Preface

This report aggregates the papers presented at the fifth miniKanren and Relational Programming Workshop, hosted on September 8, 2023 in Seattle, WA, USA and co-located with the twenty-eight International Conference on Functional Programming.

The miniKanren and Relational Programming Workshop is a workshop for the miniKanren family of relational (pure constraint logic programming) languages: miniKanren, microKanren, core.logic, OCanren, Guanxi, etc. The workshop solicits papers and talks on the design, implementation, and application of miniKanren-like languages. A major goal of the workshop is to bring together researchers, implementors, and users from the miniKanren community, and to share expertise and techniques for relational programming. Another goal for the workshop is to push the state of the art of relational programming—for example, by developing new techniques for writing interpreters, type inferencers, theorem provers, abstract interpreters, CAD tools, and other interesting programs as relations, which are capable of being "run backwards," performing synthesis, etc.

Five papers were submitted to the workshop, and each submission was reviewed by two to three members of the program committee. After deliberation, four submissions were accepted to the workshop.

In addition to the four full papers presented

- William E. Byrd gave a morning tutorial on miniKanren,
- the workshop closed with an open discussion on the future of miniKanren.

Thanks to all presenters, participants, and members of the program committee.

Nada Amin & William E. Byrd

Program Committee

Nada Amin, Harvard University, USA (Co-Chair)
Michael Arntzenius, University of Birmingham, UK
Oliver Bračevac, Galois, Inc., USA
William E. Byrd, University of Alabama at Birmingham, USA (Co-Chair)
Evan Donahue, University of Tokyo, Japan
Thomas Gilray, University of Alabama at Birmingham, USA
Ekaterina Komendantskaya, Heriot-Watt University and Southampton University, UK
Ekaterina Verbitskaia, JetBrains, Serbia

Contents

Goals as Constraints: Writing miniKanren Constraints in miniKanren

EVAN DONAHUE, University of Tokyo, Japan

We present an extension to the relational programming language miniKanren that allows arbitrary goals to run efficiently as constraints. With this change, it becomes possible to express a large number of commonly used constraints, which normally require modifications to the underlying implementation, in pure miniKanren. Moreover, it also becomes possible to express a number of new constraints that have proven difficult to realize within existing constraint authoring frameworks. We believe this approach represents a promising avenue for further extending the expressiveness of miniKanren's constraint handling capabilities.

1 INTRODUCTION

Most non-trivial miniKanren programs depend on the use of constraints beyond unification. However, in many current implementations, adding new constraints requires modifying the underlying constraint solver itself, which can be error prone and requires deep knowledge of the implementation. The situation is somewhat improved by past work on constraint authoring frameworks [1, 11], which separate constraint authoring from core language development. However, even with the use of such frameworks, adding new constraints still requires the time and expertise to ensure that the constraints themselves as well as the interactions between constraints operate efficiently and preserve program correctness.

In this paper, we propose using miniKanren itself as a language for constraint authoring. As we demonstrate, using only unification, conjunction, disjunction, `fresh`, and a simple form of negation, it is possible to express a large number of common and novel constraints—including `booleano`, `finite-domain`, `listo`, negations of these constraints, `absento`, and `absento`'s inverse `presento`—in only a few lines of miniKanren code and have them all interoperate seamlessly and efficiently. By using miniKanren as the constraint implementation language, we ensure that any miniKanren user can author useful constraints with minimal knowledge of the constraint system. Moreover, by making the barrier to constraint authoring so low, it becomes practical to create highly specific constraints specialized to a given application domain, sometimes even reusing code by transforming a goal that generates a stream of structures into a constraint that checks for instances of that same structure simplify by specifying that it should run as a constraint.

The miniKanren implementation described in this paper uses first-order representations of goals both as goals and as constraints [20]. There are no specialized representations for constraints—constraints are simply first-order goals stored in the constraint store.[1] Constraints that require specialized representations, such as those that reason over infinite quantities that cannot be expressed in finite unifications and disunifications, such as types[2], remain the domain of the framework-based approaches referenced above. Possible points of integration between such approaches will be discussed throughout this paper.

[1]To be precise, because constraints and value bindings are mutually exclusive, first-order constraint goals are stored directly in the substitution, avoiding the need for a separate constraint store and saving several calls to walk, at the cost of a more complicated unifier. This optimization is orthogonal, however, to the central idea of the paper, and so we will continue to refer to a separate constraint "store," although this detail may help better understand the architecture underlying the code examples discussed later.

[2]To be precise, type checking opaque host system structures such as Scheme symbols cannot feasibly be performed using recursive unification-based relations. This is not to say that a type checking relation could not be written as a constraint using an appropriate representation for types, and indeed such a possibility is a prominent part of planned future work on this system, as will be discussed in Section 6.

Given the duality of the representations of goals and constraints, the key idea of this paper is that the miniKanren search with constraint solving can be viewed as a conversation between two miniKanren interpreters that differ primarily in how they interpret disjunction. The first interpreter, which runs under what we will refer to as "goal semantics," is primarily responsible for forking the disjunctive **conde** form into however many interleaving branches are needed to explore the space defined by the program. Whenever this interpreter applies a primitive constraint, including ==, =/=, and others, it passes control to the second interpreter running under "constraint semantics." The constraint interpreter implements a depth-first miniKanren search over the constraint goals contained in one particular branch of the overall search to determine whether or not the collection of constraints is unsatisfiable. If it is, the branch fails. Otherwise, the interpreter replaces the set of checked constraints with a simplified set in the constraint store and passes control back to the goal interpreter to create more branches. Unlike the goal interpreter, the constraint interpreter avoids speculatively binding fresh variables, and so is guaranteed to halt and pass control back to the goal interpreter, preserving the completeness of the overall search.

Much as the miniKanren search overall can be viewed as the search for one or more substitutions that satisfy the program constraints, the constraint solver can be viewed as conducting a search for a satisfiable collection of constraints consistent with the current substitution and logically equivalent to the original store, but of equal or lesser complexity. Failure to find such a satisfiable store proves its non-existence and implies that the branch should fail. Success allows the search to proceed with a set of constraints of equal or lesser computational complexity to use as inputs to further solving.

The remainder of the paper is structured as follows: Section 2 describes the interface extensions made to the language to allow the specification of constraints and presents a list of example implementations of several constraints. Section 3 describes the implementation in detail. Section 4 discussion ongoing research and open questions related to the present work. Section 5 discusses related work.

2 INTERFACE

In this section, we implement a collection of constraints as a means to illustrate the functioning of the proposed constraint language. We extend core miniKanren with three new forms: `constraint` (2.1), which converts miniKanren goals into constraints, **`pconstraint`** (2.2), which defines new primitive constraints besides ==, and **`noto`** (2.3), which negates miniKanren goals and constraints subject to limitations described at length in Section 4.2.

2.1 `constraint`: miniKanren Goals as Constraints

The `constraint` form wraps arbitrary miniKanren goals and redefines their semantics. The semantics of `constraint` are defined in terms of conjunction, disjunction, **`fresh`**, and the class of all primitive constraints.

Normally, a miniKanren goal comprised of ==, conjunction, **`fresh`**, and especially disjunction evaluates to a stream that, after an unbounded amount of time, may yield 0 or more answers. If a goal is composed entirely of conjunctions of primitive constraints along with **`fresh`**, the semantics will be precisely the same for a goal wrapped with `constraint` as for an unconstrained goal. The key difference between constrained and unconstrained goals emerges only in the presence of disjunction.

Under the unconstrained "goal semantics," a disjunction, represented by **`conde`**, will generate a search tree in which each disjunct is solved separately in its own branch. Under constraint semantics, however, the disjunction will evaluate some number of its disjuncts and then suspend itself along with its simplified disjuncts in the constraint store of a single branch. As further constraints are

added, this disjunction will be rechecked as needed until all branches but one fail, and the primitive constraints (or indeed further compound constraints) contained in that disjunct will be solved as normal. This one change makes it possible to convert generative relations that would spawn multiple branches into constraints that operate on a single branch, as in the following examples:

2.1.1 `booleano`. The simplest non-trivial constraint we can write using `constraint` is the `booleano` constraint from Friedman and Hemann [11]. `booleano` constrains a variable to be either #t or #f. Using `constraint`, `booleano` could be written as follows:

```
(define (booleano v)
  (constrain
   (conde
     [(== v #t)]
     [(== v #f)])))
```

Assuming `v` is free, this constraint will suspend itself in the constraint store and await unification. When `v` is unified, the constraint activates and check that `v` is either #t or #f. If it is one of those two values, the constraint is satisfied and it is removed from the store. If it is bound to a different ground term, the constraint fails. Otherwise, if it is bound to a variable, the constraint returns to the constraint store attributed to the new variable. Performant miniKanren implementations often use a constraint store in which constraints are indexed by the variables on which they depend, allowing for faster constraint solving than an implementation that rechecks all constraints every time the store or substitution are modified. This implementation likewise provides a strategy for efficient constraint lookup for the generalized constraints introduced in this paper, which we discuss in Section 3.1.[3]

Likewise, if `v` ever becomes disequal to either #t or #f, the disjunction will collapse and the constraint will unify the remaining value in the substitution before removing itself from the store.

2.1.2 `finite-domain`. `booleano` generalizes naturally to an arbitrary finite domain constraint of a type alluded to in Alvis et al. [1]. `finite-domain` can be written as follows:

```
(define (finite-domain v domain)
  (constrain (fold-left disj fail (map (lambda (d) (== v d)) domain))))
```

`finite-domain` here takes a variable and a list of arbitrary domain elements. It then dynamically constructs a disjunction of unifications between the variable and each element of the domain using the explicit disjunction constructor **`disj`**.

As with `booleano`, this constraint simply checks that any value unified with `v` also unifies with some element of the domain. Note that `==` is fully general here, and the domain is free to contain fresh variables or arbitrary terms containing fresh variables as well as ground terms.

Although the finite domain constraint, fd, defined in [1] is an explicitly numeric constraint and was built to function with several other explicitly numeric constriants, the authors also discuss the possibility of other finite domain constraints over arbitrary domains, of which this is one example. It would be possible to use the present framework to implement some of the constraints described in that paper, such as `all-diff`, which could be implemented as a conjunction of disequalities. Moreover, mathematical operations such as $\leq$ could be implemented using **`pconstraint`**, described in Section 2.2. However, allowing greater interoperability between constraint solvers with specialized representations and the current framework are a topic for future investigation.

[3]We will refer to this scheme as employing "attributed variables," in the sense that constraints are attributed to variables on which they depend, although the usage here to describe a purely functional lookup scheme differs from some past usages of the term, as discussed further in Section 5.

2.1.3 `listo`. `listo`, drawn again from Friedman and Hemann [11], checks that a term unifies with a proper list. This constraint lazily walks the list and confirms that it ends—if its tail is ever fully bound—with a null list.

```
(define (listo l)
  (constrain
   (conde
     [(== l '())]
     [(fresh (h t)
        (== l (cons h t))
        (listo t))])))
```

`listo` in particular among the constraints introduced so far illustrates the duality of goals and constraints in this framework. Without the `constraint` form, `listo` would simply be a normal miniKanren goal that generates proper lists. It would be perfectly possible to define `listo` as a generative miniKanren goal and then wrap it using `constraint` only at the call site to turn it into a constraint at the programmer's discretion. Any miniKanren program that generates any arbitrary structure can likewise be turned into a constraint that tests for that structure using the `constraint` form. This allows not only easy implementation of familiar constraints, but similarly easy implementation of arbitrary constriants that may be idiosyncratic to the domain of a particular application.[4]

2.1.4 `presento`. The final constraint in this section, `presento`, is to our knowledge novel in this paper. `presento` can be understood to be the logical negation of `absento`. Instead of asserting that a given value must not appear anywhere in a term, `presento` asserts that a given value must appear somewhere in the term. `presento` is much more difficult to implement than `absento` using existing constraint frameworks due to the way in which it negates `absento`'s implicitly conjunctive relation into a disjunctive one. Because the constraint store functions as an implicit conjunction of all its constraints, and because `absento` conceptually defines a conjunction of disequality checks for every subterm in a tree, `absento` can simply distribute child `absento` constraints throughout the store with no additional bookkeeping. `presento`, by contrast, being fundamentally disjunctive, must keep within the constraint representation the entire term remaining to be checked, since any single unification that succeeds must remove all other unifications from the store. However, in the present framework `absento` and `presento` reduce to roughly the same order of implementation complexity:

```
(define (presento term present)
  (constrain
   (conde
     [(== term present)]
     [(fresh (h t)
        (== term (cons h t))
        (conde
          [(presento h present)]
          [(presento t present)]))])))
```

`presento` simply checks whether term is already equal to the present value, and if not, destructures it and recurses on its car and cdr. As usual, unification is fully general, and present can be an arbitrary miniKanren term containing free variables.

[4] One minor limitation is that, unlike the generative version of the relation, the constraint version never grounds the end of the list with null if it is not bound elsewhere in the program, instead reifying to a suspended form of the waiting constraint. It would be interesting to explore extracting disjunction constraints containing unifications and re-running them as goals as a final step before reification to fully ground returned answer terms, but we leave this investigation to future work.

2.2 `pconstraint`: Primitive Constraint Constructor

In the previous section, only == was used as a primitive goal. While == allows for a wide range of constraints on structures miniKanren is natively capable of generating, it is insufficient to define the full range of constraints usually present in miniKanren implementations. In particular, defining type constraints such as `symbolo` or `numbero` would require a disjunction of unbounded size, which cannot efficiently be represented within a miniKanren program. To support such constraints, this implementation defines the **`pconstraint`** form that acts as a constructor for new primitive constraints.

`pconstraint` accepts a list of variables on which the constraint depends, a function responsible for checking the constraint, and an arbitrary Scheme value to be passed as auxiliary data into the constraint checking function. Whenever one of the constrained variables is updated, the function receives the variable, its updated value, any constraints on the variable, and the auxiliary value. The function must return either a simplified pconstraint, or a trivial succeed or fail constraint. **`pconstraint`** was designed specifically to implement type constraints, and it may be necessary to further extend the system to handle other primitive constraints. We leave such considerations to future work.

