature on the call frame stack (CFS) during recursive goal expansions. If a goal expansion sees itself, the same signature, on the CFS, we have a resolution loop (which we refer to as a loop from now on) starting at the current goal.

Definition 2.19 (naive unification)**.** In top-down solving, the truth value of a positive goal is determined by the unification outcome. If the unification successfully returned, it is equivalent to assigning true to the goal, and vice versa. Additionally, the return substitutions set is extended by the goal function if any unbounded variable is bounded through unification.

We categorize the roles of resolution and unification played during top-down solving as we are going to extend these roles in our research.

Definition 2.20 (roles of resolution)**.** Resolution has only one role, selecting a goal, assuming that goal is true to produce a minimal model eventually.

Definition 2.21 (roles of unification)**.** Unification plays two roles, the first role is to assign a truth value to a goal, and the second role is to assign a value to a variable.

In the late 1990s, the LP community moved on to consider relational programming based on techniques other than simple resolution and unification, such as the ability to deal with numerical constraints, and those techniques are widely used in constraint-propagation systems. Many SAT

solvers adopted the constraint-propagation approach and achieved significant improvements in the early 2000s. In 2007, inspired by the Conflict Driven Clause Learning algorithm in SAT solver, Gebser et al. presented Conflict Driven Nogood Learning (CDNL) for Answer Set solving [12], and enumeration [11]. A bottom-up solver like Clingo, uses *grounding* (definition 2.9) and *constraint propagation* to obtain a model of the input program. CDNL introduces the concept of *loop nogoods* to resolve unfounded sets (definition 2.10), but as there are exponentially many, these loop nogoods are only computed on-the-fly. However, the grounding stage leaves a heavy memory footprint becoming a big issue when applying the bottom-up solver to a large-scale problem.

To keep the advantage of top-down solving without dealing with the heavy memory footprint caused by the grounding. Goal Gupta et al. introduced conductive logic (co-LP) and co-SLD resolution, co-SLD produces the greatest fixed point of a program [15]. Later Richard Min et al. evolved co-SLD to co-SLDNF to achieve normal program solving [22]. However, we believe that sticking with the least fixed point of SLD resolution is closer to stable model semantics and the transition is simpler. Therefore, in contrast to co-SLD and co-SLDNF, our algorithm still produces the least fixed points as the original resolution and focuses on adding stable model semantics under the finite Herbrand model.

Our approach attempts to solve normal programs using top-down resolution and unification with a dynamic search stream created at runtime so that we can balance the memory usage and solving time.

2.4 Connection between logic programming and miniKanren

In this section, we establish a connection between logic programming and miniKanren that shows how unification works in propositional logic and predicate logic. Moreover, we show how miniKanren handles CWA differently than Clingo under propositional and predicate scenarios. Furthermore, we explain the advantages and challenges variables brought to us and why we want to isolate variables as a starting point to build our solver. Eventually, we compare and contrast the existing attempts of the negation extension that has been built over the years.

Let us start with zero arity of variables. From the definition of propositional logic (definition 2.7), we know that there are no variables in propositional rules. Also, we know that unification usually has two roles in top-down solving (2.21), but under propositional logic, we only use one role of unification, which is producing a truth value. This can be verified by the logic $\top$ and $\bot$ representation in miniKanren (listing 2)

Listing 2. Logic $\top$ and $\bot$ in miniKanren

```
(define succeed (== #f #f))
(define fail (== #f #t))
```

The two roles of unification are coming from the extended substitution set (S), assigned a value to a variable, via continuation on *lambdag@* and the return value, produced a truth value to a goal, of *lambdag@* (listing 1). In propositional logic, we give the unification two values instead of a variable and a value; hence, we are using it to produce the truth value of a logical goal only. We generalize the example to represent a propositional logic in miniKanren, we use a nullary goal function.

Let us recall a few examples of the closed world assumption (CWA) using propositional logic. Consider the following logic program,

```
p :- q.
```

Clingo returns p, q is false, or $\emptyset$ as a result. The program's miniKanren counterpart is as follows,

```
(define (p) (q))
```

However, this program cannot produce a result in miniKanren. To run this program in miniKanren, the user has to explicitly state the CWA rules **q :- false.** in miniKanren.

```
(define (q) fail)
```

With the complete program,

```
(define (p) (q))
(define (q) fail)
```

miniKanren is able to produce the expected result.

In predicate logic, we give the unification of one variable and one value; hence, we are using it to perform two roles, unify a variable with a value and produce the truth value of a logic goal. We generalize the example to represent a predicate logic in miniKanren, we use a non-nullary goal function. Similarly, if a predicate is not showing in any rule's header, we need to define CWA specifically as well. An example logic program,

```
p(X) :- q(X).
```

will have its miniKanren counterpart as follows,

```
(define (p x) (a x))
(define (a x) fail)
```

Since in this case, we are not using unification to unify our variable with any value, it is essentially the same as propositional logic. In contrast, the CWA is implicitly handled by miniKanren if a predicate is in the rule's header, but the value is not. The example logic program,

```
p(X) :- a(X).
a(X) :- X=1..2.
```

has the same representation in miniKanren,

```
(define (p x) (a x)
(define (a x) (conde [(== x 1)] [(== x 2)]))
```

The user does not need to supplement any rules for CWA. A query like "(run 1 (q) (p 4)" gives us "()".

As we can see, variables and quantifiers drastically increase the expressiveness so that the user gets rid of writing CWA goals explicitly unless the goal does not show up in any rule's header. However, variables and quantifiers also increase the solving difficulty. Hence we tend to leave out variables first and add them back later. That is also why stable model semantics (definition 2.15, step 1) and bottom-up solving approach (section 2.3) prefer to create a propositional image from predicate rules through grounding (definition 2.9).

For instance, let us ground the previous predicate program into a propositional image in miniKanren. The idea is simple, we simply concatenate the predicate name with the variable name and the value it can take to get a new predicate name without any variables. The grounded logic program,

```
pX1 :- aX1. pX2 :- aX2.
aX1. aX2.
```

has the same grounded representation in miniKanren,

```
(define (px1) (ax1)) (define (px2) (ax2))
(define (ax1) succeed) (define (ax2) succeed)
```

It easily shows the drawback of grounding which leaves a heavy memory footprint.

