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 in pure miniKanren without modifying the underlying implementation. 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 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, some complex constraints remain difficult to express.

In this paper, we propose using miniKanren itself as a language for constraint authoring. As we demonstrate, using only the core operators of miniKanren, it is easy to express a wide range of common constraints, as well as a number of novel constraints that are difficult to express in non-relational host languages. Moreover, we show that such a constraint language interoperates well with host language constraint frameworks that are better suited for expressing constraints that cannot be expressed in pure miniKanren, such as numeric constraints. *The key idea of this paper is that constraint solving in miniKanren can be viewed as a natural extension of the normal miniKanren search procedure, and can therefore be implemented as a modified miniKanren interpreter in which constraints are represented simply as normal miniKanren goals.*

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 discusses related work.

2 INTERFACE

In this section, we introduce three new forms and implement several constraints from previous work to illustrate their use. **`constraint`** (2.1) converts miniKanren goals into constraints. **`pconstraint`** (2.2) defines new primitive constraints besides ==. Finally, **`noto`** (2.3) negates miniKanren goals.

2.1 `constraint`: miniKanren Goals as Constraints

The **`constraint`** form wraps arbitrary miniKanren goals and redefines their semantics. Normally, **`conde`** generates multiple search branches and conjoins one child disjunct to each branch. Wrapped with **`constraint`**, however, it instead generates a disjunction constraint and conjoins it with the state corresponding to the current branch. Should each disjunct fail, the branch fails, as in the following examples.

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

```
(define (booleano v)
```

```
  (constraint
   (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.

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 `listo`. `listo` checks that a term unifies with a proper list [11]. 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)
  (constraint
   (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.[1]

Importantly, here and for the rest of the paper, when we write **`fresh`**, we in fact refer to a pattern matching form, `matcho`, that will be described in Section 3.5. `matcho` has proven easier to work with for the purposes of implementing this constraint system. It is still possible to define **`fresh`** appropriately for use in constraints, and so we use it for greater familiarity in the code examples, but we will not cover its implementation in detail in this paper.

2.1.3 `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.

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

[1] 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, it reifies as a suspended form of the waiting constraint. We are currently exploring modifications to the reifier that may resolve this issue.

`presento` is much more difficult to implement than `absento` using existing constraint frameworks due to the way in which it is fundamentally disjunctive. Because the constraint store implicitly conjoins all contained constraints, `absento` can insert its child constraints independently into the store. `presento`, by contrast, must guarantee that, for instance, the child constraint on a list's head must not fail—even if it otherwise would—if the constraint on the tail succeeds. This dependency between the child constraints requires additional bookkeeping that complicates the architecture of the constraint and the store. The complexity is further increased if the constrained value is a complex list term containing free variables, as the constraint may in that instance need to handle the tree traversal logic within the context of a complex unification logic that it may not be possible to resolve immediately. In the present framework, however, both `presento` and `absento` can be expressed with roughly the same order of implementation complexity.

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.

```
(define (typeo v t?)
  (if (var? v) (pconstraint (list v) type-check t?) (if (t? v) succeed fail)))

(define (type-check var val constraint t?) ...))

(define (symbolo v) (typeo v symbol?))
(define (numbero v) (typeo v number?))
(define (pairo) (typeo v pair?))
```

`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`.

The type checking function, `type`, at present requires some knowledge of the internal representations used by the solver to implement. 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 constraint goal used to

check constraint-constraint interactions. `constriant` is another primitive constraint bound to var, such as another type constriant. The auxiliary value of primitive constraints can be used to check their interactions, such as failing when two incompatible type constraints are bound to the same variable.

2.3 `noto`: Negating Goals and Constraints

Negation has been explored from a variety of angles in past work on miniKanren [14, 19]. In this implementation, **`noto`** generalizes the usual case analysis used to perform disequality checking. It runs its subgoal, and negates the result. If the subgoal succeeds, **`noto`** fails. If the subgoal fails, it succeeds. If the subgoal returns any other constraint, that constraint is negated and placed into the store. This scheme allows for the expression of a number of constraints that depend on negation, beginning with `=/=`.

2.3.1 `=/=`. Because **`noto`** generalizes disequality solving, expressing disequality is trivial.

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

2.3.2 `not-symbolo`, `not-numbero`. **`noto`** generalizes in the same fashion to other primitive constraints besides `==`.

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

2.3.3 `not-booleano`, `not-listo`. Complex constraints built with conjunction, disjunction, and fresh work also work as expected.

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

2.3.4 `absento`. Using disequalities and negated 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 also possible to implement `absento` as a negation of `presento`, or vice versa.

3 IMPLEMENTATION

The implementation as a whole is composed of a pair of miniKanren interpreters. The first—the "stream" interpreter—interprets **`conde`** and **`fresh`** as stream constructors that generate the interleaving search tree. All other goals are viewed as constraints and are passed to the "constraint" interpreter to check for unsatisfiability within a given branch. As the constraint solver is a miniKanren interpreter, the constraints themselves are normal miniKanren goals, implemented here as first order structures. The constraint interpreter defines `==`, **`constraint`**, **`pconstraint`**, **`noto`**, **`succeed`**, and **`fail`**. It also redefines **`conde`** and **`fresh`** for the constraint solving search context.

The constraint interpreter performs a depth-first miniKanren search bounded by the rule that **fresh** goals must suspend when the variables on which they depend are free.[2] Because constraints within this framework may contain **conde**, a given miniKanren goal, viewed as a constraint, may imply a disjunction between any number of conjunctions of simpler constraints. The goal of the constraint interpreter's search is to find one such subset of mutually satisfiable primitive constraints entailed by a single constraint store much in the same way the stream interpreter must search for one subset of mutually satisfiable constraints entailed by the program overall.

In the next several sections, we review the implementation of the constraint solver. Code examples have been simplified for greater readability. The implementation is open source, and the source code should be consulted for more detail.

3.1 Conjunction

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