2.2.1 `symbolo` *&* `numbero`. In this section we define a general `typeo` relation and specialize it to arrive at versions of the usual `symbolo` and `numbero` constraints common to many miniKanren systems.

```
1  (define (typeo v t?)
2    (if (var? v) (pconstraint (list v) type t?) (if (t? v) succeed fail)))
3
4  (define (type var val reducer reducee t?)
5    (cond
6     [(succeed? reducee) (typeo val t?)]
7     [(pconstraint? reducee) (if (eq? type (pconstraint-procedure reducee))
8                                 (values fail fail)
9                                 (values reducer reducee))]
10    ...))
11
12 (define (symbolo v) (typeo v symbol?))
13
14 (define (numbero v) (typeo v number?))
```

`typeo` accepts a value or variable and a function responsible for type checking, such as `symbol?`. If it receives a value, it simply returns the trivial fail or succeed goal. If instead it receives a variable, it constructs a pconstraint, represented as a tagged vector of its three arguments: the singleton list of the variable `v`, the auxiliary data which in this case is the type checking function `symbol?`, and a function responsible for performing the type check, `type`.[5]

The type checking function, `type`, at present requires some knowledge of the internal representations used by the solver. In practice, simpler interfaces can likely be defined to handle common constraint types. The function is called each time a variable on which the constraint depends is bound, and it accepts as arguments the variable, the value (or variable) to which it has been bound, the auxiliary data (in this case, the type predicate `t?`), and a pair of constraint goals used to check constraint-constraint interactions. The `reducer` constraint is the type constraint itself, and is only supplied as an optimization to avoid needing to construct new copies of itself. The `reducee` constraint is another primitive constraint bound to var.

[5] It would be possible to combine the type checking function and the auxiliary data into a single closure, but composing the constraint out of values comparable by `equal?` permits additional optimizations inside the solver.

The first case, when `reducee` is the trivial **`succeed`** constraint, is called to check only new bindings in the substitution. It expects one argument, which is simply the constraint on the new value or variable.

The second case deals with interactions between pconstraints, such as deciding the unsatisfiability of `symbolo` and `numbero` when applied to the same variable. If the other pconstraint's type checking procedure is also type, then it is a type constraint. Because pconstraints that are `equal?` are assumed not to conflict by the solver, a pconstraint passed to the constraint checking function must be unequal and, in the case of type constraints, thereby implies failure. Pconstraint interactions return two values, corresponding to the simplified form of the current constraint, the simplified form of the other constraint. Following a convention that will be used elsewhere in this paper, the ellipsis corresponds to code omitted due to its irrelevance to the current discussion.

2.3 `noto`: Negating Goals and Constraints

Negation has been explored from a variety of angles in past work on miniKanren [14, 19]. The notion of negation required for the present discussion is comparatively less sophisticated than many of those previous treatments. We extend miniKanren with a **`noto`** form for negating goals and constraints. Because we are working with a first-order miniKanren, it is straightforward to negate goals before they are evaluated. Conjunctions and disjunctions negate to disjunctions and conjunctions of negated subgoals, respectively, following the usual logical semantics of De Morgan's laws.[6] Primitive constraints simply become wrapped in reified **`noto`** structures which are handled by the interpreter. Existing **`noto`** structures simply return their contents, negating their negation. **`fresh`**, at present, can only be negated in a limited fashion. The details and reasons for this will be discussed at greater length in Section 4.2. Equipped with **`noto`**, it becomes possible to define a range of additional constraints by negating existing goals and constraints.

2.3.1 `=/=`. `=/=` is implemented in terms of `==` and **`noto`** as in the following example:

```
(define (=/= lhs rhs) (noto (== lhs rhs)))
```

Because the constraint system itself was conceived as a generalization of the logic of disequality solving, as discussed in Section 3.1, it is natural that disequality should be easy to express.

2.3.2 `not-booleano`. Due to the definition of negation on conjunctions, disjunctions, and `==`, any constraints built with these operators become negatable.

```
(define (not-booleano v) (noto (booleano v)))
```

The negation simply converts the disjunction of `==` to a conjunction of `=/=`.

2.3.3 `not-symbolo`, `not-numbero`. Because negation operates in a general way to negate primitive constraints, any constraint built with **`pconstraint`** can be negated in the same way as `==`.

```
(define (not-symbolo v) (noto (symbolo v)))

(define (not-numbero v) (noto (numbero v)))
```

[6] Because constraints in this framework are composed of goals, **`noto`** can be used to negate ordinary goals as well as constraints, which may be useful independently of its use in defining constraints. For instance, using **`noto`**, it becomes possible to define certain limited forms of branching as demonstrated by Moiseenko [19].

2.3.4 `absento`. Using disequalities and type constraints, it becomes possible to define the familiar `absento` constraint.

```
(define (absento absent term)
  (constrain
   (=/= term absent)
   (conde
     [(noto (typeo term pair?))]
     [(fresh (h t)
        (== term (cons h t))
        (absento absent h)
        (absento absent t))])))
```

It is notable that despite being the logical negation of `presento`, `absento` is more complicated to write, requiring a type constraint not expressible using the core miniKanren operators. This discrepancy points to a nuance in the semantics of **`fresh`** that will be discussed in Section 4.2. As with all previous constraints, `==` is fully general and `absento` accepts arbitrary terms including free variables.

3 IMPLEMENTATION

The constraint solving interpreter takes over when the goal interpreter encounters a constraint, whether `==`, `=/=`, a **`pconstraint`**, or a complex goal wrapped in `constraint`. In fact, the goal interpreter defines only conjunction, disjunction, and **`fresh`**, as these all touch on the semantics of streams, and everything else it delegates to the constraint interpreter. The semantics of the constraint interpreter are similar to those of normal miniKanren, with the exception that rather than building a stream, it operates directly on the first-order goal structures.

What it means to "solve" a constraint depends heavily on the objective of the solving, as discussed further in Section 4.1. The present implementation conforms to the three objectives that unsatisfiable constraint stores should always fail, constraint stores that entail a unification should always add the corresponding binding to the substitution, and solving effort should be conserved to the greatest extent possible if not required to satisfy the first two objectives. Informally, this means that the solver, in order to solve a constraint, must walk some of its variables in the substitution until it can be determined whether the constraint implies failure, unification, or that there is insufficient information in the store to make a determination regardless of the values of the other variables on which the constraint depends. In the course of this solving procedure, it may be necessary to fetch other constraints from the store that share variables with the constraint in question in order to check for incompatibilities. When the solver is satisfied, it returns all of the walked and simplified constraints to the store and the search continues.

In the next several sections, we review the implementation of efficient disequality constraints as found in performant implementations such as faster-mk, and then generalize the disequality solving procedure to arbitrary negation and disjunction [2].

3.1 Generalized Disequality

As a preface to the main description of the constraint solver, it is worth revisiting the standard method that performant miniKanren implementations use to implement disequality constraints. Because the solving of disequality constraints contains, implicitly, the concepts of conjunction, disjunction, equality, negation, and variable attribution, it can be viewed as a special case of the overall constraint solving strategy presented in this paper, and as such will offer a useful mental model for reasoning about the details of implementation.

The traditional way to implement =/= in miniKanren involves a particularly elegant case analysis of the results of normal unification. To review, when evaluating a disequality constraint, the disequal terms are unified in the current substitution. If the unification fails, the terms can never be made equal, and so the constraint is redundant and can be removed from the store. If the unification succeeds without modifying the substitution, the terms are already equal and the constraint is unsatisfiable, and so it fails. Finally, if the unification succeeds, the bindings it has added to the substitution represent the remaining unifications that must be made to render the constraint unsatisfiable, and they can be interpreted directly as a new, simplified disequality constraint to be returned to the store. This case analysis of unification is due to Bürckert [5], and is the standard means of implementing disequality in miniKanren. The central idea of this paper is that precisely the same strategy generalizes to any miniKanren goal, and can therefore transform that goal into a constraint.

An example will serve both to recall the specifics of disequality solving as well as illustrate the more general principles that underlie the present approach to more complex constraints. Consider the following pair of disequality constraints:

$$z \neq 1 \wedge (x, z) \neq (y, w) \tag{1}$$

Assume a substitution of $\{z \mapsto x\}$. First, z and 1 are unified, yielding the substitution extension $x = 1$, which the disequality constraint interprets as $x \neq 1$ and adds to the constraint store. Next, (x, z) and (y, w) are unified, resulting in the logical extension to the substitution equivalent to $x = y \wedge x = w$, which the disequality constraint negates to yield $x \neq y \vee x \neq w$. This constraint must then be added to the store attributed to both x and y. This is because either might be unified with the other, causing the constraint to become unsatsifiable, and so this constraint should be checked no matter which is bound.[7] Finally, the store consists of two attributed variables, with y pointing to its single constraint, and x pointing to a pair of constraints.

Viewed more abstractly, this admittedly somewhat motley collection of terms contains all of the necessary properties to serve as an exemplar most of the more general constraint solving and attribution scheme developed in this paper. To begin with, not just == but any primitive constraint can be analyzed in terms of the same case analysis.

New constraints introduced with **`pconstraint`**, for instance, may also trivially succeed or fail, in which case they can be discarded or the search can be terminated, respectively. They can also return a simpler solved form that must be added back to the constraint store, attributed to the appropriate variables. When negated, the case analysis is similarly inverted. Trivial successes become failures and vice versa. The solved form is simply wrapped in a negated container and treated as an atomic constraint on the same variables. When the negated container is itself rechecked for further solving, it again simply solves its child, and performs the same inverted case analysis.

Variable attribution likewise generalizes from the case of disequality. A primitive constraint like $x \neq y$ must be attributed to both of its variables because either could cause the constraint to fail. However, a disjunction of multiple disequalities, such as $x \neq y \vee x \neq w$, may ignore all but the first disjunct as so long as a single disjunct remains satisfiable, the constraint as a whole remains satisfiable. In this case, the constraint must be attributed to x and y but not w.

Given the above description of the normal functioning of disequalities, it is possible to view the overall handling of constraints as a generalization of each part of the disequality solver, as described in the following section.

[7] This is somewhat implementation dependent. Some unification schemes, such as those described in Bender et al. [3], cause pairs of unified free variables to be well ordered, and therefore permit the attribution of a disequality constraint to only one variable.

3.2 Conjunction

The primary interface to the constraint solving interpreter is via the `solve-constraint` function. Consider the following partial listing:

```
1 (define (solve-constraint g s ctn resolve committed pending)
2   (cond
3     [(succeed? g) (if ... (solve-constraint ctn s succeed resolve committed
       pending))]
4     [(conj? g) (solve-constraint (conj-lhs g) s (conj (conj-rhs g) ctn) resolve
       committed pending)]
5     ...)))
```

The interpreter accepts the constraint goal to be solved, `g`, the state, `s`, and four additional goals, `ctn`, `resolve`, `committed`, and `pending`. These naming conventions will remain consistent throughout the rest of the paper.

`g` and `s` are self-explanatory. `ctn` is so named because the interpreter is written in a depth-first manner using a "conjunction-passing style" in which the future of the computation, `ctn`, represented as the conjunction of all goals to the "right" of the currently evaluated goal, is passed as an argument to the solver. When the interpreter receives a conjunction, it calls itself recursively on the left-hand side while conjoining the right hand side to the current `ctn`. When the solver later finishes solving the current constraint `g`, it will be called with the trivial **`succeed`** goal as the current constraint, which will prompt the interpreter—subject to conditions discussed in more detail in the following sections in connection with the remaining three arguments—to proceed with solving the next conjunct of the current `ctn`. Concretely, calling the solver with $g \mapsto x \neq 1 \wedge y \neq 2$ and $ctn \mapsto z \neq 3$ will first trigger the conjunction condition, calling the solver recursively with $g \mapsto x \neq 1$ and $ctn \mapsto y \neq 2 \wedge z \neq 3$, and then subsequently with $g \mapsto succeed$ and then $g \mapsto y \neq 2$ and $ctn \mapsto z \neq 3$, provided that none of the constraints fail.

3.3 Unification

Consider the following partial listing of the unification solver, which is called from `solve-constraint` when `g` is a unification constraint:

```
1 (define (solve-== g s ctn resolve committed pending)
2   (let-values ([(bindings recheck s) (unify s (==-lhs g) (==-rhs g))])
3     (if (fail? bindings) (values fail fail failure)
4         ...
5         (solve-constraint succeed s ctn (conj recheck resolve) (conj committed
      bindings) pending))))
```

This definition of unification will look familiar from its standard implementation elsewhere. The unifier is called, the resulting state is checked for failure, and if it has not failed, the solver proceeds to run any constraints that need to be rechecked based on the new bindings. Line 2 calls out to a unifier that works like most miniKanren unifiers with the exception that it returns two goals in addition to the state. `bindings` is a conjunction of unification goals representing the extensions made to the state `s` (analogous to the newly extended prefix of the substitution in association list based implementations, but represented using explicit first-order goals rather than a list of bindings). `recheck` represents the conjunction of constraints on all of the newly bound variables. The next two lines illustrate the remainder of the plumbing of the solver.

Line 3 checks whether the unification has failed by checking whether the bindings consist of the trivial **`fail`** goal, and returns the failure signature—three values consisting of two instances of the

trivial **fail** goal and one failure stream in place of the state. The two returned goals constitute a first-order representation of the changes made to s. The first corresponds to the committed constraint, and the second to pending. Consider line 5. The unification constraints representing the new bindings are conjoined to committed and passed to further solving. All three—committed, pending, and s—comprise the return signature of the overall solve-constraint function.

If the program as a whole consists entirely of unifications, then the return value will be the conjunction of all of those unifications alongside the trivial **succeed** goal, which is the default value for pending, and the state. The committed constraint represents, as a first-order goal, all changes that have been committed directly to the state, such as extending its substitution. Re-solving the committed constraint on top of the original state should yield a logically equivalent state. As such, the top level solver simply discards this constraint and returns the state as is. The reason this constraint exists is that, in the case of disjunction and negation, the state represents a kind of working memory for the solver that will not ultimately reflect the state of the overall search. In such cases, it is necessary to discard the state, leaving only committed as a record of the changes made to it. committed will then be folded into, for instance, a disjunction constraint and conjoined to pending instead in the returned values. Pending constraints are not reflected in the current state, and are added to the store at the end of the solving process. This distinction will be further elaborated upon in the following sections on negation and disjunction, respectively.

The final architectural element of the solver is the resolve constraint, which is conceptually equivalent to the ctn constraint. Both are conjunctions of goals waiting to be solved. The difference is that ctn contains the constraints remaining to be solved from the initial constraint received from the goal solver, whereas resolve contains constraints that started out already in the store, and were removed by, for instance, a unification, and must be re-solved later. As such, the constraints relevant to the current unification, recheck, are conjoined with resolve before further solving, and will later be pulled out and solved once ctn has been exhausted. Intuitively, while constraints received from the goal interpreter and stored in ctn are necessarily not yet reflected in the state, constraints conjoined to resolve were initially in the state when the constraint interpreter began solving the current constraint. As such, committed must contain a record of the changes made to the state, which corresponds to the logical simplification of the ctn constraint, whereas it need not contain re-solved constraints already contained in the state, and so resolve may be discarded from the final output, although it must be checked to ensure consistency. This distinction is important for the correctness of the negation constraint, as discussion in Section 3.4.