To the best of our knowledge, only a few attempts have been made to add negation to miniKanren. The constructive negation introduced by Evgenii Moiseenko built upon universally quantified disequality constraints only works for the stratified normal program [23]. Also, the semantics of negation is more like a filter, where the solver executes the goal inside the negation, then gets a differential set between the atoms before and after the negated goal, and eventually subtracts the differential set from the original set. Their semantics do not handle the non-stratified normal program like the stable model semantics we used in stableKanren. Regarding the input program being not fully declarative, Moiseenko presented an example where the negation operator applies to an unbounded free variable;

```
(run 1 (q) (noto (== q 1)) (== q 0))
```

We consider such an unbounded variable unsafe (definition 3.2), but can be resolved by compilation time rewriting (through a macro) so that the rule follows the desired format (definition 2.3);

```
(run 1 (q) (== q 0) (noto (== q 1)))
```

Hence the variable is bounded and safe to use. If such an unsafe variable is a body variable quantified by $fresh$, stable model semantics consider the variable should be groundable. Whether finite or infinite, each variable should be able to get a set of values under a positive goal. Therefore, the forall operator has a domain that it can iterate through. Our transformation in section 3.5 covered such cases.

We believe that we could give the negation a more widely accepted semantics by integrating the stable model semantics based on miniKanren. So that we can produce an answer for a problem like the two-person game [27], [23] in section 3.10.

3 OUR APPROACH AND METHODOLOGY

In this section, we present our system, stableKanren [1], which extends miniKanren with stable model semantics. Instead of describing in pseudo-code, we present our algorithms in Scheme [10] so that the reader can verify the implementations directly in Scheme[2]. We build our solver on top of the core miniKanren[3]. The core miniKanren only integrates the essence of Prolog [9], and our extension adds negation support under stable model semantics.

From stable model semantics (definition 2.15), we derive the following relationship between the stable model of Π and the minimal model of the reduct Π^M.

Definition 3.1 (stable model property)**.** Let Π be a propositional program and M be an interpretation. M is a stable model of Π if M is the minimal model of the reduct Π^M.

Stable model semantics consider a minimal model of the corresponding reduct program to be one of the program's stable models; therefore, we build our solver to find a minimal model and the reduct program simultaneously. The underlying resolution process guarantees to produce a minimal model as long as the input program can be handled. The key idea of the reduct (in stable model semantics) and our algorithms is the negative goals work as a checker or remover while the positive goals work as a generator as usual. In step 3 of the stable model semantics (definition 2.15), the reduct created from the interpretation removes the loop and the negation completely so that the program can be handled by using traditional Prolog to produce the minimal model ([14], Remark 2). Then the interpretation needs to be verified that it is the minimal model of the reduct program.

[1] https://github.com/stable-Kanren/stable-Kanren
[2] https://github.com/cisco/ChezScheme
[3] https://github.com/miniKanren/simple-miniKanren

Unlike the bottom-up grounding and constraint-propagation approach, where the negations are ignored until the truth value propagation phase; we dealt with negation as soon as we encounter one. Therefore, we want to make sure the variable is safe (definition 3.2) before the negative goal.

Definition 3.2 (variable safety)**.** The positive goals always ground all variables before calling any negative goals; otherwise, the variable is unsafe to use in negative goals.

In this paper, our algorithms focus on the normal program clause and its stableKanren counterpart as defined in definition 2.3. We assume all input logic programs and their stableKanren equivalent are normal programs as defined in definition 2.4, and so from now on, we refer to normal programs and normal program clauses as simply programs and clauses. Following the input format guarantees the variables are safe in the negative goals for most cases. There are some measures to ensure the variable safeness, but we leave them as future work and we assume the variables are safe in this paper.

We have overcome a few challenges to solve normal programs, and we present them here. Originally, resolution has only one role (definition 2.20) and unification plays two roles (definition 2.21). A positive goal is proven to be true iff all of its variables are successfully unified, and a positive goal is grounded iff all of its variables are unified with a value. To get a stable model, we would like to have our algorithm grant more roles to resolution and unification. Both resolution and unification require changes to support the loop and negation introduced by the normal program. We are showing the changes we made in the following sections. For resolution, it has four more roles, distributing negation to the unification level (section 3.3, 3.5), producing truth value for the loop (section 3.7), getting the domain of a variable (section 3.6), and continuing execution after getting a partial result (section 3.9). For unification, it has to produce the truth value for negated unification (section 3.2). Any non-functional language can implement our algorithm, but the reader will see that our approach of utilizing macros and continuations is not only concise but it is also easy to understand, extend, and experiment with different optimization algorithms in the future. Those functional programming traits provide great aids in tackling the challenges mainly from dynamic features after introducing negation to predicate logic, like complementing rules (section 3.3, 3.5), obtaining fresh variables domain (section 3.6), iterating all values (section 3.6), and handling non-monotonic results (section 3.8). More challenges remain unsolved and we will discuss them in the future work section.

To support solving negative goals, we introduced a set of new macros *noto*, *defineo*, $\overline{conde}$, $\overline{fresh}$, $conde^t$, and $fresh^t$, etc. Some macros like *defineo*, $conde^t$, and $fresh^t$ are not completed in one section and will get expanded in later sections.

3.1 Meeting with new friends

Before we jump into our macros and algorithms, let us make some fundamental changes to the internal continuation of *lambdag@* (listing 1) and introduce the negation operator. We need two auxiliary variables along with the substitution in *lambdag@* to help us solve normal programs; a *negation counter (n)* and a *call stack frame (cfs)*.

Definition 3.3 (negation counter)**.** A negation counter records how many negations we have encountered during the resolution so far, where an even number means a positive goal and an odd number means a negative goal.

Definition 3.4 (call stack frame)**.** A call stack frame (CFS) records the goal functions we have invoked during the resolution so far, so the starting goal is at the bottom of CFS and the current goal is at the top of CFS.

We place the new *n* and *cfs* to the internal continuation of *lambdag@* in listing 3.

Listing 3. Extended lambdag@ with n and cfs

```
(define-syntax lambdag@
  (syntax-rules (:) ((_ (n cfs S) e ...) (lambda (n cfs S) e ...))))
```

To represent the negation operator, we add one new macro, *noto* (listing 4), which we picked because *not* is already taken as a Scheme keyword.

Listing 4. New macro noto

```
(define-syntax noto
  (syntax-rules ()
    ((noto (name params ...))
     (lambdag@ (n cfs S) ((name params ...) (+ 1 n) cfs S)))))
```

The *noto* operator simply increases the negation counter, *n*, by one as shown above, then passes the new *n* to the next continuation. We also need to modify the rest of the goal functions in the system, including *run*, ==, $fresh$, *conde*, *bind*, and *bind*$*$. Mainly for the goal functions defined by *lambdag@* or calling the goal function, we need to add a negation counter and a cfs to meet the contract between continuations.

Listing 5. Unification with updated lambdag@

```
(define (== u v)
  (lambdag@ (n cfs s)
    (cond [(unify u v s) => (lambda (s+) (unit s+))]
          [else (mzero)])))
```

3.2 Upside down

As we have mentioned originally, unification has two roles (definition 2.21) and we have shown that before negation the variables are bounded already, therefore unification only assigns truth value to a negative goal. Also, for resolution, we want to retain its nice property of producing a minimal model. Hence, we reuse the existing resolution process to prove the negative goal "not g" is true. We deal with the base case first, where negation directly applies to the unification, and we will discuss how to distribute the negation from a higher-level goal to this base case through resolution in section 3.3.

