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.