3.4 Negation

Generalized negation operates analogously to the specialized case of disequalities. The same case analysis by which disequality constraints interpret the results of unification can be applied to general constraints such as type constraints and others defined with **pconstraint**. Negated constraints simply solve their child constraints and invert the result, converting **succeed** to **fail**, **fail** to **succeed**, and non-trivial constraints to their negations. Consider the following listing:

```
1 (define (solve-noto g s ctn resolve committed pending)
2   ...
3   (let-values ([(c p s2) (solve-constraint g s succeed succeed succeed succeed)])
4     (if (succeed? p)
5         (solve-constraint succeed (store-constraint s (noto c)) ctn resolve (conj
      committed (noto c)) pending)
6         (solve-constraint succeed s (conj (disj (noto c) (noto p)) ctn) resolve
      committed pending))))
```

`g` in this case is the positive form of the goal contained within the negated structure. If the negated goal is $x \neq 1$, represented internally as the tagged vector (**`noto`** `(== x 1)`), then `g` is `(== x 1)`. Note that the initial call to `solve-constraint` on line 3 is called with `ctn`, `resolve`, `committed`, and `pending` all set to **`succeed`**. This creates a distinct, "hypothetical" context in which the solver can evaluate the positive version of the goal in isolation and without reference to future conjuncts of the original negated goal. This results in constraint representing the changes committed to the state, `c`, the pending constraints not contained in the returned state, `p`, and the resulting state `s2`. Because the solver was operating in a hypothetical mode, the state itself is not consistent with the negation of `g` and `s2` is simply discarded, leaving only `c` and `p` as the results of the solving process. Continuing the above example and assuming a substitution of $x \mapsto y$, `c` would be $y = 1$, as returned by the unification constraint solver, and `p` would be **`succeed`**.

If the true continuation and return value constraints had been used in the solving step, then `c` and `p` would represent conjunctions not only of `g`, but also of past (`committed` and `pending`) and future (`ctn` and `resolve`) solving. When `c` and `p` are later negated to arrive at the solved form of the negated constraint, this would also implicitly negate other constraints conjoined with the negation but not negated themselves, yielding incorrect results. For instance, when solving the constraint $x \neq 1$ (and so $g \mapsto x = 1$), if `ctn` was $z \neq 2$, and was passed into the solver at this point, the resulting `c` would be $y = 1 \wedge z \neq 2$, which, when negated, yields $y \neq 1 \vee z = 2$, which was not the original constraint.

Once the `c` and `p` deltas have been returned, the constraint must be added to the store. `c` and `p`, again, represent two conjuncts that collectively constitute the simplification of `g`. Their negation, consequently, has the form $\neg c \vee \neg p$. In the current implementation, pending constraints—`p` in this case—may contain disjunctions containing in turn some disjuncts that have lazily not been normalized. When negation turns these into conjunctions, those variables require additional solving. As a result, only when p is **`succeed`**, and $\neg p$ is therefore **`fail`** can we add the negated constraint directly to the store and continue solving. If any pending constraints are received, the negated constraint must be further solved by the disjunction solver before it can be added to the store. This explains the conditional in the final step of the routine, and will be further elaborated upon in the next section on disjunction.

Before proceeding with the remaining constraints, two remarks are in order. First, it is now possible to return, briefly, to the `solve-constraint` function and its handling of **`succeed`**:

```
1  (define (solve-constraint g s ctn resolve committed pending)
2    (cond
3     [(succeed? g)
4      (if (succeed? ctn)
5          (if (succeed? resolve)
6              (values committed pending s)
7              (let-values ([(c p s) (solve-constraint resolve s succeed succeed
       committed pending)])
8                (if (failure? s) (values fail fail failure)
9                    (values committed pending (store-constraint s p)))))
10         (solve-constraint ctn s succeed resolve committed pending))]
11     ...)))
```

When the `g` is **`succeed`**, constraints are first pulled from `ctn` on line 10, as described earlier. Once `ctn` has been exhausted, the constraints removed from the state to be rechecked as a result of the solving process, contained in `resolve` are solved on line 7, but they are added only to the state, and not to the deltas represented by `committed` and `pending`, which are returned unchanged on line 9. For this reason, the normalized constraints returned to the negation solver represent only the negated

constraint and can be negated safely. Finally, once all future constraints have been exhausted, the delta values are returned along with the state on line 6.

Second, we note that while disequality, implemented as a negation of unification, fits naturally into the above framework, the current implementation handles disequality separately for optimization purposes. Unification as traditionally implemented in miniKanren can be viewed as lying the extreme end of a spectrum from lazy to eager solvers for equality and disequality constraints. Unifying the list `(x1 x2 . . . x99)` with `(1 2 . . . 99)` makes 99 unifications in the same substitution. This is inevitable when solving an implicit conjunction as in the case of unifying lists. However, as disunifying lists yields a disjunction of disequalities rather than a conjunction of equalities, it is only necessary to ascertain that one primitive disequality holds to prove that the constraint as a whole cannot fail. The current implementation uses a special purpose disunifier that halts execution at the first still satisfiable disequality and returns the remainder of the disunification as a simple disunification between unwalked lists, or in this case, assuming `x1` is unbound, $x1 \neq 1 \vee (x2 \ldots x99) \neq (2 \ldots 99)$. This results in significant speed ups for large disequalities, at the potential cost of needing to re-solve the tail of the disequality multiple times in multiple future branches. In future research, we intend to explore the costs and benefits of such trade-offs in practice.

3.5 Disjunction

Disjunction is the most conceptually complex constraint type due to the fact that it admits a variety of possible solvers with different properties. Before describing the implementation, we first introduce a non-exhaustive list of possible "levels" of constraint solving that differ primarily in how disjunctions are handled. The current system implements level 2, but as we discuss in Section 4.1, a further exploration of when these levels may be applicable in practice is warranted, as it is not at all clear if and when level 2 is to be preferred.

3.5.1 Solving Levels. A level 1 solver fails when the constraints in the store entail a failure. In practice, it seems reasonable to expect that most applications will require at least this level of constraint solving, although a weaker solver is theoretically imaginable. In order to achieve this level of solver, only a single non-failing disjunct of any disjunction needs to be solved. So long as one disjunct has been proven not to fail, the disjunction as a whole cannot fail, and so the values of the remaining disjuncts are irrelevant. This level is relatively simple to implement, however it fails to give desirable behavior when evaluating constraints such as `booleano`, as $(x = \top \vee x = \bot) \wedge x \neq \bot$ implies $x = \top$, which may be an important fact for a user of the solver to know, but a level 1 solver that only evaluates the first disjunct will be unable to prove it.

A level 2 solver handles cases such as `booleano` by adding the requirement that any unifications entailed by the constraint store should be added as a binding in the substitution, so that the full list of entailed bindings can be read directly from the substitution. Specifying the appropriate behavior of the disjunction solver to achieve this requirement without performing additional work is quite complex. As a first step, we can require the solver to solve at least two disjuncts. This will handle `booleano`, as the second disjunct will be checked and found to be a failing branch, and the only remaining constraint will be $x = \top$, which can be added directly to the substitution. So long as two disjuncts do not fail, then it can be guaranteed that the disjunction as a whole will not collapse to a single disjunct, and so this gives some assurance that unifications within the disjunction cannot be committed to the store, however, there is still the case in which a unification common to all disjuncts can be factored out.

Consider the disjunction $(x = 1 \wedge y = 1) \vee (x = 1 \wedge y = 2) \vee (x = 1 \wedge y = 2)$. This disjunction clearly entails the fact that $x = 1$, yet there is no bound on the number of disjuncts that may need

to be solved in order to prove this. Instead, the implementation must continue to solve disjuncts so long as there are unification constraints common to each solved disjunct and then extract the common constraints from the representations of the disjuncts. This rule handles the case of `booleano`, as if the initial disjunct contains a unification, the solver must check at least the second to confirm there are no common unifications. Moreover, if the first disjunct contains no unifications, then the solver can stop immediately, as no other disjunct can share unifications with it and the disjunction as a whole cannot fail. Note that because the solver maintains a normalized record of the deltas added to the state when solving each constraint, unifications made to the state should appear in their simplified form in these deltas and be accessible through simple inspection of the returned constraints. This somewhat complex rule yields the implementation presented below, although first we conclude the thought with a brief description of hypothetical levels 3 and 4.

A level 3 solver solves all disjuncts and moreover requires all constraints common to all disjuncts to be factored out. We combine these requirements into one level not for any necessary reason, but because we believe a level 3 solver so defined represents a potentially interesting subject for further research, as described in Section 4.1. Finally, a level 4 solver adds the requirement that disjuncts not be redundant. $x \neq 1 \vee x \neq 2 \vee x \neq 1$, for instance, must reduce to $x \neq 1 \vee x \neq 2$. Disequality solvers that re-use unification in the usual way are level 4 constraint solvers over the limited subclass of disjunctions of simple disequalities.

3.5.2 Implementation of Disjunction. Consider the following listing:

```
1  (define (solve-disjunction g s ctn resolve committed pending)
2    (let-values ([(c-lhs p-lhs s-lhs) (solve-constraint (disj-lhs g) s ctn resolve
       succeed succeed)])
3      (let* ([lhs (conj c-lhs p-lhs)])
4        (if (fail? lhs)
5            (solve-constraint (disj-rhs g) s ctn resolve committed pending)
6            (let*-values
7                ([(c-rhs p-rhs s-rhs)
8                  (if (conj-memp lhs ==?)
9                      (solve-constraint (disj-rhs g) s ctn resolve succeed succeed)
10                     (values succeed (conj (disj-rhs g) ctn) s))]
11                     [(rhs) (conj c-rhs p-rhs)])
12             (if (fail? rhs)
13                 (values (conj committed c-lhs) (conj pending p-lhs) s-lhs)
14                 (values committed
15                         (conj pending (disj-factorized lhs rhs)) s)))))))
```

`solve-disjunction` first solves the left-hand disjunct of the binary disjunction structure that constitutes the constraint. Like the negation solver, `committed` and `pending` are both **succeed**, which ensures that the returned constraints reflect only simplifications of constraints contained within the disjunct as opposed to constraints added previously to the state. Unlike negation, however, here it is permissible and indeed required to pass in the true continuation conjunctions `ctn` and `resolve`. The continuation conjunctions can be viewed as distributed over the disjunction and conjoined individually to each disjunct. The solver can then inspect each disjunct and extract common unifications, whether supplied by the disjuncts themselves or by the continuation, and add them to the substitution.

If the left-hand disjunct fails, the solver simply solves the right-hand disjunct on line 5. Otherwise, if the left-hand side contains any unifications within the top level conjunction (as checked by `conj-memp` using the fact that simplifying code elsewhere has extracted common unifications and

conjoined them with the overall disjunction) the right-hand constraint is solved, checked for failure, and returned disjoined with the left-hand side.

It is useful to check for the failure of the right-hand side explicitly on line 13 because this allows us to return the state, `s-lhs`, which already reflects the changes made by the left-hand side. Because each disjunct has made different hypothetical changes to the state, returning the complete disjunction of both sides requires that we throw away each state and return only the original state. Solved disjunctions are not replaced in the store until the very end of the solving process.

Stepping back, the constraint solving process can be viewed as that of ensuring that all conjoined constraints are compatible and that at least one combination of disjuncts in all conjoined disjunctions in the store are compatible. We can view this abstractly as removing all disjunctions, and for each disjunct in the first, iterating through all disjuncts in the second and so on until we iterate through all possible combinations. Once we reach a combination that does not fail, we suspend the search early using the technique on line 10. If it is determined that the right-hand side does not need to be solved, signifying an early detection of a satisfiable collection of disjuncts, instead of a solved right-hand side, we disjoin the unsolved right-hand side conjoined with the continuation.[8] This is exactly analogous to the bind operation in the standard interleaving search. The right hand side represents an incomplete stream, and the continuation conjunctions are conjoined to that stream to be applied to each element of it later in the computation.

Finally, the new disjunction is conjoined to the pending delta constraint because it is not reflected in the state. It is not added to the state simply because there is no reason to at this point in the computation. The future of the computation—`ctn` and `resolve`—has already been exhausted in the context of solving the disjuncts. All that is left to do is return the solved values, and adding the disjunction to the store at this point would at best yield the same result as adding it at the end, and at worst create unnecessary work as the state is discarded by a higher level disjunction. Avoiding this extra work is the primary reason the return signature divides constraints into those in the state and those not yet added, and could be simplified to a single return value in less optimized implementations.

3.6 Attributed Variables

Once the constraints have been sufficiently solved, they must be added back to the constraint store so the search can progress. For simple implementations that recheck all constraints at each step, this poses no issue. However, many implementations use a version of attributed variables whereby constraints in the store are indexed by the variables on which they depend. When those variables are modified, either by unification or by the addition of another constraint, the constraints already indexed under that variable can be rechecked without wasting effort on unrelated constraints. The only question, then, is on which variables does a given constraint depend?

With the exception of disjunction, this question is mostly straightforward. Primitive constraints such as unification depend on all of their variables[9], while negation and conjunction depend on all of the attributed variables of their children. Because the store itself can be viewed as a conjunction of all the constraints it contains, storing a conjunction directly in the store can be simplified to storing all of its children independently.

Disjunctions are the more difficult case, and the variables on which they depend themselves depend on the level of solving performed. For a level 2 solver, disjunctions beginning with a non

[8] If the disjunction is right-branching, checking left before right will terminate after the first non-failing, non-unifying left-hand side. This left-branching property is the default generated by **conde**, but can be foiled in the current implementation by adversarial programming, compromising efficiency but not correctness.

[9] Implementations that assign variables a unique, ordered id can simplify this by standardizing the store to only add constraints to the lowest id, for instance.

unifying disjunct depend only on the variables in that disjunct, while others may depend on more. In essence, whichever disjuncts were solved by the solver must contribute their attributed variables to the overall disjunction. The ability to determine after the fact how many disjuncts were solved leads to a potentially delicate computation that depends heavily on the representation used for the solved form of the disjunction. If too few disjuncts are searched for variables, the solver will not preserve the invariants of its level of solving. Too many and the store will assign the constraint to stale variables that may already be bound.