We modify the naive unification (definition 2.19) to produce a truth value under negation. If the negation directly applies to naive unification, the truth value is determined by the opposite of the naive unification outcome. We added a *negation counter* to lambdag@ (listing 3) to allow the goal function to handle both positive and negative goals. So our algorithm only modifies the unification (listing 5) outcome to the negated unification (listing 6).

Listing 6. Negated unification

```
(define (== u v)
  (lambdag@ (n cfs s)
    (if (even? n)
      (cond [(unify u v s) => (lambda (s+) (unit s+))]
            [else (mzero)])
      (cond [(unify u v s) => (lambda (s+) (mzero))]
            [else (unit s)]))))
```

In this modified unification, under the negation branch, a failed unification becomes successful returning an unextended substitution set, and a successful unification fails by terminating the search stream. Furthermore, we can introduce the *inequality constraint* for the failed unification, so the unbounded variable will have a list of values it cannot be bound with. We are leaving this inequality constraint as future work. When negation does not apply to unification directly, we need to design an algorithm (macro) to distribute it down through the resolution.

3.3 Down to the base

From the resolution (definition 2.16), we know that it is always trying to prove a goal to be true. We decided to reuse the existing resolution process to prove negative goal "not g" is true and we showed one part of the semantics can be achieved by modifying the unification process to negated unification as a base case in section 3.2. The inductive case has to be handled by resolution in this section. As long as the resolution can distribute the negation to the base case (unification level), we complement a normal program into a format that can be handled by resolution.

From definition 2.4, we know that a goal "g" could match with multiple statements' (rules) heads, and if any of those statements' bodies can be proven to be true, then "g" is also proven to be true. So we use disjunction (*conde*) to connect different rule's bodies under the same rule's head, and each rule's body is a conjunction of literals. Hence, a goal function's body is in *Disjunctive Normal Form (DNF)*, and in our stableKanren representation, each positive goal function has at most one *conde* operator inside. We complement the propositional rule's body first, as it is simpler to deal with. We will expand our transformer to support predicate rules in section 3.5. The following example program "p" is written in stableKanren;

Listing 7. A goal function with DNF body

```
(define (p) (conde [(q) (r)] [(s) (t)]))
```

For propositional rules, DeMorgan's law is sufficient to distribute the negation to the unification level. After the conversion, the complemented goal function's body is in *Conjunctive Normal Form (CNF)*. So, our stableKanren representation is a conjunction of multiple *conde*s with negation distributed to each atom.

Listing 8. A complemented goal function with CNF body

```
(define (not-p)
  (conde [(noto (q))] [(noto (r))])
  (conde [(noto (s))] [(noto (t))]))
```

We use two macros, $\overline{conde}$ (listing 9) and $conde^t$ (listing 10), to implicitly create the complement process we described.

Listing 9. New macro complement-conde

```
(define-syntax complement-conde
  (syntax-rules (conde)
    ((_ (conde (g0 g ...) (g1 g^ ...) ...))
     (conde-t (g0 g ...) (g1 g^ ...) ...))
    ((_ (g0 g ...))
     (conde-t (g0 g ...)))))
```

The macro $\overline{conde}$ replaces each *conde* in the clause with the $conde^t$. If there is no *conde* in the clause, but only a conjunction of sub-goals, then we are treating this as a special case of the *conde* (still a DNF), and the $\overline{conde}$ simply adds one $conde^t$ to it.

Listing 10. New macro conde-t

```
(define-syntax conde-t
  (syntax-rules ()
    ((_ (g0 g ...) (g1 g^ ...) ...)
      (fresh () (conde [g0] [g] ...)
                (conde [g1] [g^] ...) ...))))
```

The macro $conde^t$ complements all clauses in the body from DNF to CNF. Notice here, unlike the "not-p" example we showed in listing 8, there is no 'noto' operator in our macro template. The reason is we are wrapping "p" and "not-p" as one goal function in section 3.4 using macro 'defineo' (listing 11) and complement only invokes under a negation scenario where the negation counter (definition 3.3) is an odd number. Therefore, there is always an implicit negation in our $\overline{conde}$ and $conde^t$ macros, and 'g0', 'g', 'g1' work as 'noto g0', 'noto g', 'noto g1', etc.

3.4 United as one

As Clark has pointed out, it is appropriate to regard the completion of the normal program, not the normal program itself, as the prime object of interest when dealing with negation [5]. Even though a programmer only gives a system the normal program, the normal program is completed by the system and what the programmer is actually programming with is the combination of the two.

We have shown a simple completion of propositional rules in section 3.3. Now, we want a goal function that can handle two types of goals, namely the positive goal and the negative goal. We define a macro *defineo* (listing 11) so that the user only needs to define the original rules and this macro will implicitly create a goal that has the user's original rules and our negated complement rules.

Listing 11. New macro defineo

```
(define-syntax defineo
  (syntax-rules ()
    ((_ (name params ...) exp ...)
     (define name (lambda (params ...)
       (lambdag@ (n cfs s)
         ((cond
           [(even? n) (fresh () exp ...)]
           [(odd? n) (complement-conde exp ...)]) n cfs s)))))))
```

The *defineo* will define a goal function that combines the original rules *exp*... with the complement rules *complement-conde exp* .... During execution, this goal function picks the corresponding rule set based on the value of the negation counter. If n is even, use the original rules, and if n is odd, use the complement rules.

3.5 Stretch out

We have laid a good framework, which allows us to add more features and continuously evolve our algorithm. Let us advance from propositional logic to predicate logic, this process introduces variables and corresponding quantifiers (∃, ∀) to the logic statement as we have shown in section

2.2. In logic programming, there are two types of variables in each rule. The *head variable* (definition 2.5) and the *body variable* (definition 2.6). Remember that head variables in a negative goal are always bounded, as said at the beginning of section 3. Therefore, to simplify the discussion, as a good starting point, we focus on the case with head variables only. We deal with body variables in section 3.6. Given the logic statement;

$$H(X,Y) \leftarrow \exists X,Y(B_1(X,Y) \wedge B_2(X) \wedge B_3(Y))$$

And we know that under negation variables X, Y in the above rule always have values x, y. So we safely drop the $\exists$ quantifier and focus on the rule's body transformation with assigned values to prove $\neg H$ is true. We can obtain the propositional form of the representation as follows;

$$\neg H^{x,y} \overset{?}{\leftarrow} transform(B_1^{x,y} \wedge B_2^x \wedge B_3^y)$$