```
(define (solve-constraint g s ctn resolve delta)
  (cond
   [(succeed? g) (if ... (solve-constraint ctn s succeed resolve delta))]
   [(conj? g) (solve-constraint (conj-lhs g) s (conj (conj-rhs g) ctn) resolve
    delta)]
   ...)))
```

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

`g` and `s` are self-explanatory. `ctn` is so named due to a structural analogy with continuations and continuation-passing style. 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 for the current goal `g`, 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—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.

3.2 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 delta)
2   (let-values ([(bindings recheck s) (unify s (==-lhs g) (==-rhs g))])
3     (if (fail? bindings) (values fail failure)
4         (solve-constraint succeed s ctn (conj recheck resolve) (conj delta
      bindings)))))
```

[2] Recall that **fresh** in this case is implemented as a pattern matching form that possesses explicit references to the variables on which it depends.

This definition of unification will look familiar from its standard implementation elsewhere. The unifier is called, the resulting state is checked for failure. 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`.[3] `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 if so returns the failure signature—the trivial **`fail`** goal and the **`failure`** stream. The **`failure`** stream corresponds to the failure mode of the input state `s`, and the **`fail`** goal likewise corresponds to the failure mode of the input parameter `delta`, which is a first order representation of the constraints that have been added to `s` during this execution of the constraint solver.

Consider line 4. The unification constraints representing the new bindings are conjoined to `delta` and passed to further solving. Should the current constraints ultimately prove satisfiable, the constraint solver will return `s` and `delta`, both of which contain the information about which bindings were made at this stage in the solver. `delta` can be viewed as an extension of the representation of the state that tracks changes made during solving. It is primarily useful during negation and disjunction, as the current state representation is difficult to negate or disjoin. A constraint without negation or disjunction will ultimately discard `delta` and simply return `s` as the product of solving.

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 stream interpreter, 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, `delta` 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 3.3.

3.3 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. Conjunctions and disjunctions are negated using De Morgan's laws in the usual way. Consider the following listing:

```
1 (define (solve-noto g s ctn resolve delta)
2   (let-values ([(d s2) (solve-constraint (noto g) s succeed succeed succeed)])
3               (solve-constraint succeed (store-constraint s (noto d)) ctn resolve
      (conj delta (noto d)))))
```

[3] This is 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.

The negation of `g`, (**noto** `g`), is solved recursively on line 2 and then negated before being returned to the store and simultaneously to the `delta` constraint on line 3. Note that the initial call to `solve-constraint` is invoked with `ctn`, `resolve`, and `delta` 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 the returned delta, `d`, containing only changes made by the positive version of goal `g` and not by the right-hand conjuncts of the original goal. As a result, `d` can simply be negated and returned to the store before further solving. Had $y = 1$ been passed as `ctn`, for example, it would have returned conjoined to `d`, and subsequently negated to $y \neq 1$ before being returned to the store, which is not correct.