Once the attributed variables have been determined, the current implementation copies the constraint to each key in the store.

4 FUTURE WORK

The implementation described in this paper is the subject of active and ongoing research. At present, two particular concerns, the comparison of solving levels and the semantics of `fresh`, are the most promising topics for further work.

4.1 Solving Levels

As described in Section 3.5.1, there are a number of possible ways to define the solving of disjunctions. While the current implementation can best be understood as a version of level 2 with some additional optimizations, concerns related to those raised in Section 3.6 raise questions about when and whether different levels might be preferable. The rationale for a level 2 solver is that it is essentially lazy. It avoids unnecessary work beyond that required to check for failure or for unification. However, such laziness in a non-deterministic context leads to a host of potential issues.

The implementation described in this paper distributes the continuation conjunctions over the disjuncts of each disjunction. This yields combinatorial solved forms for disjunctions as each disjunct must contain identical copies of any constraints in the continuation that do not directly interact with those in the disjunct. This problem can be solved by expanding the factoring of shared unifications to all constraint types. For instance $(x = 1 \vee x = 2) \wedge y \neq 1$ solves to $(x = 1 \wedge y \neq 1) \vee (x = 2 \wedge y \neq 1)$, which can be factored back into $y \neq 1 \wedge (x = 1 \vee x = 2)$. This avoids the need to solve $y \neq 1$ repeatedly in each disjunct and makes the constraints returned as answers to the user more intelligible.

However, such factoring depends on solved forms being structurally comparable. Several factors in the current implementation conspire against this condition, including the opaqueness of `fresh` closures, the laziness of a level 2 solver, and the current structure of the constraint store. Current implementations of attributed variables often attribute copies of disequality constraints such as $x \neq y$ to both x and y. However, when x is unified, this leaves a stale $y \neq x$ constraint in the store unless additional effort is expended to simplify other constraints. When the only constraint with this property is the simple disequality, the problem rarely becomes critical. However, in the presence of more complex disjunctions such as those described in this paper, such stale constraints often frustrate the ability to factor entire disjunctions out of higher level disjunctions and create rapidly expanding constraints. Even a pair of `absentos` in the context of a relational interpreter can create massive constraints, with implications for both performance and interpretability. We believe these issues can be resolved by finding the right balance of solving level and constraint store architecture, and this work remains ongoing.

4.2 Fresh

To this point, this paper has glossed over the use and implementation of `fresh`, although it is one of the most interesting avenues for further research. The central problem with implementing `fresh` in a constraint context is deciding when to stop the search. Without `fresh`, constraints are

essentially finite, and the depth-first constraint interpreter need not worry about divergence. With the introduction of **fresh**, however, it is necessary to clarify how that determination can be made.

The present implementation partially punts on this question by replacing most uses of **fresh** with a pattern matching form based on Donahue [10]. Whenever **fresh** is used to destructure an argument to the relation, the pattern match form `matcho` can be used instead, for example `matcho ([x (a . d]) . . .)` destructures the variable `x` and unifies it with the pair of fresh variables `a` and `d` before executing the goals signified by … in that lexical context. Unlike **fresh**, it is clear on which variables `matcho` depends, and the constraint can simply be suspended on any one of its variables until all have been bound, at which point it runs its internal goals. If every recursive relation in a program is wrapped in `matcho` rather than **fresh**, the depth first interpreter need not concern itself with halting, as any free variable will halt further recursion.

The current implementation, at the time of writing, does not support the use of classic **fresh** in recursion, although previous versions of the code base have done so. Doing so simply requires expanding the **fresh** until the interpreter encounters a disjunction with a non-failing branch, or in the case of a level 2 interpreter, a disjunction that can be confirmed not to share unifications in common between its disjuncts. This requires additional solving beyond the `matcho`, and in general `matcho` has proven easier to optimize than opaque **fresh**.

In either case, however, **fresh** presents an interesting problem for future research. Namely, that of how to negate it. Previous work has proposed a variety approaches to assigning semantics to **fresh** under negation or quantification [7, 14, 16, 19]. We are actively exploring whether any of these approaches might fit within the current framework. To clarify the issues at play, refer to the implementations of `presento` and `absento` in Sections 2.1.4 and 2.3.4 respectively. In theory, these should be inverses of one another, and so each should be equivalent to the negation of the other. However, note that the two are slightly structurally incongruous.

To begin with, there is no analog in `presento` for the clause in `absento` that succeeds when the term is not a pair. This clause itself is strange in that it requires the invocation of a type constraint entirely outside the system of equalities and disequalities. The reason for this is that the `pairo` constraint supplies, in this context, a missing semantic notion of universal quantification. Conceptually, negating the **fresh** clause in `presento` must mean either that the term is not present in the car or cdr of the current pair or that the term itself is not a pair and cannot be destructured. Negating this **fresh** form must therefore produce a disjunction between the equivalent destructuring form in `absento` as well as a universally quantified term that succeeds if the term cannot be destructured as a pair no matter what the car or cdr. This is different from saying simply that the term disequals a given pair with a fresh car and cdr since such a constraint will still succeed for a pair and simply apply the constraints that its car and cdr must not be some arbitrary free variables.

Moreover, although the recursive calls are clear negations of one another—a negated conjunction transforming into a disjunction of negations or vice versa—the unification that destructures the pair remains unchanged in both, which is difficult to square with a naive semantics in which this is simply another clause conjoined to the recursive calls. This suggests, moreover, that the entire construct of introducing the fresh variables and using them to destructure the input list is a single conceptual operation that requires its own semantics, and perhaps its own form. Combining these two clauses into a single destructuring form, either for the programmer or implicitly in the compiler, might allow for its direct negation. Naively negating the goal returned from `matcho` is sufficient to negate some simpler list constraints, but the case of `absento` and `presento` hints that a more general notion of negation may be required that takes into account the pattern matching component.

Fundamentally, **fresh** seems to encompass multiple separate semantic notions. One is destructuring input pairs defined outside the scope of the **fresh**, and another is constraining subrelations within the **fresh** itself to have the same value for some subset of their parameters. By using the

same form for both, `fresh` creates issues for optimization, and may complicate the question of quantification or negation in miniKanren. Work on this subject remains an open area of research and deserves further study.

5 RELATED WORK

Within the domain of miniKanren research, this paper is most closely in conversation with prior work on constraint authoring frameworks [1, 11]. Unlike these approaches, which facilitate the development of domain specific constraints that make heavy use of specialized representations, this paper presents a strategy for leveraging only the core operators of miniKanren to express a wide variety of constraints that have to this point required such specialized implementations. The benefit of the present approach is that it greatly lowers the barrier to authoring constraints that can be expressed within this framework not only by uniformly handling constraint optimization and interoperation, but also by allowing the expression of constraints in miniKanren, which is particularly well suited to expressing constraints on structures that are themselves necessarily expressible in miniKanren. That said, much work remains to be done on bridging the gap and allowing such constraint authoring frameworks to interoperate with the system presented in this paper to allow for the expression of constraints that lie outside of miniKanren's core representational facilities.

More generally, the solving of simultaneous equations and disequations within the framework of logic programming has developed an extensive literature since its introduction [8]. This early work has been surveyed in Comon [9]. The central design of the solver proposed in this paper in particular generalizes the disequality constraint solver originally proposed by Bürckert [5] and further elaborated upon in Buntine and Bürckert [4], which was subsequently adapted for miniKanren by Byrd [6].

The strategy for avoiding unnecessary constraint checking by assigning constraints to specific variables that may make them unsatisfiable if bound or further constrained is based on what can be viewed as an implementation of attributed variables, albeit in a functional style [17]. Attributed variables, roughly, offer a general means to associate additional information with specific variables, and have found particular application in extending logic programming languages with constraint systems, as is being done here [12, 13]. The original approach to attributing disequalities to variables on which this paper builds originated with Ballantyne et al [2]

This paper also engages to a lesser extent with previous work in miniKanren concerned with the semantics of negation, universal quantification, and `fresh` [7, 14, 16, 18, 19]. In particular, it outlines some desiderata for a concept of universal quantification and associated negation of `fresh` forms and outlines some considerations and motivations that may offer a starting point for future work on the subject rooted in the relevant miniKanren literature on pattern matching forms that present alternative eliminators for destructuring input data and introducing fresh variables [10, 15].

6 CONCLUSION

This paper introduced an extension to miniKanren that allows for the interpretation of goals as constraints, and used this extension to implement a wide variety of useful constraints. However, in so doing, it raised a number of questions that remain open areas for further exploration.

Much work remains to be done on the constraint system itself, from clarifying and confirming the correctness of the various possible levels of constraint simplification enabled by the system's central premise, to further optimizing the representations used for constraints and constraint solving procedures. Questions remain too regarding the best paths for integrating the present work with past work developing constraint authoring frameworks and highly specialized constraints. This work has also raised fundamental question about the semantics of negation and quantification

in miniKanren that cut across a number of research areas. Finally, given the range of constraints this and future related systems make it possible to express, however, it is also worth wondering what kind of applications they may enable, from variations on relational interpretation to as yet unresearched domains. In particular, one of the motivating cases driving this research has been the prospect of running entire complex relations such as relational interpreters and relational type inferencers as constraints, and studying the effect this might have on the ability to compose such relations efficiently by letting the constraint system decompose and reorder them. Further work on the current implementation is required before such experiments can be undertaken.

Because our constraint solver reuses representations and algorithms that already exist in most miniKanren implementations, and particularly those that already use first order representations of goals, and moreover because our solver replaces much of the code dedicated to implementing individual constraints, the implementation burden on top of an existing miniKanren system is relatively minimal. It is therefore our hope that this work can help facilitate the more rapid exploration and prototyping of new types of constraints and the new applications they enable.

7 ACKNOWLEDGMENTS

We thank Will Byrd for discussions of early versions of this idea and Evgenii Moiseenko for clarifying some points of previous work. We also thank the anonymous reviewers for their suggestions.

REFERENCES

[1] Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. 2011. cKanren: miniKanren with Constraints. (2011).

[2] Michael Ballantyne et al. 2020. Faster miniKanren [Source Code]. (2020). https://github.com/michaelballantyne/faster-miniKanren

[3] David C Bender, Lindsey Kuper, William E Byrd, and Daniel P Friedman. 2009. Efficient Representations for Triangular Substitutions: a Comparison in MiniKanren. *Unpublished manuscript* (2009).

[4] Wray L Buntine and Hans-Jürgen Bürckert. 1994. On Solving Equations and Disequations. *Journal of the ACM (JACM)* 41, 4 (1994), 591–629.

[5] Hans-Jürgen Bürckert. 1988. Solving disequations in equational theories. In *9th International Conference on Automated Deduction: Argonne, Illinois, USA, May 23–26, 1988 Proceedings 9*. Springer, 517–526.

[6] William Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph. D. Dissertation. Indiana University.

[7] William. E. Byrd. 2013. Relational Synthesis of Programs. webyrd.net/cl/cl.pdf

[8] A Colmerauer. 1984. Equations and Inequations on Finite and Infinite Trees. In *Proc. of the International Conference on Fifth Generation*.

[9] Hubert Comon. 1991. Disunification: a Survey. (1991).

[10] Evan Donahue. 2021. Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables. *miniKanren and Relational Programming Workshop* (2021).

[11] Daniel P Friedman and Jason Hemann. 2017. A Framework for Extending microKanren with Constraints. In *Proceedings of the 2017 Workshop on Scheme and Functional Programming*.

[12] Christian Holzbaur. 1990. *Specification of Constraint Based Inference Mechanisms Through Extended Unification.* Ph. D. Dissertation. University of Vienna.

[13] Christian Holzbaur. 1992. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *Programming Language Implementation and Logic Programming: 4th International Symposium, PLILP'92 Leuven, Belgium, August 26–28, 1992 Proceedings 4*. Springer, 260–268.

[14] Ende Jin, Gregory Rosenblatt, Matthew Might, and Lisa Zhang. 2021. UniversaL Quantification and Implication in MiniKanren. In *miniKanren and Relational Programming Workshop*. 12.

[15] Andrew W Keep, Michael D Adams, Lindsey Kuper, William E Byrd, and Daniel P Friedman. 2009. A Pattern Matcher for MiniKanren or How to Get into Trouble with CPS Macros. *Technical Report CPSLO-CSC-09-03* (2009), 37.

[16] Dmitry Kosarev, Daniil Berezun, and Peter Lozov. 2022. Wildcard Logic Variables. In *miniKanren and Relational Programming Workshop*.

[17] Serge Le Huitouze. 1990. A New Data Structure for Implementing Extensions to Prolog. In *Programming Language Implementation and Logic Programming: International Workshop PLILP'90 Linköping, Sweden, August 20–22, 1990*

Proceedings 2. Springer, 136–150.

[18] Weixi Ma and Daniel P Friedman. 2021. A New Higher-Order Unification Algorithm for λKanren. In *miniKanren and Relational Programming Workshop*. 113.

[19] Evgenii Moiseenko. 2019. Constructive Negation for MiniKanren. In *Proceedings of the miniKanren and Relational Programming Workshop*.

[20] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-Order MiniKanren Representation: Great for Tooling and Search. In *Proceedings of the miniKanren and Relational Programming Workshop*. 16.

Semi-Automated Direction-Driven Functional Conversion

EKATERINA VERBITSKAIA, JetBrains Research, Serbia
IGOR ENGEL, JetBrains Research, Germany
DANIIL BEREZUN, JetBrains Research, Netherlands

One of the most attractive applications of relational programming is inverse computation. It offers an approach to solving complex problems by transforming verifiers into solvers with relatively low effort. Unfortunately, inverse computation often suffers from interpretation overhead, leading to subpar performance compared to direct program inversion. A prior study introduced a functional conversion scheme capable of creating inversions of miniKanren specifications with respect to a known fixed direction. This paper expands upon it by providing a semi-automated functional conversion algorithm. Our evaluation demonstrates a significant performance improvement achieved through functional conversion.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**.

Additional Key Words and Phrases: program inversion, inverse computations, relational programming, functional programming, conversion

1 INTRODUCTION

One of the most attractive applications of relational programming is *inverse computation*. It is helpful, when the program being inverted is a relational interpreter of some sort: this way an interpreter for a programming language may be used for program synthesis, a type checker — to solve type inhabitation problem and so on [3, 4]. Constructing relational interpreters out of functional implementations can be done automatically by relational conversion [5]. miniKanren along with relational conversion are capable of inverse computations. However, it is important to note that inverse computations exhibit lower performance compared to directly executing an inversion of the original program due to the interpretation overhead [1, 2].