It would seem that we can still distribute negation over the rule's body using DeMorgan's law, just like we did in section 3.3;

$$\begin{aligned} &\neg H^{x,y} \leftarrow \neg(B_1^{x,y} \wedge B_2^x \wedge B_3^y). \\ \overset{?}{=}\ &\neg H^{x,y} \leftarrow (\neg B_1^{x,y} \vee \neg B_2^x \vee \neg B_3^y). \end{aligned}$$

However, the transformation does not produce the expected result for $\neg H(X,Y)$. Let's say $B_1(X,Y)$ unifies X with one of the values $\{1,2\}$, and $B_2(X)$ unifies X with one of the values $\{2,3\}$. The original positive rule evaluated $H(X,Y)$ as $H(2,Y)$, where X could only unify to 2. Therefore, the complement goal $\neg H(X,Y)$ is expected to be true for $X \neq 2$. However, using the rule complemented by DeMorgan's law, we are getting $\neg H(X,Y)$ is expected to be true for $X \neq \{1,2,3\}$. To fix this issue, we need a new approach other than DeMorgan's law to transform the rule.

As the resolution is trying to prove each sub-goal one by one in each rule (statement), to prove "$\neg H$" to be true, we know "H" is failing somewhere in the rule's body among one of the rules. For a rule's body, each sub-goal could fail, and when a sub-goal has failed, then the prior sub-goals must have succeeded. To capture this property, the transformation of it should be a disjunction of the negation to each sub-goal in conjunction with all sub-goals before the current one. Furthermore, we noticed that each sub-goal works as a checker and the values are checked by unification independently inside the sub-goal could fail. Therefore, the transformation not only applies to the sub-goal but also to each variable so that we can distribute the negation to the unification level.

$$\neg H \leftarrow \neg B_1^x \vee (B_1^x \wedge \neg B_1^y) \vee (B_1^x \wedge B_1^y \wedge \neg B_2^x) \vee (B_1^x \wedge B_1^y \wedge B_2^x \wedge \neg B_3^y)).$$

In general, each B_n^v is a goal function in stableKanren, so we can implement the above transformation as a macro $fresh^t$;

Listing 12. New macro fresh-t

```
(define-syntax fresh-t
  (syntax-rules ()
    ((_ (x ...) g0) g0)
    ((_ (x ...) g0 g ...)
      (conde [g0]
             [(fresh ()
                (noto g0)
                (fresh-t (x ...) g ...))]))))
```

It is a recursive process that iterates through all sub-goals and unifications with distributive law. Once again, $fresh^t$ is working as "not exist", so the negation counter carries an implicit negation

with an odd number during the runtime. Hence, $g0$ means *noto g0*, solving the negative goal, and *noto g0* means $g0$, solving the positive goal. We introduce the following new macro $\overline{fresh}$ to implicitly create the complement body rule;

Listing 13. New macro complement-fresh

```
(define-syntax complement-fresh
  (syntax-rules (fresh)
    ((_ (fresh (x ...) g0 g ...))
      (fresh (x ...)
        (fresh-t (x ...) g0 g ...)))
    ((_ g0 g ...) (fresh-t () g0 g ...))))
```

When there are not just head variables in the rule, but also body variables introduced by $fresh$, the macro $\overline{fresh}$ replaces each $fresh$ in the rule with $fresh^t$. These unbounded body variables require additional changes in our transformation (section 3.6), but we can treat them the same way to simplify our transformation for now. When there is no $fresh$ in the rule, which means the rule only has head variables, then we are treating this as a special case of $fresh$, and $\overline{fresh}$ simply adds one $fresh^t$ to it. We also need to replace the *conde* with $\overline{fresh}$ in our $conde^t$ macro (listing 10). Eventually, DeMorgan's law got replaced with our transformer to the predicate program as follows;

Listing 14. Extended conde-t with complement-fresh

```
(define-syntax conde-t
  (syntax-rules ()
    ((_ (g0 g ...) (g1 g^ ...) ...)
      (fresh ()
        (complement-fresh g0 g ...)
        (complement-fresh g1 g^ ...) ...))))
```

3.6 Code on the fly

In this section, we deal with body variables introduced by $fresh$ (∃) in the rule's body. Once we finish the transformation we introduced in section 3.5 of the original rule, all ∃ quantifiers over body variables in the rule are turned into ∀ quantifiers. In the beginning, all body variables are unbounded. An unbounded body variable does not impact our transformation since the negated unification (section 3.2) returns false on an unbounded variable and forces resolution to choose another path (naive unification) to bind a value to that variable. We use the concept of a *generator* to represent the first naive unification that unifies the unbounded body variable to a set of values. We have two issues concerning bounded body variables when resolution reaches a generator: getting the domain of a body variable and iterating over all values using resolution.

The domain is based on the resolution context and resolution has to change the course from finding one answer (∃) to finding all answers (∀). Hence, we need to construct a new program using the resolution context at runtime to achieve our goal. We introduce a set of macros handling these issues. We modify $fresh^t$ (listing 12) in listing 15 to find the generator of the body variable, then we get the domain of the variable by assuming the domain space is finite, and finally we iterate through the domain and create a conjunction search stream over all possible values.

Listing 15. Extended fresh-t with forall

```
1 (define-syntax fresh-t
```

```
2   (syntax-rules ()
3     ((_ (x ...) g0) g0)
4     ((_ (x ...) g0 g ...)
5       (conde
6         [g0]
7         [(lambdag@ (n cfs s)
8            ((fresh ()
9               (noto g0)
10              (lambdag@ (nn ff ss)
11                (let* ([diff (- (length ss) (length s))]
12                       [ext-s (get-first-n-elements ss diff)]
13                       [argv (list x ...)]
14                       [b-vars (find-bound-vars argv ext-s)])
15                (if (null? b-vars)
16                  ((fresh-t (x ...) g ...) nn ff ss)
17                  (((forall (x ...) (g ...) b-vars)
18                    (domain-values g0 b-vars cfs s)) n cfs s))))
19             ) n cfs s))])))))
```

In line 7, we are preserving the substitution before executing $g0$. After executing $g0$, we get a new substitution in line 10. From lines 11 to 14, we compute the difference between the lengths of the two substitutions, and we are using the difference to get the delta of substitutions after executing $g0$. We check if any new body variables have been bound to a value. If no variable got the value, then we keep running future sub-goals (g ...) as normal in line 16. Otherwise, we obtain all values of the variables and check that all future sub-goals (g ...) can be proven true for all values of bounded vars in lines 17 and 18.

The domain of body variables is fetched through the *domain-values* macro (listing 16).