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 delta)
2    (cond
3     [(succeed? g)
4      (if (succeed? ctn)
5          (if (succeed? resolve)
6              (values delta s)
7              (let-values ([(d s) (solve-constraint resolve s succeed succeed
       delta)])
8                (if (fail? d) (values fail failure)
9                    (values delta s))))
10         (solve-constraint ctn s succeed resolve delta))]
11     ...)))
```

When `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. However, if `resolve` is solved, only the original `delta` is returned on line 9, not the subsequently solved `d`. This change ensures that during negation solving, constraints removed from the store do not pollute the returned delta and become incorrectly negated.[4] Finally, once all future constraints have been exhausted, the delta values are returned along with the state on line 6.

3.4 Disjunction

Consider the following listing:

```
1  (define (solve-disj g s ctn resolve delta)
2    (let-values ([(d-lhs s-lhs) (solve-constraint (disj-lhs g) s succeed succeed
       succeed)])
3      (cond
4       [(fail? d-lhs) (solve-constraint (disj-rhs g) s ctn resolve delta)]
5       [(succeed? d-lhs) (solve-constraint succeed s ctn resolve delta)]
6       [else (let-values ([(d-rhs s-rhs) (solve-constraint (disj-rhs g) s succeed
       succeed succeed)])
7           (if (fail? d-rhs)
8               (solve-constraint succeed s-lhs ctn resolve (conj delta d-lhs))
9               (solve-constraint succeed (store-constraint s (disj d-lhs d-rhs)) ctn
       resolve (conj delta (disj d-lhs d-rhs)))))])))
```

[4]Note that this procedure necessarily throws away the work done to solve rechecked constraints. We are currently experimenting with alternative designs that retain more of that work.

`solve-disjunction` first solves the left-hand disjunct on line 2. Like the negation solver, `ctn`, `resolve`, and `delta` are all **`succeed`**, which ensures that the returned constraints reflect only simplifications of constraints contained within the disjunct.

If the left-hand disjunct fails, the solver simply solves the right-hand disjunct on line 4. If it succeeds, the rest of the disjuncts can be skipped. Otherwise, the right-hand side is solved on line 6 and it is disjoined with the left-hand side and returned to the store on line 9. If the right-hand side fails, the results of the left-hand side are returned to the store, reusing the state produced by solving the left-hand side as an optimization on line 8.

Stepping back, the disjunction constraint finally makes clear what it means to view constraint solving as search in this instance. Each disjunction must search among its child disjuncts for at least one that does not fail in the current state. When the state moves down the right or left-hand branches of the disjunction constraint, it accumulates one child disjunct. When it passes through all conjoined disjunction constraint contained in `ctn` or `resolve`, it will have ensured that there is at least one subset of disjuncts that are mutually satisfiable in the current state. Failure to find such a subset proves the unsatisfiability of the store, and the branch fails.

Unsatisfiability is relatively easy to detect as it only requires finding one non-failing disjunct in each disjunction. Ensuring that unifications entailed by the constraint store are added directly to the substitution, such as when `booleano` is conjoined with $x \neq \top$ and therefore unifies $x = \bot$, requires that additional disjuncts be checked. The simplified implementation above naively checks all disjuncts, but work is ongoing to investigate possible benefits of laziness in the disjunction solver.

3.5 Matcho

In most cases in the current implementation, the pattern matching form `matcho` is used in place of **`fresh`**. `matcho` destructures tree terms and binds their elements to variables in a new lexical scope as in the following example:

```
(matcho ([xxs (x . xs)]) ...)
```

This form destructures `xxs` and binds its head and tail to `x` and `xs`, respectively, before processing child goals. The internal representation of a `matcho` goal consists of a list of free variables on which it depends, a list of bound values, and a closure for processing the final patterns. Solving simply involves looking up the free variables, adding them to the bound variable list as they become bound, and suspending in the constraint store on encountering a variable that is still free in the current substitution. This procedure guarantees that constraints will only run until they exhaust the bound values in the substitution, preserving the completeness of the search.

`fresh` can also be used in constraints, although it is more difficult to optimize. For that reason it is usually preferred to write constraints with the pattern matching form. Opaque **`fresh`** goals must expand until they yield a disjunction containing at least one non-failing disjunct. Whereas the stream interpreter would create two branches on such a disjunction, the constraint interpreter suspends the computation in the constraint store.

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, 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 the simple solver above, it is possible to attribute disjunctions to all of their child goals' variables. However, lazy implementations can get away with fewer. Work on this subject remains ongoing.

Once the attributed variables have been determined, the current implementation copies pointers to the constraint to each variable index in the store. Constraints are stored separately to avoid stale constraints proliferating in the store.

4 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] to the best of our knowledge.

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 offers a practical implementation of negation for constraint authoring that would be interesting to compare with more complex forms of negation studied in previous work.

5 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. Much work remains to be done on the constraint system itself, from further studying the effects of laziness to exploring integrations with solvers that require specialized representations. 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 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 this 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 this 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.

6 ACKNOWLEDGMENTS

We thank Will Byrd for discussions of early versions of this idea, Evgenii Moiseenko for clarifying some points of previous work, and Greg Rosenblatt for identifying an important edge case. 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.