Relational programs do not exist on their own: they are a part of a host program, which utilizes query results in some way. The host languages are not expected to be able to process logic variables, nondeterminism and other aspects of relational computations. The host program usually only deals with a finite subset of answers, which have been reified into a ground representation, meaning they do not include any logic variables.

When a relation is expected to produce ground answers, and the direction in which it is intended to be run is known, then it becomes possible to convert it into a function which may execute significantly faster than its relational counterpart. Performance improvement comes from reducing interpretation overhead as well as replacing expensive unifications with considerably faster equality checks, assignments and pattern matches of a host language. An informal functional conversion scheme was introduced in the paper [9]. We are building upon this research effort, presenting a semi-automatic functional conversion algorithm and implementation for a minimal core relational programming language microKanren. This paper focuses on converting to the target languages of Haskell and OCaml, although other languages can also be considered as potential target languages. Our evaluation showed performance improvement of 2.5 times for propositional formulas synthesis and up to 3 orders of magnitude improvement for relations over Peano numbers.

Authors' addresses: Ekaterina Verbitskaia, kajigor@gmail.com, JetBrains Research, Belgrade, Serbia; Igor Engel, igorengel@mail.ru, JetBrains Research, Bremen, Germany; Daniil Berezun, daniil.berezun@jetbrains.com, JetBrains Research, Amsterdam, Netherlands.

2 BACKGROUND

In this section, we give the abstract syntax of microKanren version used in this paper and describe a concept of modes which was developed earlier for other logic languages.

2.1 Normal Form Abstract Syntax of microKanren

To simplify the functional conversion scheme, we consider microKanren relations to be in the superhomogeneous normal form used in the mercury programming language [7]. Converting an arbitrary microKanren relation into the normal form is a simple syntactic transformation, which we omit.

In the normal form, a term is either a variable or a constructor application which is flat and linear. Linearity means that arguments of a constructor are distinct variables. To be flat, a term should not contain any nested constructors. Each constructor has a fixed arity n. Below is the abstract syntax of the term language over the set of variables V.

$$\mathcal{T}_V = V \cup \{C_n\,(x_1, \ldots, x_n) \mid x_i \in V; i \neq j \Rightarrow x_i \neq x_j\}$$

Whenever a term which does not adhere to this form is encountered in a unification or as an argument of a call, it is transformed into a conjunction of several unifications, as illustrated by the following examples:

$$\begin{aligned} C\,(x_1, x_2) \equiv C\,(C\,(y_1, y_2), y_3) &\Rightarrow x_1 \equiv C\,(y_1, y_2) \wedge x_2 \equiv y_3 \\ C\,(C\,(x_1, x_2), x_3) \equiv C\,(C\,(y_1, y_2), y_3) &\Rightarrow x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3 \\ x \equiv C\,(y, y) &\Rightarrow x \equiv C\,(y_1, y_2) \wedge y_1 \equiv y_2 \\ add^o\,(x, x, z) &\Rightarrow add^o\,(x_1, x_2, z) \wedge x_1 \equiv x_2 \end{aligned}$$

Unification in the normal form is restricted to always unify a variable with a term. We also prohibit using disjunctions inside conjunctions. The normalization procedure declares a new relation whenever this is encountered.

The complete abstract syntax of the microKanren language used in this paper is presented in figure 1.

$\mathcal{D}_V^N$	:	$R_n\,(x_1, \ldots, x_n) = \mathbf{Disj}_V, x_i \in V$	normalized relation definition
$\mathbf{Disj}_V$	:	$\bigvee\,(c_1, \ldots, c_n), c_i \in \mathbf{Conj}_V$	normal form
$\mathbf{Conj}_V$	:	$\bigwedge\,(g_1, \ldots, g_n), g_i \in \mathbf{Base}_V$	normal conjunction
$\mathbf{Base}_V$	:	$V \equiv \mathcal{T}_V$	flat unification
	|	$R_n\,(x_1, \ldots, x_n), x_i \in V, i \neq j \Rightarrow x_i \neq x_j$	flat call

Fig. 1. Abstract syntax of microKanren in the normal form

2.2 Modes

A mode generalizes the concept of a direction; this terminology is commonly used in the conventional logic programming community. In its most primitive form, a mode specifies which arguments of a relation will be known at runtime (input) and which are expected to be computed (output). Several logic programming languages have mode systems used for optimizations, with mercury standing out among them. mercury[1] is a modern functional-logic programming language with a complicated mode system capable not only of describing directions, but also specifying if a relation in the given mode is deterministic, among other things [6].

[1]Website of the mercury programming language: https://mercurylang.org/

Given an annotation for a relation, mode inference determines modes of each variable of the relation. For some modes, conjunctions in the body of a relation may need reordering to ensure that consumers of computed values come after the producers of said values so that a variable is never used before it is bound to some value. In this project, we employed the least complicated mode system, in which variables may only have an *in* or *out* mode. A mode maps variables of a relation to a pair of the initial and final instantiations. The mode *in* stands for $g \to g$, while *out* stands for $f \to g$. The instantiation f represents an unbound, or *free*, variable, when no information about its possible values is available. When the variable is known to be *ground*, its instantiation is g.

In this paper, we call a pair of instantiations a mode of a variable. figure 2 shows examples of the normalized microKanren relations with modes inferred for the forward and backward directions. We use superscript annotation for variables to represent their modes visually. Notice the different order of conjuncts in the bodies of the add^o relation in different modes.

```
let double^o x^{g→g} r^{f→g} =
  addo^o x_1^{g→g} x_2^{g→g} r^{f→g} ∧
  x_1^{g→g} ≡ x_2^{g→g}

let rec add^o x^{g→g} y^{g→g} z^{f→g} =
  (x^{g→g} ≡ 0 ∧ y^{g→g} ≡ z^{f→g}) ∨
  (x^{g→g} ≡ S x_1^{f→g} ∧
   add^o x_1^{g→g} y^{g→g} z_1^{f→g} ∧
   z^{f→g} ≡ S z_1^{g→g})
```

(a) Forward direction

```
let double^o x^{f→g} r^{g→g} =
  addo^o x_1^{f→g} x_2^{f→g} r^{g→g} ∧
  x_1^{g→g} ≡ x_2^{g→g}

let rec add^o x^{f→g} y^{f→g} z^{g→g} =
  (x^{f→g} ≡ 0 ∧ y^{f→g} ≡ z^{g→g}) ∨
  (z^{f→g} ≡ S z_1^{g→g} ∧
   add^o x_1^{f→g} y^{f→g} z_1^{g→g} ∧
   x^{f→g} ≡ S x_1^{g→g})
```

(b) Backward direction

Fig. 2. Normalized doubling and addition relations with mode annotations

3 FUNCTIONAL CONVERSION FOR MICROKANREN

In this section, we describe the functional conversion algorithm. The reader is encouraged to first read the paper [9] on the topic, which introduces the conversion scheme on a series of examples.

Functional conversion is done for a relation with a concrete fixed direction. The goal is to create a function which computes the same answers as microKanren would, not necessarily in the same order. Since the search in microKanren is complete, both conjuncts and disjuncts can be reordered freely: interleaving makes sure that no answers would be lost this way. Moreover, the original order of the subgoals is often suboptimal for any direction but the one which the programmer had in mind when they encoded the relation. When the relational conversion is used to create a relation, the order of the subgoals only really suits the forward direction, in which the relation is often not intended to be run (in this case, it is better to run the original function).

The mode inference results in the relational program with all variables annotated by their modes, and all base subgoals ordered in a way that further conversion makes sense. Conversion then produces functions in the intermediate language. It may then be pretty printed into concrete functional programming languages, in our case Haskell and OCaml.

3.1 Mode Inference

We employ a simple version of mode analysis to order subgoals properly in the given direction. The mode analysis makes sure that a variable is never used before it is associated with some value.

It also ensures that once a variable becomes ground, it never becomes free, thus the value of a variable is never lost. The mode inference pseudocode is presented in listing 1.

```
1  modeInfer (R_i (x_1, ..., x_{k_i}) ≡ body) = (R_i (x_1, ..., x_{k_i}) ≡ (modeInferDisj body))
2
3  modeInferDisj (⋁ (c_1, ..., c_n)) = ⋁(modeInferConj c_1, ..., modeInferConj c_n)
4
5  modeInferConj (⋀ (g_1, ..., g_n)) =
6    let (picked, theRest) = pickConjunct([g_1, ..., g_n]) in
7    let moddedPicked = modeInferBase picked in
8    let moddedConjs = modeInferConj (⋀ theRest) in
9    ⋀(moddedPicked : moddedConjs)
10
11 pickConjunct goals =
12   pickGuard goals <|>
13   pickAssignment goals <|>
14   pickMatch goals <|>
15   pickCallWithGroundArguments goals <|>
16   pickUnificationGenerator goals <|>
17   pickCallGenerator goals
```

Listing 1. Mode inference pseudocode

Mode inference starts by initializing modes for all variables in the body of the given relation according to the given direction. All variables that are among arguments are annotated with their *in* or *out* modes, while all other variables get only their initial instantiations specified as f.

Then the body of the relation is analyzed (see line 1). Since the body is normalized, it can only be a disjunction. Each disjunct is analyzed independently (see line 3) because no data flow happens between them.

Analyzing conjunctions involves analyzing subgoals and ordering them. Let us first consider mode analysis of unifications and calls, and then circle back to the way we order them. Whenever a base goal is analyzed, all variables in it have some initial instantiation, and some of them also have some final instantiation. Mode analysis of a base goal boils down to making all final instantiations ground.

When analyzing a unification, several situations may occur. Firstly, every variable in the unification can be ground, as in $x^{g\to g} \equiv O$ or in $y^{g\to ?} \equiv z^{g\to ?}$ (here ? is used to denote that a final instantiation is not yet known). We call this case *guard*, since it is equivalent to checking that two values are the same.

The second case is when one side of a unification only contains ground variables. Depending on which side is ground, we call this either *assignment* or *match*. The former corresponds to assigning the value to a variable, as in $x^{f\to ?} \equiv S\, x_1^{g\to g}$ or $x^{g\to g} \equiv y^{f\to ?}$. The latter — to pattern matching with the variable as the scrutinee, as in $x^{g\to g} \equiv S\, x_1^{f\to ?}$. Notice that we allow for some variables on the right-hand side to be ground in matches, given that at least one of them is free.

The last case occurs when both the left-hand and right-hand sides contain free variables. This does not translate well into functional code. Any free logic variable corresponds to the possibly infinite number of ground values. To handle this kind of unification, we propose to use *generators* which produce all possible ground values a free variable may have.

We base our ordering strategy for conjuncts on the fact that these four different unification types have different costs. The guards are just equality checks which are inexpensive and can reduce the search space considerably. Assignments and matches are more involved, but they still take much less effort than generators. Moreover, executing non-generator conjuncts first can make some of the variables of the prospective generator ground thus avoiding generation in the end. This is the base reasoning which is behind our ordering strategy.

The function `pickConjunct` selects the base goal which is least likely to blow up the search space. The right-associative function <|> used in lines 12 through 16 is responsible for selecting the base goals in the order described. The function first attempts to pick a base goal with its first argument, and only if it fails, the second argument is called. As a result, `pickConjunct` first picks the first guard unification it can find (`pickGuard`). If no guard is present, then it searches for the first assignment (`pickAssignment`), and then for the match (`pickMatch`). If all unifications in the conjunction are generators, then we search for relation calls with some ground arguments (`pickCallWithGroundArguments`). If there are none, then we have no choice but selecting a generating unification (`pickUnificationGenerator`) and then a call with all arguments free (`pickCallGenerator`).

Once one conjunct is picked, it is analyzed (see line 7). The picked conjunct may instantiate new variables, thus this information is propagated onto the rest of the conjuncts. Then the rest of the conjuncts is mode analyzed as a new conjunction (see line 8). If any new modes for any of the relations are encountered, they are also mode analyzed.

It is worth noticing that any relation can generate infinitely many answers. We cannot judge the relation to be such generator solely by its mode: for example, the addition relation in the mode $\mathsf{add}^o\ \mathsf{x}^{g\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{f\to g}$ generates an infinite stream, while $\mathsf{add}^o\ \mathsf{x}^{f\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{g\to g}$ does not.

3.2 Conversion into Intermediate Representation

To represent nondeterminism, our functional conversion uses the basis of μKanren — the stream data structure. A relation is converted into a function with n arguments which returns a stream of m-tuples, where n is the number of the input arguments, and m — the number of the output arguments of the relation. Since stream is a monad, functions can be written elegantly in Haskell using do-notation (see figure 4). We use an intermediate representation which draws inspiration from Haskell's do-notation, but can then be pretty-printed into other functional languages. The abstract syntax of our intermediate language is shown in figure 3. The conversion follows quite naturally from the modded relation and the syntax of the intermediate representation.

$$
\begin{array}{rcll}
\mathcal{F}_V & = & \textbf{Sum}\ [\mathcal{F}_V] & \text{concatenation of streams} \\
& | & \textbf{Bind}\ [([V], \mathcal{F}_V)] & \text{monadic bind for streams} \\
& | & \textbf{Return}\ [\mathcal{T}_V] & \text{return of a tuple of terms} \\
& | & \textbf{Guard}\ (V, V) & \text{equality check} \\
& | & \textbf{Match}_V\ (\mathcal{T}_V, \mathcal{F}_V) & \text{match a variable against a pattern} \\
& | & R_n([V], [G]) & \text{function call} \\
& | & \textbf{Gen}_G & \text{generator}
\end{array}
$$

Fig. 3. Abstract syntax of the intermediate language $\mathcal{F}$

A body of a function is formed as an interleaving concatenation of streams (**Sum**), each of which is constructed from one of the disjuncts of the relation. A conjunction is translated into a sequence of bind statements (**Bind**): one for each of the conjuncts and a return statement (**Return**) in the

end. A bind statement binds a tuple of variables (or nothing) with values taken from the stream in the right-hand side.

A base goal is converted into a guard (**Guard**), match (**Match**), or function call, depending on the goal's type. Assignments are translated into binds with a single return statement on the right. Notice, that a match only has one branch. This branch corresponds to a unification. If the scrutinee does not match the term it is unified with, then an empty stream is returned in the catch-all branch. If a term in the right-hand side of a unification has both *out* and *in* variables, then additional guards are placed in the body of the branch to ensure the equality between values bound in the pattern and the actual ground values.