Listing 16. New macro domain-values

```
(define-syntax domain-values
  (syntax-rules ()
    [(_ g0 bounded-vars cfs s)
      (take #f (lambdaf@ ()
        ((fresh (tmp) g0 (== tmp bounded-vars)
           (lambdag@ (f_n f_c final_s)
             (cons (reify tmp final_s) '())))) 0 cfs s)))]))
```

This macro uses the generator $g0$, bounded variables, current CFS, and current substitution to construct an internal program to generate all values. The *take #f* is the underlying implementation of the *run∗* interface; the reader can refer to the *run∗* implementation to learn more details [9]. We reset the negation counter to 0 to clearly state that we are using positive rules to get the domain.

The conjunction search stream over all possible values is created by the $forall$ (listing 17).

Listing 17. New macro forall

```
(define-syntax forall
  (syntax-rules ()
    [(_ (x ...) (g ...) vars)
```

```
(let ([var-list (remove-var-from-list (list x ...) vars)])
(define (iterate-values values)
  (lambdag@ (n cfs s)
    (if (null? values)
      (unit s)
      (inc (bind* n cfs
        ((fresh-t (var-list) g ...)
         n cfs (ext-s-forall vars (car values) s))
        (iterate-values (cdr values)))))))
iterate-values)]))
```

It is a recursive process that iterates through all values of the bounded body variables and creates an incomplete stream (inc) of conjunction (bind*) over the future sub-goals under the current value and the future sub-goals under other values. It has to remove bounded body variables from the variable list "x ..." so that the future $fresh^t$ can properly detect other unbounded variables that are getting bounded.

We could not easily complete such a task to code on the fly without using these traits from functional language. The three macros may not look very elegant to experienced functional programmers, but they are simpler and more concise than non-functional implementations.

3.7 Not a strange loop

In this section, we deal with the loops (definition 2.18) we encounter during resolution. We believed that when the resolution reaches a loop point instead of a unification, the resolution has to produce a truth value without unification but via coinduction. Also, our algorithm needs to ensure that resolution handles the loop without creating a reduct beforehand. We categorize the loop as either a *positive loop* or a *negative loop* before introducing the *coinductive resolution*.

Definition 3.5 (positive loop)**.** When a call is in a positive loop, there is no negative goal involved on the CFS.

Definition 3.6 (negative loop)**.** When a call is in a negative loop, there is at least one negative goal involved on the CFS.

We handle positive loops (definition 3.5) first, and negative loops (definition 3.6) will be handled later. The positive loop encountered during resolution produces false as the truth value. It shall return false due to the *rationality principle*: one shall not believe anything until one is compelled to believe. Therefore, it shall end the current resolution, so the resolution can try other possible paths to find a fact to unify with. This also meets the property of minimal model semantics.

Using the negation counter (definition 3.3), we can further define *odd negative loop* and *even negative loop* from definition 3.6.

Definition 3.7 (odd negative loop)**.** When a call is in an odd negative loop, there are an odd number of negative goals involved in the loop.

Definition 3.8 (even negative loop)**.** When a call is in an even negative loop, there are an even number of negative goals involved in the loop.

The negative loop encountered during resolution produces truth values as follows. If we see an odd negative loop, we return false. If we see an even negative loop, we return a choice of true or false.

Overall, our coinductive resolution has three types of loops to handle, a positive loop, an even negative loop, and an odd negative loop. We are using *calling frame stack (CFS)* (listing 3) to handle the loop scenario with coinductive resolution. Unlike tabling miniKanren, another extension of core miniKanren, where the solver records the result globally [3], our CFS uses runtime information to prevent proving the same goal multiple times. The record we stored on the CFS consists of a *signature* and *n* (the value of the negation counter). The *signature* is the goal function name with parameters grounded to the bounded value. If the parameter has no bounded value, we use the parameter name as part of the signature. During runtime, a signature must be an exact match to the existing signature on the CFS to show we are in a loop. We extended our *definео* macro (listing 11) to use this CFS to determine a loop in listing 18.

Listing 18. Extended defineo with loop handling

```
1  (define-syntax defineo
2    (syntax-rules ()
3      ((_ (name params ...) exp ...)
4        (define name (lambda (params ...)
5          (let ([argv (list params ...)])
6          (lambdag@ (n cfs s)
7            (let* ([args (map (lambda (arg) (walk* arg s)) argv)]
8                   [signature (list `name args)]
9                   [record (seen? signature cfs)])
10             (if (and record #t)
11               (let ([diff (- n (get-value record))])
12               (cond [(and (= 0 diff) (even? n)) (mzero)]
13                     [(and (= 0 diff) (odd? n)) (unit s)]
14                     [(and (not (= 0 diff)) (odd? diff)) (mzero)]
15                     [(and (not (= 0 diff)) (even? diff))
16                       (choice c mzero)]))
17               ((cond [(even? n) (fresh () exp ...)]
18                      [(odd? n) (complement exp ...)])
19        n (expand-cfs signature n cfs) s))))))))))
```

In line 5, we obtain a list of parameter variables, and we try to see if each variable is bound to a value or not in line 7. If the variable has a substitution, it will be replaced by a value, otherwise, it will be the parameter's name. In line 9, we check if we have encountered the signature during resolution or not. If we have encountered the signature on CFS (line 10), then we use the difference of the negation counter value to find out the loop type in line 11. From lines 12 to 13, the difference is 0, which means we had a positive loop, so minimal model semantics apply here. The return value depends on the negation counter before the positive loop. According to minimal model semantics, all atoms inside the positive loop are evaluated as false, unless there are other ways to break the loop, and the negation over the positive loop is evaluated as true. From lines 14 to 16, the difference is not 0, which means we had a negative loop, so stable model semantics apply here. We return false if it is an odd negative loop and return a choice of true or false if it is an even negative loop. If we do not see any loop, we are solving the goal as we talked about in the previous section (listing 11) in lines 17 to 19. Notice that, in line 19, we are expanding the CFS (listing 19) while solving the goal.

Listing 19. A function to expand CFS

```
(define (expand-cfs k v cfs)
  (adjoin-set (make-record k v) cfs))
```

There are some interesting consequences of our handling of loops and negations, based on the properties of stable model semantics. As we have mentioned in section 2.2, unlike monotonic reasoning which has only one minimal model, non-monotonic reasoning defined by stable model semantics could have three outcomes, no model, one model, or multiple models. Each outcome corresponds to one kind of loop we handled in this section. In the beginning, we assumed there was one model and the resolution did not reach any contradiction or splitting. A positive loop causes the reduct program to produce a minimal model smaller than the given interpretation unless the positive loop has a fact to unify with. It does not change the number of models. An odd negative loop caused the reduct program to reach an unusable case (contradiction). If all reduct programs, created from all interpretations, are unusable, then the program becomes unsatisfiable and has no model. An even negative loop leads to two models (splitting).