Generators (**Gen**) are used for unifications with free variables on both sides. A generator is a stream of possible values for the free variables, and it is used for each variable from the right-hand side of the unification. The variable from the left-hand side of the unification is then simply assigned the value constructed from the right-hand side. Our current implementation works with an untyped deeply embedded microKanren, in which there is not enough information to produce generators automatically. We decided to delegate the responsibility to provide generators to the user: a generator for each free variable is added as an argument of the relation. When the user is to call the function, they have to provide the suitable generators.

4 EXAMPLES

In this section, we provide some examples which demonstrate mode analysis and conversion results.

4.1 Multiplication Relation

Figure 4 shows the implementation of the multiplication relation $\texttt{mul}^o$, the mode analysis result for mode $\texttt{mul}^o\ \texttt{x}^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}^{g\to g}$, and the results of functional conversion into Haskell and OCaml.

Note that the unification comes last in the second disjunct. This is because before the two relation calls are done, both variables in the unification are free. Our version of mode inference puts the relation calls before the unification, but the order of the calls depends on their order in the original relation. There is nothing else our mode inference uses to prefer the order presented in the figure over the opposite: $\texttt{mul}^o\ \texttt{x}_1^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g} \wedge \texttt{add}^o\ \texttt{y}^{g\to g}\ \texttt{z}_1^{g\to g}\ \texttt{z}^{g\to g}$. However, it is possible to derive this optimal order, if determinism analysis is employed: $\texttt{add}^o\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g}\ \texttt{z}^{g\to g}$ is deterministic while $\texttt{mul}^o\ \texttt{x}_1^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g}$ is not. Putting nondeterministic computations first makes the search space larger, and thus should be avoided if another order is possible.

Functional conversions in both languages are similar, modulo the syntax. The Haskell version employs do-notation, while we use let-syntax in the OCaml code. Both are syntactic sugar for monadic computations over streams. We use the following convention to name the functions: we add a suffix to the relation's name whose length is the same as the number of the relation's arguments. The suffix consists of the letters `I` and `O` which denote whether the argument in the corresponding position is *in* or *out*. The function `msum` uses the interleaving function `mplus` to concatenate the list of streams constructed from disjuncts. To check conditions, we use the function `guard` which fails the monadic computation if the condition does not hold. Note that even though patterns for the variable `x0` in the function `addoIOI` are disjunct in two branches, we do not express them as a single pattern match. Doing so would improve readability, but it does not make a difference when it comes to the performance, according to our evaluation.

4.2 The Mode of Addition Relation which Needs a Generator

Consider the example of the addition relation in mode $\texttt{add}^o\ \texttt{x}^{g\to g}\ \texttt{y}^{f\to g}\ \texttt{z}^{f\to g}$ presented in figure 5. The unification in the first disjunct of this relation involves two free variables. We use a generator

```
let rec mul^o x y z = conde [
  (x ≡ 0 ∧ z ≡ 0);
  (fresh (x1 z1)
    (x ≡ S x1 ∧
     add^o y z1 z ∧
     mul^o x1 y z1)) ]
```

(a) Implementation in miniKanren

```
let rec mul^o x^{f→g} y^{g→g} z^{g→g} =
  (x^{f→g} ≡ 0 ∧ z^{g→g} ≡ 0) ∨
  (add^o y^{g→g} z1^{f→g} z^{g→g} ∧
   mul^o x1^{f→g} y^{g→g} z1^{g→g} ∧
   x^{f→g} ≡ S x1^{g→g})
```

(b) Mode inference result

```
muloOII x1 x2 = msum
  [ do { let {x0 = 0}
       ; guard (x2 == 0)
       ; return x0 }
  , do { x4 ← addoIOI x1 x2
       ; x3 ← muloOII x1 x4
       ; let {x0 = S x3}
       ; return x0 } ]
addoIOI x0 x2 = msum
  [ do { guard (x0 == 0)
       ; let {x1 = x2}
       ; return x1 }
  , do { x3 ← case x0 of
            { S y3 → return y3
            ; _ → mzero }
       ; x4 ← case x2 of
            { S y4 → return y4
            ; _ → mzero }
       ; x1 ← addoIOI x3 x4
       ; return x1 } ]
```

(c) Functional conversion into Haskell

```
let rec muloOII x1 x2 = msum
  [ ( let* x0 = return 0 in
      let* _ = guard (x2 = 0) in
      return x0 )
  ; ( let* x4 = addoIOI x1 x2 in
      let* x3 = muloOII x1 x4 in
      let* x0 = return (S x3) in
      return x0 ) ]
and addoIOI x0 x2 = msum
  [ ( let* _ = guard (x0 = 0) in
      let* x1 = return x2 in
      return x1 )
  ; ( let* x3 = match x0 with
        | S y3 → return y3
        | _ → mzero in
      let* x4 = match x2 with
        | S y4 →  return y4
        | _ → mzero in
      let* x1 = addoIOI x3 x4 in
      return x1 ) ]
```

(d) Functional conversion into OCaml

Fig. 4. Multiplication relation

`gen_addoII0_x2` to generate a stream of ground values for the variable `z` which is passed into the function `addII0` as an argument. It is up to the user to provide a suitable generator. One of the possible generators which produces all Peano numbers in order and an example of its usage are presented in figure 5b.

The generators which produce an infinite stream should be inverse eta-delayed in OCaml and other non-lazy languages. Otherwise, the function would not terminate trying to eagerly produce all possible ground values before using any of them.

It is possible to automatically produce generators from the data type of a variable, but it is currently not implemented, as we work with an untyped version of microKanren.

5 EVALUATION

To evaluate our functional conversion scheme, we implemented the proposed algorithm in Haskell. We compared execution time of several OCanren relations in different directions against their

```
let rec add^o x^{g→g} y^{f→g} z^{f→g} =
  (x^{g→g} ≡ 0 ∧ y^{f→g} ≡ z^{f→g}) ∨
  (x^{g→g} ≡ S x_1^{f→g} ∧
   add^o x_1^{g→g} y^{f→g} z_1^{f→g} ∧
   z^{f→g} ≡ S z_1^{g→g})
```

(a) Mode inference result

```
genNat = msum
  [ return 0
  , do { x ← genNat
       ; return (S x) } ]
runAddoII0 x = addoII0 x genNat
```

(b) Generator of Peano numbers

```
addoI00 x0 gen_addoI00_x2 = msum
  [ do { guard (x0 == 0)
       ; (x1, x2) ← do { x2 ← gen_addoI00_x2 ; return (x2, x2) }
       ; return (x1, x2) }
  , do { x3 ← case x0 of { S y3 → return y3 ; _ → mzero }
       ; (x1, x4) ← addoI00 x3 gen_addoI00_x2
       ; let {x2 = S x4} ; return (x1, x2) } ]
```

(c) Functional conversion

Fig. 5. Addition relation when only the first argument is *in*

functional counterparts in the OCaml language. Here we showcase two relational programs and their conversions. The implementation of the functional conversion[2] as well as the execution code[3] can be found on Github.

5.1 Evaluator of Propositional Formulas

In this example, we converted a relational evaluator of propositional formulas: see figure 6. It evaluates a propositional formula `fm` in the environment `st` to get the result `u`. A formula is either a boolean literal, a numbered variable, a negation of another formula, a conjunction or a disjunction of two formulas. Converting it in the direction when everything but the formula is *in* (see figure 6a), allows one to synthesize formulas which can be evaluated to the given value. The conversion of this relation does not involve any generators and is presented in figure 6b.

We ran an experiment to compare the execution time of the relational interpreter vs. its functional conversion. In the experiment, we generated from 1000 to 10000 formulas which evaluate to true and contain up to 3 variables with known values. The results are presented in figure 7. The functional conversion improved execution time of the query about 2.5 times from 724*ms* to 291*ms* for retrieving 10000 formulas.

5.2 Multiplication

In this example, we converted the multiplication relation in several directions and compared them to the relational counterparts: see figure 8. Functional conversion significantly reduced execution time in most directions.

In the forward direction, we run the query `mul^o n 10 q` with `n` in the range from 100 to 1000, and the functional conversion was 2 orders of magnitude faster: 927*ms* vs 9.4*ms* for the largest `n`, see figure 8a. In the direction which serves as division we run the query `mul^o (n /10) q n` with `n` ranging from 100 to 1000. Here, performance improved 3 orders of magnitude: from 24*s* to 0.17*s* for

[2]The repository of the functional conversion project https://github.com/kajigor/uKanren_transformations
[3]Evaluation code https://github.com/kajigor/miniKanren-func

```
let rec eval^o st^{g→g} fm^{f→g} u^{g→g} =
  ( fm^{f→g} ≡ Lit u^{g→g}) ∨
  ( elem^o z^{f→g} st^{g→g} u^{g→g} ∧
    fm^{f→g} ≡ Var z^{g→g} ) ∨
  ( not^o v^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    fm^{f→g} ≡ Neg x^{g→g} ) ∨
  ( or^o v^{f→g} w^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    eval^o st^{g→g} y^{f→g} w^{g→g} ∧
    fm^{f→g} ≡ Disj x^{g→g} y^{g→g}) ∨
  ( and^o v^{f→g} w^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    eval^o st^{g→g} y^{f→g} w^{g→g} ∧
    fm^{f→g} ≡ Conj x^{g→g} y^{g→g}) ∨
```

(a) Mode inference result

```
evaloIOI x0 x2 = msum
  [ do { let {x1 = Lit x2}
       ; return x1 }
  , do { x7 ← elemoOII x0 x2
       ; let {x1 = Var x7}
       ; return x1 }
  , do { x5 ← notoOI x2
       ; x3 ← evaloIOI x0 x5
       ; let {x1 = Neg x3}
       ; return x1 }
  , do { (x5, x6) ← oroOOI x2
       ; x3 ← evaloIOI x0 x5
       ; x4 ← evaloIOI x0 x6
       ; let {x1 = Disj x3 x4}
       ; return x1 }
  , do { (x5, x6) ← andoOOI x2
       ; x3 ← evaloIOI x0 x5
       ; x4 ← evaloIOI x0 x6
       ; let {x1 = Conj x3 x4}
       ; return x1 } ]
```

(b) Functional conversion

Fig. 6. Evaluator of propositional formulas

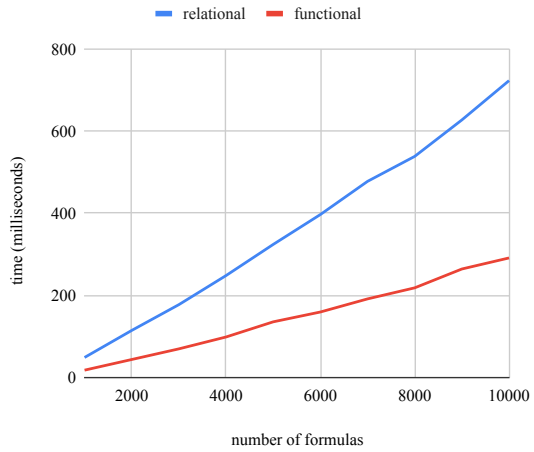
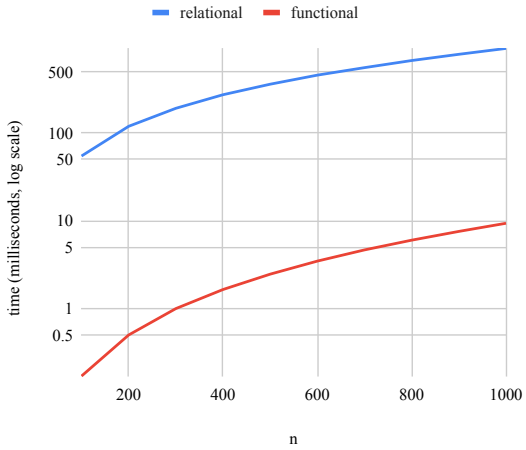


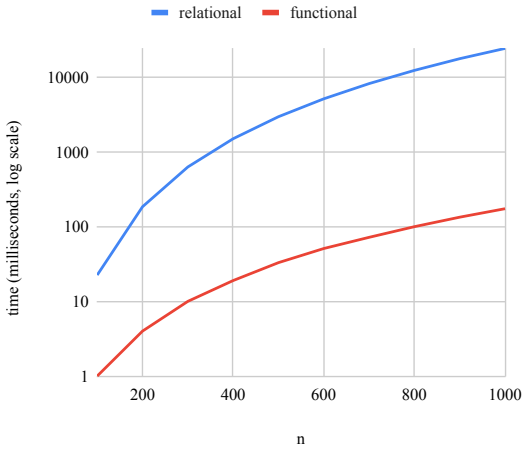
Fig. 7. Execution time of the evaluators of propositional formulas, `eval [true ; false ; true] q true`

the largest n, see figure 8b. Even more impressive was the backward direction `mul^o x^{f→g} y^{f→g} z^{g→g}`. Querying for all 16 pairs of divisors of 1000 (`mul^o q r 1000`) took OCanren about 32.9s, while the functional conversion succeeded in 1.1s.

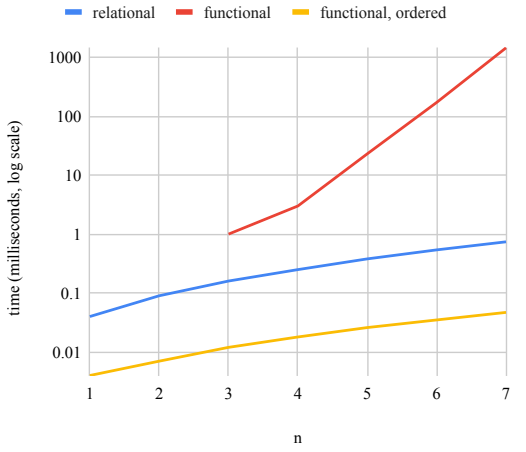
What was surprising was the mode `mul^o x^{g→g} y^{f→g} z^{f→g}`. In this case, the functional conversion was not only worse than its relational counterpart, its performance degraded exponentially with



(a) Multiplication: `mulo n 10 q`



(b) Division: `mulo (n /10) q n`



(c) Generation: `take n (mulo 10 q r)`

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{mul}^o\ \mathtt{x}^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}^{f\to g} = \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{O}\ \wedge\ \mathtt{z}^{f\to g} \equiv \mathtt{O})\ \vee \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g}\ \wedge \\
\qquad \mathtt{add}^o\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \mathtt{z}^{f\to g}\ \wedge \\
\qquad \mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}\ )
\end{array}
$$

(d) Inefficient mode

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{mul}^o\ \mathtt{x}^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}^{f\to g} = \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{O}\ \wedge\ \mathtt{z}^{f\to g} \equiv \mathtt{O})\ \vee \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g})\ \wedge \\
\qquad \mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \wedge \\
\qquad \mathtt{add}^o\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}\ \mathtt{z}^{f\to g}\ )
\end{array}
$$

(e) Efficient mode

Fig. 8. Execution times of the multiplication relation

the number of answers asked. It took almost 1450*ms* to find the first 7 pairs of numbers `q` and `r` such that `10 * q = r`, while OCanren was able to execute the query in 0.74*ms* (see figure 8c). The source of this terrible behavior was the suboptimal order of the calls in the second disjunct of the $\mathtt{mul}^o$ relation in the corresponding mode (see figure 8d). In this case, the call $\mathtt{add}^o\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \mathtt{z}^{f\to g}$ is put first, which generates all possible triples in the addition relation before filtering them by the call $\mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}$. The other order of calls is much better (see figure 8e): it is an order of magnitude faster than its relational source. To achieve the better of these two orders automatically, we delay picking any call with all arguments free. It is not clear if these heuristics are universal.