Let us take a look at the difference between an empty set as one stable model and no stable model. We currently cannot distinguish the two as they both return "()" in stableKanren. The user needs to run the query on the opposite goal to identify. If the positive query returns "()" (false), but the negative query returns "(_.0)" (true), the program has one model with the truth value of the query goal as false. If the positive query returns "()" (false), and the negative query also returns "()" (false), the program has no model, and the query goal has no truth value since the program is unsatisfiable. We will add an output "unsatisfiable" in our future work, so the user does not need to verify by themselves.

For example, consider the following program

```
a :- b.                          (defineo (a) (b))
b :- c.                          (defineo (b) (c))
c :- a.                          (defineo (c) (a))
```

This program has an empty set as the stable model. The query "(run 1 (q) (a))" gives "()" (false), but the query "(run 1 (q) (noto (a)))" gives "(_.0)" (true).

The no model (unsatisfiable) case is demonstrated in the following program

```
a :- not b.                      (defineo (a) (noto (b)))
b :- not c.                      (defineo (b) (noto (c)))
c :- not a.                      (defineo (c) (noto (a)))
```

This program is unsatisfiable and has no models. Both the query "(run 1 (q) (a))" and the query "(run 1 (q) (noto (a)))" gives "()" (false).

So the positive loops do not create any more issues with the number of models we got, but the negative loops have a contradiction and a splitting issue we need to handle. We will discuss the splitting issue in section 3.8 and the contradiction issue in section 3.9.

3.8 Confine the multiverse

In this section, let us confine the splitting universe created by the even negative loop. Consider the following logic program and its stableKanren representation

```
a :- not b.                      (defineo (a) (noto (b)))
b :- not a.                      (defineo (b) (noto (a)))
```

This program has two answer sets, {a} and {b}. However, simply being able to produce different answer sets is insufficient. We need to maintain the partial result as well. Since, unlike the definite program, the partial result affects the future search process in the normal program. As we can see {a} and {b} are mutually exclusive. We must consider these cases in stableKanren so that queries like, "(run 1 (q) (a) (b))", do not return true.

We have shown that resolution may visit an even negative loop from different entry goals and get different outcomes for the same goal, we need to save the result we obtained in the first place. The truth value produced by an even negative loop shall be consistent iff the partial result is locally tabled on the *partial answer set (PAS)*.

We combine PAS (p) with the existing substitutions (s) as a combined pair (c) on the internal continuation of *lambdag@* (listing 20) as we want p passing through goal functions just like s.

Listing 20. Extended continuation of lambdag@ with PAS

```
(define-syntax lambdag@
  (syntax-rules ()
    ((_ (n cfs c) e ...)
      (lambda (n cfs c) e ...))
    ((_ (n cfs c : s p) e ...)
      (lambda (n cfs c)
        (let ([s (car c)] [p (cadr c)]) e ...)))))
```

In the updated *lambdag@*, We added one more macro pattern to break the combined pair c into s and p so that they can be accessed by the expression e inside the template later. Similar to the changes we made in section 3.1, we modify the rest of the macros has *lambdag@* involved like, *run*, ==, $fresh$, *conde*, *bind*, and $bind*$ to meet the contract between continuations. We show how to use this updated *lambdag@* in extended $defineo$ (listing 21);

Listing 21. Extended defineo with partial answer set

```
1  (define-syntax defineo
2  ;;; omit macro pattern and template
3    (lambdag@ (n cfs c : s p)
4     (let* (;;; omit variables assignment
5            [res (seen? signature p)]
6            [sign (+ n (get-value res))])
7      (cond [(and res (even? sign)) (unit c)]
8            [(and res (odd? sign)) (mzero)]
9      (else
10       (if (and record #t)
11       ;;; omit loop handling
12       ((cond [(even? n) (fresh () exp ... (ext-p `name argv))]
13              [(odd? n) (complement exp ... (ext-p `name argv))])
14        n (expand-cfs signature n cfs) c))))))))))))
```

Unrelated details are omitted in this code snippet to minimize disturbance. The key idea is that resolution checks the local tabling to see if it computed the result *res* in line 5, and the return value in lines 7 and 8 depends on the sign of the value we stored. Lastly, the local tabling is updated by "ext-p" before we successfully return from a goal in lines 12 and 13.

We implicitly append an extra updating step "ext-p" (listing 22) after each predicate definition.

Listing 22. A function to update PAS

```
(define (ext-p name argv)
  (lambdag@(n cfs c : S P)
    (let ((key (map (lambda (arg) (walk arg S)) argv) ))
      (list S (adjoin-set (make-record (list name key) n) P)))))
```

After we finish the predicate proving process, the updating step will update the context environment, therefore allowing a later proving process to get information from the context to know if the current goal has been proved or not. If the goal has been proved before, we can reuse it without proving it again.

Let us go back to the example, "(run 1 (q) (a) (b))". During the proving process for "a", we proved "b" to be false, so when "a" is proved to be true the system knows that it cannot prove "b" to be true, since the context maintained the partial result we obtained from proving "a". Without retaining this partial result context, the system will treat "b" as a fresh new goal and produce true for it, which is not what we would expect to see.

Currently, we only output the values bound to the query goals. The user may see the same values produced multiple times since they may come from a different answer set or a different path of proving. We can output the complete answer set we stored on PAS (listing 20) as auxiliary information in future work.

3.9 Do not stop

There is one more thing that shows the specialty of the normal program under stable model semantics requires us to treat the program as an integral part. Unlike the definite program, the partial result of the normal program may not be the final result since the odd negative loop may create an *unavoidable contradiction.*

Definition 3.9 (unavoidable contradiction)**.** An unavoidable contradiction means there is an atom in a normal program that neither can be proven true nor false due to the odd negative loop always creating a contradiction.

Even though we compute the partial minimal model and the corresponding partial reduct program, we cannot guarantee the partial result is part of the final answer due to the missing *optimal substructure* property in NP-Hard problems. It is possible that there is an odd negative loop in other parts of the program that will create an unavoidable contradiction based on the given partial result.

For example, consider the following program

```
a.                                  (defineo (a) succeed)
b.                                  (defineo (b) succeed)
p :- a, not p.                      (defineo (p) (a) (noto (p)))
```

If we have the query "(run 1 (q) (a))", and we only consider the partial result, then the partial result will be that "a" is true. However, for the normal program, we need to consider the contradiction imposed by the rule containing an odd negative loop. In this case, the rule "p :- a, not p." causes the program to be unsatisfiable. Under stable model semantics, we are making sure each atom will get an assignment of either true or false no matter if it appears in the query or not. This ensures that after we get the partial result "a", the system will also check for "b" and "p". In this case, "b" will be added to the partial result, but we will get contradictions for "p" and "not p", and

eventually this contradiction on both sides causes the system to overturn all partial results we obtained (unsatisfiable).