5.3 Deterministic Directions

Running in some directions, relations produce deterministic results. For example, any forward direction of a relation created by the relational conversion produces a single result, since it mimics the original function. The guard directions are semi-deterministic: they may fail, but if they succeed, they produce a single unit value. If the addition relation is run with one of the first two arguments *out*, it acts as subtraction and is also deterministic.

For such directions, there is no need to model nondeterminism with the Stream monad. Semi-determinism can be expressed with a Maybe monad, while deterministic directions can be converted into simple functions. Our implementation of functional conversion only restricts the computations to be monadic, it does not specify which monad to use. By picking other monads, we can achieve performance improvement. For example, using Maybe for division reduces its execution time 30 times in addition to the 2 orders of magnitude improvement from the functional conversion itself: see figure 9

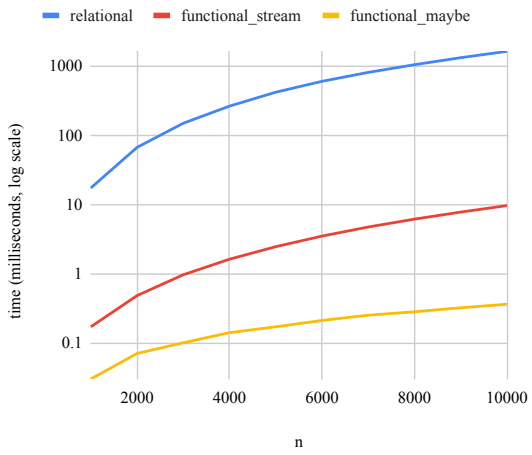


Fig. 9. Execution time of division: `take n (mul q 10 1000)`

6 DISCUSSION

Our experiments indicated that the functional conversion is capable of improving performance of relational computations significantly in the known directions. The improvement stems from eliminating costly unifications in favor of the cheaper equality checks and pattern matches. Besides this, we employed some heuristics which push lower-cost computations to happen sooner while delaying higher-cost ones. It is also possible to take into account determinism of some directions and improve performance of them even more by picking an appropriate monad.

It is not currently clear if the heuristics we used are universal enough. However, it is always safe to run any deterministic computations because they never increase the search space. We believe that it is necessary to integrate determinism check in the mode analysis so that the more efficient modes such as the one presented in figure 8e could be achieved more justifiably.

We also think that further integration with specialization techniques such as partial deduction may benefit the conversion even more [8]. For example, the third argument of the propositional evaluator can be either **true** or **false** . Specializing the evaluator for these two values may help to shave off even more time.

7 CONCLUSION AND FUTURE WORK

In this paper, we described a semi-automatic functional conversion of a microKanren relation with a fixed direction into a functional language. We implemented the proposed conversion and applied it to a set of relations, resulting in significant performance enhancement, as demonstrated in our evaluation. As part of the future work, we plan to augment the mode analysis with a determinism check. We also plan to integrate the functional conversion with specialization techniques such as partial deduction.

REFERENCES

[1] Sergei Abramov and Robert Glück. 2000. Combining Semantics with Non-standard Interpreter Hierarchies. In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, Sanjiv Kapoor and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–213.

[2] Sergei Abramov and Robert Glück. 2002. *Principles of Inverse Computation and the Universal Resolving Algorithm*. Springer Berlin Heidelberg, Berlin, Heidelberg, 269–295. https://doi.org/10.1007/3-540-36377-7_13

[3] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. Association for Computing Machinery, New York, NY, USA, 8—-29. https://doi.org/10.1145/2661103.2661105

[4] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational interpreters for search problems. In *Relational Programming Workshop*. 43.

[5] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed relational conversion. In *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*. Springer, 39–58.

[6] David Overton, Zoltan Somogyi, and Peter J Stuckey. 2002. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 109–120.

[7] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64.

[8] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2021. An Empirical Study of Partial Deduction for miniKanren. *arXiv preprint arXiv:2109.02814* (2021).

[9] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2022. On a Direction-Driven Functional Conversion. In *Relational Programming Workshop*.

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 $fresh$ ($\exists$) in the rule's body. Once we finish the transformation we introduced in section 3.5 of the original rule, all $\exists$ quantifiers over body variables in the rule are turned into $\forall$ 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 ($\exists$) to finding all answers ($\forall$). 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.

klogic: miniKanren in Kotlin

YURY KAMENEV, No affiliation, Russia
DMITRII KOSAREV, St. Petersburg State University, Russia
DMITRY IVANOV, No affiliation, Russia
DENIS FOKIN, No affiliation, Russia
DMITRII BOULYTCHEV, St. Petersburg State University, Russia

One of the distinguishable features of embedded domain-specific languages, like miniKanren, is an easy integration with a host language. Recently, we were asked to port our relational programs in OCaml to Java Virtual Machine (JVM). As a result, we got a miniKanren implementation in Kotlin, which resembles OCanren (miniKanren in OCaml). In this paper, we describe the peculiarities of a relational language implementing on JVM using Kotlin.

CCS Concepts: • **Software and its engineering → Functional languages**; *Constraint and logic languages.*.

Additional Key Words and Phrases: miniKanren implementation, Kotlin, relational programming

1 INTRODUCTION

One of the appealing features of relational programming is an easy interaction between general-purpose and relational programming languages. We often find it useful to prototype a solution in miniKanren, and later improve the performance by splitting big relational programs into a number of relational programs, connected by a functional glue. We usually use OCaml and the OCanren DSL to do this, but recently we were asked to make our work executable on JVM.

The straightforward solution to execute OCanren in JVM would be using an OCamlJava [4] compiler, but this project looks quite dead. Using JNI could be error-prone (for inexperienced JVM users like us). Another approach would be a source-to-source transformation from OCaml to a JVM-friendly language. We left it as a future work, because, at first, it is difficult for us to estimate the difficulty of the task, and, second, it requires a working implementation of a relational DSL in JVM.

There are plenty of relational libraries for JVM languages, e.g. for Scala (Scalogno [1]) and Core.Logic. However, according to TIOBE [2], Kotlin is the most popular JVM language except for Java itself, but Java lacks the syntax needed for a compact miniKanren implementation. In our mind, more professional developers and enthusiasts can give into a new Kotlin library and contribute to it. Our customer, who is interested in relational programming in OCanren as well values Kotlin very much.

As a result, we developed a relational programming library klogic[1] in Kotlin. The reference implementation was OCanren, but we needed to do a few things differently because of JVM quirks. In this paper we report the main technical decisions about Kotlin implementation, recollect the peculiarities of the OCanren implementation in OCaml, discuss the programming experience from an OCaml developer point of view, and do some benchmarking.

2 OCANREN REMINDER

The klogic implementation is based on OCanren: typed embedding of miniKanren to OCaml. In this section, we describe basic blocks of OCanren, their implementation in klogic is given in the next section. A few features (for example, disequality constraints) in klogic and OCanren are implemented similarly and will be left out of discussion.

[1] https://github.com/UnitTestBot/klogic (access date: 2023-06-15)

Unification is being performed over a tree of pointers, which represent OCaml values in the runtime. Most algebraic values are the nodes of the tree, basic types and algebraic constructors without arguments are the leaves. Inspection and reconstruction of the trees is performed via a low-level non type-safe interface.

No proper OCaml types could not be assigned to values injected into logic domain, because their representation differs. This is done for performance considerations.

Types for logic representations. In typed relational programs you can find three sorts of types: for non-relational ground representations, logic domain, and types of relational search results, that represent values *reified* (extracted) *from logic domain*. For the reified values we use a straightforward solution: we declare the algebraic data type to represent either logic variables or values.

```
type 'a logic = Var of var_idx | Value of 'a
```

In principle, we can use the same `logic` type for representations in a logic domain. It will lead to poor performance, which is demonstrated in [6]. These days OCanren uses slightly different types for a logic domain than the ones described in [6], and we add details about it. For every subvalue embedded into the logic domain, we decorate a type of this subvalue with the predefined type `type 'a ilogic`. For example, integer constants will have in the logic domain type `int ilogic`. The same holds for other types considered primitive in OCanren: strings, integers, floats, booleans. In Scheme context, the closest analogue would be symbols. The injection of primitive values a to logic domain could be done using predefined function `inj`.

```
val inj: 'a -> 'a ilogic
```

The injection of user-defined algebraic data types is more complicated. We want to allow the placement of logic values in all subparts of data type, not only in positions of type variables. To achieve that, we abstract away all concrete type parameters, and get "fully abstract data type". We can parameterize it by substituted types to get the type isomorphic to the original one, or parameterize with `ilogic` types to get the type for logic domain, or parameterize with the type `logic`, to get a type of reified representation. In Listing 2 one can see three types: the one without logic variables, the logic counterpart, and for reified results. One can note how the second type is being constructed by addition of `ilogic` type everywhere, and the reified type by replacing `ilogic` with `logic`.

```
(int * bool, 'a) fa_list as 'a
((int ilogic * bool ilogic) ilogic, 'a) fa_list ilogic as 'a
((int logic * bool logic) logic, 'a) fa_list logic as 'a
```

The injection of user-defined types is essentially an application of a constructor to injected values, followed by the primitive injection `inj`. In the previous implementation [6], the representation of injected types had two type parameters to track ground and reified types of values. This approach requires predefined OCaml functors, one functor per type arity. In the current OCanren implementation, this is not needed, which allows to get rid of functors and control reified type with more flexibility.

Reification. Reification for user data type is a composition of a primitive predefined reifier (for strings, integers, floats, and booleans), a predicate to distinguish variables from values, and a fixpoint combinator to get reifiers for recursive values. All reifiers are two-parametric types which track in the parameters the type in the logic domain, and the type we are going to reify into. In practice, we found it convenient to have two kind of reifiers. The default one reifies to "logic" types, which can represent variables using `logic` type mentioned above. It could be considered a general form of reification. Another one is called *projection*, which reifies from a logic domain to an original

```
type 'a list = [] | (::) of 'a * 'a list
type ('a, 'b) fa_list = Nil | Cons of 'a * 'b
type 'a iso_list = ('a, 'a iso_list) fa_list
type 'a injected_list = ('a, 'a injected_list) fa_list ilogic
type 'a logic_list = ('a, 'a logic_list) fa_list logic
```

Fig. 1. A few examples of types in OCanren using list data type. The `list` type is a default linked list defined in OCaml, conventional constructors have a special treatment in the parser. The `fa_list` is a fully abstract version of a list with constructors renamed. The `iso_list` is a definition via `fa_list` of the type isomorphic to the original list. We can add `ilogic` / `logic` types to the definition of the type `iso_list` to get the types for logic/reified domains.

ground representation, or raises an exception when a free variable is encountered. It reifies to the type without holes for logic variables, and these sort of types is more approachable for integration of functional and relational code. The projection should be used for relations, when we are sure that the answer of relational search is ground.

```
val reify_list : ('a, 'b) Reifier.t ->
  ('a injected_list, 'b logic_list) Reifier.t
val prj_exn_list : ('a, 'b) Reifier.t ->
  ('a injected_list, 'b list) Reifier.t
```

In [6] we track the type of reified values during injection process, and the type of reified values is fixed. Now we outsource to the user the construction of reified values, and the one is empowered to construct values of a desired type. It is possible to project a list from the logic domain to the standard OCaml list type. In the previous approach, we were limited to reification to `'a iso_list`. It required a manual conversion to default OCaml lists, which was cumbersome.

The klogic implementation will reuse these ideas, because both OCanren and klogic are embedded to a typed language. There is no two types of reifiers per every data type in Scheme, a single implicit reifier is enough.

Primitives. In original miniKanren for Scheme, most of the primitives are implemented using a macro system. In OCanren, we apply similar macro for `fresh`, but `conde` is just a function that takes a list of goals. We also have infix binary high-order relations for disjunction and conjunction.

3 IMPLEMENTATION

In this section, we describe peculiarities of unification with user-defined types in klogic.

3.1 Types for logic representation

In the klogic, we introduce a special interface called *Term* to represent the logic domain. There are two types of logic terms — logic variables and logic values (that can store other logic terms inside). Logic variables are represented as inline wrappers of integer index which distinguishes variables; logic values can store any value, logic or not. So, for a simple implementation (that an original Scheme[2]) it would be enough to represent the logic domain by declaring the interface *Term* with one existing inheritor — the inline class *Var* — and allowing users to inherit the *Term* for implementing user's logic types.

[2] https://github.com/michaelballantyne/faster-minikanren (access date: 2023-06-06)

Unfortunately, this implementation would have an important disadvantage — it allows users to write goals that unify values of different types (for example, natural numbers and linked lists). These sorts of unification should fail, and it sounds reasonable in a typed language to forbid such unifications at compile time. In general, it means that each logic variable has to be created with some known logic type to which it can be reified and with terms of which it can be unified. To handle such a case, we decided to use parameterized types that are represented by Kotlin generics.

Actually, we have three possible types of unifications:

- unification of two logic values, both of the same type;
- unification of two logic variables, both over the same type;
- unification of a logic value and a logic variable of the same type.