Therefore, the resolution should continue checking all untouched rules and values, even after finishing the given query goals. Currently, we are using some bottom-up ideas to perform this task to make sure we check all values and rules after the initial resolution has finished proving query goals. We record all rules defined by *defineo* in a "program-rules" set. Important notice here, as we are treating all rules as a whole program, the user needs to remember to "reset-program" between different programs.

Then we modify the "run" interface (listing 23).

Listing 23. An updated run interface

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (lambdaf@ ()
         ((fresh (x) g0 g ...
          (lambdag@ (negation-counter cfs c : S P)
            (if (null? (take 1 (lambdaf@ ()
                    ((check-all-rules program-rules x)
                      negation-counter cfs c))))
              (mzero)
              (cons (reify x S) '()))))
          negation-counter call-frame-stack empty-c))))))
```

At the end of the original query goals "g0 g ...", instead of using "(reify x S)" to produce the final result, we invoke "check-all-rules" (listing 24) to check our untouched rules against unavoidable contradictions. If there is an unavoidable contradiction, the whole program is unsatisfiable; otherwise, we find at least one valid final result.

Listing 24. A function to check all rules

```
(define (check-all-rules rules-set x)
  (lambdag@ (n cfs final-c : S P)
    (let ((rule (fetch-rule rules-set)))
      (if (and rule #t)
        (let*
          ([goal (get-key rule)]
           [arity (get-value rule)]
           [vals (get-values goal (construct-var-list arity))])
          (bind* n cfs
            ((check-rule-with-all-values goal vals) n cfs final-c)
            (check-all-rules (cdr rules-set) x)))
        (cons (reify x S) '())))))
```

In this function, we obtain all values of a rule's head, then run resolution on both positive and negative goals for each value using "check-rule-with-all-values" (listing 25).

Listing 25. A function to check a rule with all values

```
(define (check-rule-with-all-values rule values)
  (lambdag@(_ cfs c)
    (if (null? values)
      (unit c)
      (inc
        (mplus*
          ; check positive goal, negation counter = 0
          (bind* 0 cfs
            ((apply (eval rule) (car values)) 0 cfs c)
            (check-rule-with-all-values rule (cdr values)))
          ; check negative goal, negation counter = 1
          (bind* 1 cfs
            ((apply (eval rule) (car values)) 1 cfs c)
            (check-rule-with-all-values rule (cdr values)))
        )))))
```

Eventually, the resolution traversed all values if there is no contradiction showing on both positive and negative goals at the same time, and so we have the final answer. This part can also be done in a purely top-down approach, which we leave as future work.

The unsatisfiability comes from having a negation rule such that it cannot be proven true or false. This can only be avoided by adding some external supporting rule like "p :- b." to the program, so that we will be able to prove "p" to be true bypassing (avoiding) the odd negative loop, thus resolving the unavoidable contradiction. The new program will be

```
a.
b.

p :- a, not p.
p :- b.
```

```
(defineo (a) succeed)
(defineo (b) succeed)
(defineo (p) (conde
               [(a) (noto (p))]
               [(b)]))
```

Eventually, leading the system to maintain the partial result we obtained.

3.10 Back to the game

Now, let us go back to the two-person game example [27], which Moiseenko's constructive negation was unable to handle [23].

```
edge(a, b).
edge(b, a).
edge(b, c).
edge(c, d).

win(X) :- edge(X, Y),
    not win(Y).
```

```
(defineo (edge x y)
  (conde
    [(== x 'b) (== y 'c)]
    [(== x 'a) (== y 'b)]
    [(== x 'b) (== y 'a)]
    [(== x 'c) (== y 'd)]))

(defineo (win x)
  (fresh (y) (edge x y)
    (noto (win y))))
```

Under stable model semantics, such a program has two stable models considering predicate "win", $\{win(a), win(c)\}$ and $\{win(b), win(c)\}$. We can verify our solver can produce the right answer with some example queries.

Listing 26. Example queries and output in stableKanren

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

For the last output, we have explained the reason for getting multiple duplicated answers at the end of section 3.8.

4 CONCLUSIONS AND FUTURE WORK

This paper introduced stableKanren, a core miniKanren extension with normal program solving ability under stable model semantics. Inspired by the property of stable model semantics, our main idea was to get the minimal model and its corresponding reduct program at the end of the computation. Therefore, our work extended top-down unification and resolution algorithms, giving more semantics to them so that they can handle normal program solving. The extended unification works as a base case to produce truth values under the negation scenario. We retained the nice property of resolution to produce a minimal model by always attempting to prove a goal to be true. Then the extended resolution takes care of the loop scenario by producing truth values without unification. We designed the rule's body transformation enabling resolution to distribute negation through a compound goal to the base unification level. Moreover, the existential quantifier turned into the universal quantifier under negation. So we directed the resolution to compute the domain of the fresh variables and changed the execution course to traverse all values. Lastly, the resolution does not stop after obtaining a partial answer set. Rather, it extends its execution to check all rules and values in the program to ensure no contradiction exists.

By utilizing two functional programming constructs, macros and continuations, our work demonstrates how easily the above features can be implemented in a functional language Scheme. We laid a foundational framework for us moving into future implementations. One future work is going to have a detailed discussion and solution about variable safety. Currently, we assume the variables are safe under negation, the input program follows the format in definition 2.3. For example, given a logic program with four facts.

```
a(1).                          (defineo (a x)
a(3).                            (conde [(== x 1)]
                                        [(== x 3)]))

b(1).                          (defineo (b x)
b(2).                            (conde [(== x 1)]
                                        [(== x 2)]))
```

And the inference rule does not follow the format in definition 2.3. So the negative goal uses an unsafe variable.

```
p(X) :- not a(X), b(X).
```

```
(defineo (p x)
  (noto (a x))
  (b x))
```

A query on such a goal with an unsafe variable is unable to produce the expected answer. The user needs to bind the unsafe variable with a value to get the answer. So the query '(p x)' returns an empty list, and the query '(p 2)' returns true.

```
> (run* (x) (p x))
()
```

```
> (run* (x) (p 2))
(_.0)
```

If we adjust the order of the inference rule to match the format in definition 2.3. The variable is safe to use and the query produces the expected answer.

```
(defineo (p x)
  (b x)
  (noto (a x)))
```

```
> (run* (q) (p q))
(2)
```

Therefore, the variable safety can be resolved by compilation time rewriting through a macro so that the rule follows the desired format (definition 2.3). However, the user also can have a query on a negative goal using an unsafe query variable.

```
> (run* (q) (noto (p q)))
(1)
```

```
> (run* (q) (noto (p 3)))
(_.0)
```

As we can see, the declarative stable model semantics produce 'not p(1), not p(3)' as part of the answer set. But our first query can only output one result, we have to bind the unsafe variable with a value to get another result. Hence, having a runtime variable safety solution is also necessary. The user will have a fully declarative way of using our system once we resolve variable safety issues.

Additionally, we need to prove the soundness and completeness of our algorithms w.r.t stable model semantics. The proof can be derived from the property of stable model semantics (definition 3.1). More future work includes, but is not limited to, supporting constraint rules and choice rules, solving the infinite Herbrand model, incorporating inequality constraints, applying the bottom-up Conflict Driven No-good Learning (CDNL) algorithm [12] to top-down, etc. Even though in functional programming we use streams to simulate backtracking, we still have the potential of applying CDNL so that we can cut off other sibling streams if the current one generates a conflict. The challenges include, but are not limited to, tagging the streams with an ID similar to how Clingo tags decision levels and the communication between different streams. The above future work will improve our solver's expressiveness, performance, and robustness.

REFERENCES

[1] Dirk Abels, Julian Jordi, Max Ostrowski, Torsten Schaub, Ambra Toletti, and Philipp Wanko. 2021. Train Scheduling with Hybrid Answer Set Programming. *Theory and Practice of Logic Programming* 21, 3 (2021), 317–347. https://doi.org/10.1017/S1471068420000046

[2] Chitta Baral and Michael Gelfond. 1994. Logic programming and knowledge representation. *Journal of Logic Programming* 19-20, SUPPL. 1 (1 Jan. 1994), 73–148. https://doi.org/10.1016/0743-1066(94)90025-6

[3] William E. Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph. D. Dissertation. USA. Advisor(s) Friedman, Daniel P. AAI3380156.

[4] William E. Byrd, Gregory Rosenblatt, Michael J. Patton, Thi K. Tran-Nguyen, Marissa Zheng, Apoorv Jain, Michael Ballantyne, Katherine Zhang, Mei-Jan Chen, Jordan Whitlock, Mary E. Crumbley, Jillian Tinglin, Kaiwen He, Yizhou Zhang, Jeremy D. Zucker, Joseph A. Cottam, Nada Amin, John Osborne, Andrew Crouse, and Matthew Might. 2020. mediKanren: a System for Biomedical Reasoning. In *miniKanren Workshop.* http://minikanren.org/workshop/2020/minikanren-2020-paper7.pdf

[5] Keith L. Clark. 1978. Negation as Failure. *Logic and Data Bases* (1978), 293–322. https://doi.org/10.1007/978-1-4684-3384-5_11

[6] JÃŒrgen Dix. 1995. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundam. Inform.* 22 (03 1995), 257–288.

[7] Carmine Dodaro and Marco Maratea. 2017. Nurse Scheduling via Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning*, Marcello Balduccini and Tomi Janhunen (Eds.). Springer International Publishing, Cham, 301–307.

[8] D.P. Friedman, W.E. Byrd, and O. Kiselyov. 2005. *The Reasoned Schemer.* MIT Press. https://books.google.com/books?id=_xciAQAAIAAJ

[9] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (2nd ed.). The MIT Press.

[10] Daniel P. Friedman and Matthias Felleisen. 1996. *The Little Schemer (4th Ed.).* MIT Press, Cambridge, MA, USA.

[11] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. Conflict-driven answer set enumeration. In *Logic Programming and Nonmonotonic Reasoning: 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007. Proceedings 9*. Springer, 136–148.

[12] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. Conflict-Driven Answer Set Solving. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence* (Hyderabad, India) *(IJCAI'07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 386–392.

[13] Martin Gebser, Arne König, Torsten Schaub, Sven Thiele, and Philippe Veber. 2010. The BioASP Library: ASP Solutions for Systems Biology. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, Vol. 1. 383–389. https://doi.org/10.1109/ICTAI.2010.62

[14] Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. Cambridge, MA, 1070–1080.

[15] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. 2007. Coinductive Logic Programming and Its Applications. In *Proceedings of the 23rd International Conference on Logic Programming* (Porto, Portugal) *(ICLP'07)*. Springer-Verlag, Berlin, Heidelberg, 27–44. http://dl.acm.org/citation.cfm?id=1778180.1778186

[16] Intl. Organization for Standardization. 1995. *ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core.* 199 pages. https://www.iso.org/standard/21413.html

[17] Intl. Organization for Standardization. 2000. *ISO/IEC 13211-2:2000: Information technology — Programming languages — Prolog — Part 2: Modules.* https://www.iso.org/standard/20775.html

[18] Roland Kaminski, Torsten Schaub, and Philipp Wanko. 2017. *A Tutorial on Hybrid Answer Set Solving with clingo.* Springer International Publishing, Cham, 167–203. https://doi.org/10.1007/978-3-319-61033-7_6

[19] John W. Lloyd. 1987. *Foundations of Logic Programming, 2nd Edition.* Springer. https://doi.org/10.1007/978-3-642-83189-8

[20] Kyle Marple, Elmer Salazar, and Gopal Gupta. 2017. Computing Stable Models of Normal Logic Programs Without Grounding. *CoRR* abs/1709.00501 (2017). arXiv:1709.00501 http://arxiv.org/abs/1709.00501

[21] John McCarthy. 1960. *Programs with Common Sense.* Technical Report. Cambridge, MA, USA.

[22] Richard Min and Gopal Gupta. 2010. Coinductive Logic Programming with Negation. In *Logic-Based Program Synthesis and Transformation*, Danny De Schreye (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–112.

[23] Evgenii Moiseenko. 2019. Constructive negation for minikanren. In *ICFP 2019, The miniKanren and Relational Programming Workshop.*

[24] Raymond Reiter. 1981. On closed world data bases. In *Readings in artificial intelligence.* Elsevier, 119–140.

[25] J. A. Robinson and E. E. Sibert. 1982. *LogLisp: Motivation, Design and Implementation.* K.L. Clark and S.-A. Tarnlund (Eds.): Logic Programming, Academic Press, New York, 229–314.

[26] Miroslaw Truszczynski. 2012. Connecting first-order ASP and the logic FO (ID) through reducts. *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz* (2012), 543–559.

[27] M. H. Van Emden and K. L. Clark. 1984. *The Logic of Two-Person Games.* Prentice-Hall, Inc., USA, Chapter 12, 320–340.

[28] M. H. Van Emden and R. A. Kowalski. 1976. The Semantics of Predicate Logic as a Programming Language. *J. ACM* 23, 4 (Oct. 1976), 733–742. https://doi.org/10.1145/321978.321991

[29] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM* 38, 3 (July 1991), 619–649. https://doi.org/10.1145/116825.116838

[30] Philip Wadler. 1985. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–128.