These observations motivate us to implement unification function `unify` with the following signature.

```
fun <T> unify(first: Term<T>, second: Term<T>) = ...
```

But this approach has a drawback — it allows the creation of logic variables over non-logic types because the parameter does not have any restrictions. The key solution to handle it is making the parameter a logic type too. It leads to the following self-recursive declarations.

```
interface Term<T : Term<T>> { ... }
class Var<T : Term<T>> : Term<T> { ... }
```

Having these declarations, we introduce type-checking at compile time for unifications, but another problem arises — implementing a mechanism of unifications for arbitrary logic types. A unification requires traversing all fields of a logic value that need to be unified with an another logic value. It can be done using Java reflection, but such an approach has significant shortcomings. Firstly, using reflection leads to a substantial decrease in performance. Secondly, with reflection, we lose flexibility because we cannot allow a user to specify what fields should be unified and what should not. We came up with another solution. An abstract implementation of unification is provided in the base class for logic terms. It walks both logic terms and unifies all values that are defined for current logic term by a user (Listing 1). As a result, to define a new logic term a user should inherit from the `CustomTerm` and implement two abstract properties — `subtreesToUnify` and (optionally) `subtreesToWalk`.

Listing 1. Declaration of user-defined type CustomTerm in kLogic (a sketch)

```
sealed interface Term<T : Term<T>> {
    fun unify(
            other: Term<T>,
            unificationState: UnificationState
    ): UnificationState? {
        val walkedThis = walk(unificationState.substitution)
        val walkedOther =
            other.walk(unificationState.substitution)

        return walkedThis.unifyImpl(walkedOther, unificationState)
    }

    fun unifyImpl(walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState?
}

@JvmInline
value class Var<T : Term<T>>(val index: Int) : Term<T> {
    override fun unifyImpl(
        walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState? {
        ...
    }
}

interface CustomTerm<T : CustomTerm<T>> : Term<T> {
    val subtreesToUnify: Array<*>

    val subtreesToWalk: Array<*>
        get() = subtreesToUnify

    override fun unifyImpl(
        walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState? {
        ...
    }
}
```

3.2 Variables construction

Each logic variable is identified by its unique integer identifier, so to be able to create fresh variables we need to maintain a mechanism that creates new unique identifiers. We implemented it quite simply by storing an integer index of the last created fresh variable and issuing the incremented value of this index for a new fresh variable.

In addition to the variable index, among unifications, we need to maintain a set of current added constraints (for now we have only disequality constraints) and substitution of created logic variables to other logic terms (values or variables). We reused the standard concept of a *state* (Fig. 2) — an immutable union of the constraints and the substitution (represented by a persistent map from logic variables to logic terms), which changes by unifications.

Listing 2. Definition of the state in KLOGIC

```
data class State(
    val substitution: Substitution,
    val constraints: PersistentSet<Constraint<*>> =
        persistentHashSetOf(),
)
```

Speaking about creating fresh variables, there is an important detail in implementation. The creation introduces a *lazy* goal, i.e. inserts a *delay*, the user-accessible interface should encourage users to create with a single delay a few fresh variables at once. So, we need to have many different functions for creating different numbers of fresh variables. In some programming languages (most of those are descendants of C programming language, for example), we can automatically generate such functions at the compile-time using *macro* mechanism. For some reason, KOTLIN in particular, and JAVA in general, do not have macros, although some of their features may be replaced by processing of annotations. Moreover, the number of method's parameters in JAVA is limited to 255[3], and creating more than 255 fresh variables at once is impossible. So, in theory different functions for creating different numbers of fresh variables could be generated somehow using Java annotations. For now, we have a few predefined implementations (from 1 to 8 fresh variables, to be exact), and functions to create more than 8 fresh variables at once has to be implemented manually by a user.

The absence of macro mechanism in the language affects the way we write relational programs. All MINIKANREN primitives should be defined as functions, which are evaluated in call-by-value strategy. In some cases, it requires an explicit inverse-eta-delay insertion before a recursive call of relation. KLOGIC and OCANREN have primitives for that, but in SCHEME it is not needed because `conde` and `fresh` primitives are implemented using a macro. We modified the code of our benchmarks to make execution traces the same as in SCHEME. As a result, sometimes relation definitions are not an idiomatic KLOGIC/OCANREN code.

There is an important detail in implementing and using functions for creating fresh variables connected with KOTLIN type system. Since that language was not designed to have type inference, and the usage of overloaded functions is widespread, we are obliged to manually specify types of created fresh variables when using `fresh` — consider an example in Fig. 2. On the same listing, there is a subtlety about unification of empty logic lists. The first `` `===` `` unifies a logic variable with an empty logic list. A swap of these two arguments is allowed in SCHEME and OCANREN but not in KOTLIN: type inference can't guess the type of elements of an empty logic list using the

[3] https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-4.html#jvms-4.3.3 (access date: 2023-06-06)

```
fun <T : Term<T>>
    appendo(x: ListTerm<T>, y: ListTerm<T>, xy: ListTerm<T>): Goal
  =
    ((x `===` nilLogicList()) `&&&` (y `===` xy))
  `|||`
    freshTypedVars<T, LogicList<T>, LogicList<T>>
        { head, tail, rest ->
            (x `===` head + tail) `&&&`
            (xy `===` head + rest) `&&&`
            appendo(tail, y, rest)
        }
```

Fig. 2. Appendo in KLOGIC. Infix binary operators for conjunction, disjunction, and unification should be written in backticks. Overloaded primitive for creation of fresh variables (almost always) requires types specification.

type of a logic variable. In the benchmark implementations, we adapted many unifications to make unification traces of SCHEME, OCANREN and KLOGIC exactly the same.

4 KLOGIC VS. OCANREN

In this section, we discuss programming with KLOGIC from OCANREN programmer's point of view.

The first thing that catches the eye: type annotations are everywhere. Sometimes we can omit type annotations for fresh variables, for example, when a variable is not used, but these cases are rare. This aspect of KLOGIC is not a particular implementation decision, but rather an artifact of KOTLIN. The host language is not designed to have a powerful type inference, and in the presence of overloaded functions (for example, the infix function + from Listing 2 which is a relational version of `cons`) the compiler's ability to infer types is limited.

The `freshTypedVars` primitive could be annotated by types in two ways: either in angle brackets or without angle brackets but near every introduced variable. For example, you could create fresh logic variables either via `freshTypedVars<LogicInt, LogiInt> { n, m -> ... }` or `freshTypedVars { n: Term<LogicInt>, m: Term<LogicInt> -> ... }`. With the first approach the types and the names of variables are textually separated and readability is reduced. In the second approach we need to specify explicitly, that we create a variable of a type of logic values (just `LogicInt` would be a ground integer), and type annotations become longer. Sadly, the ex-OCANREN developer should decide between two suboptimal approaches. We speculate that ex-SCHEME developer would feel even more frustration.

The proposed encoding of user-defined types is a decent step forward, in comparison to another KOTLIN implementation[4] we found. The KOTLINKANREN implementation proposed to use universal representation for values in logic domain, and user data types should be converted to this representation and back, which could be error-prone and could raise issues related to user-defined types' representability. The KLOGIC approach is better, but in comparison to OCANREN we could wish that the approach would require less boilerplate code. Ideally, the method `subtreesToUnify` should be generated using some macro/annotation mechanism to protect developers from mistakes.

[4] https://github.com/neilgall/KotlinKanren (access date: 2023-06-06)

OCanren doesn't allow us to skip non-logic subvalues of a logic value during unification currently, and klogic is theoretically better in this approach. Sadly, we don't have a good example to demonstrate the usefulness of this feature.

Klogic currently lacks a few optimizations present in OCanren and faster-miniKanren. The implementation of disequality constraints is very straightforward. Also, we don't optimize the unification with recently created fresh variables (also known as *set-var-val* optimization in faster-miniKanren).

5 BENCHMARKS

We compare faster-miniKanren in Racket with OCanren in OCaml and klogic in Kotlin in a few performance tests involving relational arithmetic [5] and Scheme relational interpreter [3]. We picked the same relations to benchmark as the paper [6] did. All benchmarking was performed on the desktop machine Intel© Core™ i7-4790K (16Gb RAM). Some software was taken from the official Ubuntu 23.04 x86_64 repository: Racket 8.7 (compiled to native code using Chez backend) and OpenJDK 17.0.7 (as the last Java LTS version). OCaml compiler 4.14+flambda was installed using OPAM[5] package manager. Running benchmarks was implemented using language-specific libraries: benchmark[6] for OCaml, benchmark[7] for Racket and JMH[8] for Kotlin.

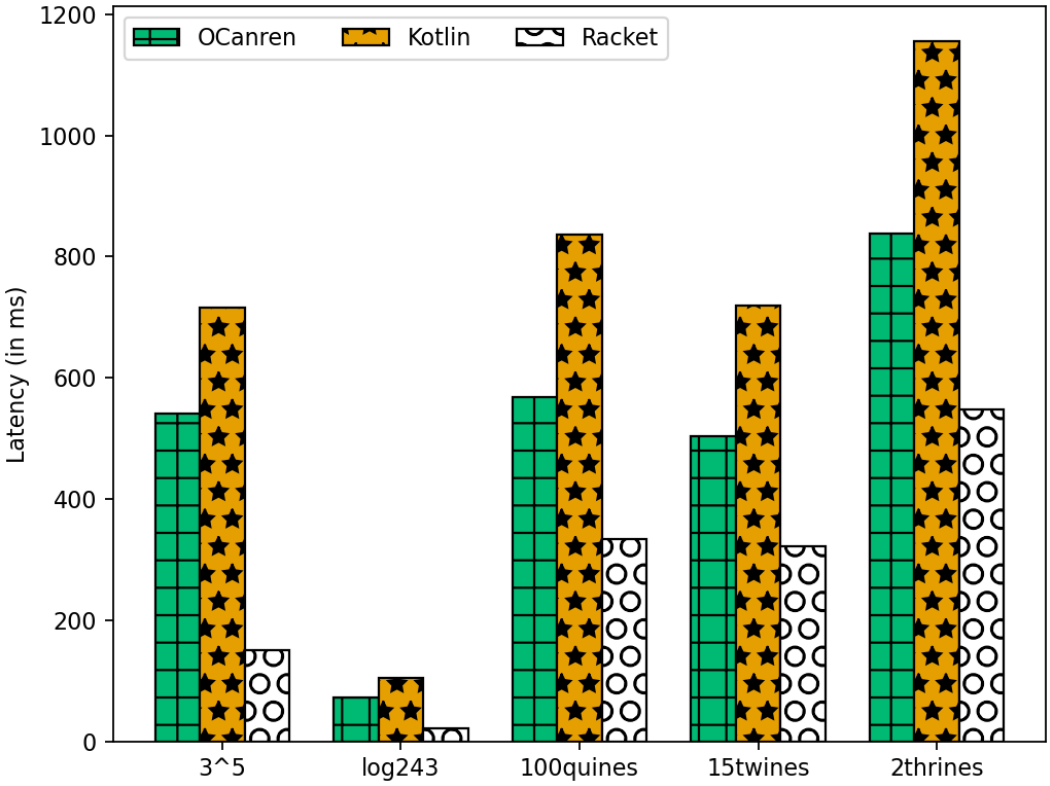
In the accompanying repository[9] we can find unification counts and unification traces for relations being benchmarked. It also has a submodule for a modified klogic implementation where traces of unifications can be toggled on/off using an environment variable. Unfortunately, this check is not optimized out by just-in-time compilation in JVM, and we need to comment it out manually. That's why the benchmarks implementation in klogic lives in another repository[10].

The table 3 demonstrates that if we worry only about performance, the faster-miniKanren implementation should be recommended instead the other implementations. Today klogic has a naive implementation of disequality constraints, which decreases performance of relational interpreter. However, disequality constraints are not involved to the Oleg numbers benchmarks. We can't explain why klogic unperformed, but the set-var-val optimization from faster-miniKanren should be applicable there. In the draft of the paper, we presented more auspicious benchmarks of klogic but the search order had not been the same between three implementations. The big changes of numbers demonstrates that evaluation of the performance as number of unifications per second could be misleading. The context switching of relational streams may seriously affect performance (it has been observed already [7]).

6 CONCLUSIONS AND FUTURE WORK

We presented klogic — a library for relational programming in Kotlin. It is designed to have a typed representation of logic values, which prevents developers from a certain class of mistakes. The disequality constraints are available, but other domain-specific constraints aren't (we believe that *absento* is not required for our representation of logic values). The implementation is rather straightforward in a few places. For example, optimizations from faster-miniKanren (lazy disequality constraints and storage for variable bindings inside variables) are not yet implemented. At the moment performance is slightly worse comparatively to OCanren, and we speculate that missing optimizations may change that.

[5] https://opam.ocaml.org/ (access date: 2023-08-22)
[6] https://github.com/Chris00/ocaml-benchmark (access date: 2023-08-22)
[7] https://github.com/stamourv/racket-benchmark (access date: 2023-08-22)
[8] https://github.com/openjdk/jmh (access date: 2023-08-22)
[9] https://github.com/Kakadu/miniKanren_exec_order/tree/tracing (access date: 2023-08-22)
[10] https://github.com/Kakadu/klogic/tree/tracing (access date: 2023-08-22)



	Unification count	OCanren (in ms)	Kotlin (in ms)	Racket (in ms)
3^5	433854	541.80	715.06	150.63
$log_3 243$	56277	72.48	105.08	22.24
100 quines	150732	567.75	836.67	334.36
15 twines	148525	502.90	719.67	321.27
2 thrines	224658	838.77	1155.47	548.38

Fig. 3. Performance results for various relations (in OCanren, Kotlin and Racket) involving Oleg numbers [5] (time of first answers of exponentiation 3^5 and inverse) and Scheme relational interpreter [3] (100 first quines, 15 twines and 2 thrines). The search order in the implementations is the same. On the Y axis we have latency in milliseconds (less is better).

The process of making execution traces for tree implementations exactly the same took much more time than we initially expected. In future, it would be great to have an automatic translator of relational programs between klogic, OCanren, and Racket, because the manual comparison of implementations' code is cumbersome.

REFERENCES

[1] Nada Amin, William E. Byrd, and Tiark Rompf. 2019. Lightweight Functional Logic Meta-Programming. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 225–243.

[2] TIOBE Software BV. 2022. *TIOBE Index*. TIOBE Software BV. Retrieved June 9, 2023 from https://www.tiobe.com/tiobe-index/

[3] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional*

Programming (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

[4] Xavier Clerc. 2013. OCaml-Java: OCaml on the JVM. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–181.

[5] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). In *Functional and Logic Programming*, Jacques Garrigue and Manuel V. Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 64–80.

[6] Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (2016), 1–22. https://doi.org/10.4204/EPTCS.285.1

[7] Dmitry Rozplokhas and Dmitry Boulytchev. 2022. Scheduling Complexity of Interleaving Search. In *Functional and Logic Programming*, Michael Hanus and Atsushi Igarashi (Eds.). Springer International Publishing, Cham, 152–170.