A Framework for Extending microKanren with Constraints

Jason Hemann Daniel P. Friedman
Indiana University
Indiana, USA
{jhemann,dfried}@indiana.edu

We present a framework for building CLP languages with symbolic constraints based on microKanren, a domain-specific logic language shallowly embedded in Racket. A language designer provides the names and violation conditions of atomic constraints. We rely on Racket's macro system to generate a black-box constraint solver and other components of the microKanren embedding. The framework itself and the implementation of common Kanren constraints amounts to just over 100 lines of code. Our framework is both a teachable implementation for constraint logic programming as well as a test-bed and prototyping tool for constraint systems.

1 Introduction

Constraint logic programming (CLP) is both an extension and a generalization of traditional logic programming – it provides a way to extend logic programming languages with new constraints and simultaneously situates pure logic programming as an instance of CLP, in which unification is itself the solver. Constraint logic programming has proven itself a highly declarative programming paradigm applicable to a broad class of problems [18]. Jaffar and Lassez's CLP scheme [16] generalizes the model of logic programming to include constraints over particular problem *domains* (i.e. structures), and the scheme explicitly separates constraint satisfiability from the inference, control, and variable management.

Since at least the development of this theoretical foundation, implementers have engendered many CLP languages of a wide variety. Traditionally these languages are extensions of Prolog but this is by no means the only choice. Such languages' implementations often closely couple inference and constraint satisfaction to leverage domain-specific knowledge and improve performance. This specialization and tight coupling could force an aspiring constraint implementer to rewrite a large part of the system to integrate their additions. LP metainterpreters, another common approach to implementing CLP systems, are unsatisfying in different ways. The user pays some price for the interpretive overhead, even if we use techniques designed to mitigate the cost [25]. Moreover, implementing intricate constraint solvers in a logic language can be unpleasant, and performance concerns may pressure implementers to sacrifice purity.

We encountered still different issue in extending Kanrens with constraints. miniKanren [12] is a pure logic language implemented as a purely functional, shallow embedding in its host language, here Racket [8]. microKanren is our approach to demystifying some of the complexities of miniKanren implementations. We sought to build a short and clear LP language implementation readable by functional programmers. We separate inference and unification from the surface syntax, allowing functional programmers in call-by-value languages to implement the core logic programming features without the syntactic sugar. The reference implementation is just over

Submitted to:
WFLP 2016

50 lines long, and since its initial release others have built upwards of 50 implementations in more than 25 host languages (see `miniKanren.org`.) There are small syntactic differences between the miniKanren and microKanren languages and their various implementations. We elide these details and describe them collectively as "Kanrens" unless otherwise important. In either case a Kanren embedding gives functional programmers access to logic programming in languages that do not natively support it.

As originally implemented, microKanren's only constraint is syntactic equality. Many of the most interesting typical uses of Kanrens require *symbolic constraints* [6] beyond equality. The most commonly used Kanren language augmented with symbolic constraints implementation balloons to upwards of a thousand lines of code, and seem somewhat baroque compared to the 50 line microKanren. Some of this added heft comes from the constraint solving, and the remainder is simplification and answer projection. Adding each new constraint requires utilizing domain knowledge on an ad-hoc basis, and complicates constraint solving, simplification, and answer projection. This situation calls out for a different design and decomposition of the problem.

We suggest a framework for building microKanren-like languages with symbolic constraints. Our framework uses Racket's macro system to generate shallow functional embeddings of logic languages augmented with constraints. The language designer provides the names of the atomic constraints—Racket identifiers—and the conditions for constraint violations in the form of predicates. For us, the definition of a constraint system is the set of constraint interactions that cause failure. Our macros generate a black-box constraint solver, as well as other components of the embedding. The languages generated by our framework return the bag of constraints in an unsolved form. We intend to extend the framework to also simplify constraints to a canonical form, eliminate redundancies, and project answers with respect to initial query variables. In the extension we envision, the language designer describes rewriting-rules for constraint sets, and the framework does the rest. We envision using our framework as a tool for rapidly prototyping constraints and CLP languages, and also as an educational artifact for functional programmers.

We describe the Kanren term language, the domain of our symbolic constraints, in Section 3. We use this framework to implement common miniKanren constraints in Section 6, as well as suggestive new ones in Section 7. Our untyped shallow embedding gives the language designer a fair amount of flexibility, and we describe future directions and alternate design choices in Section 9. This paper is a literate document; it contains the full implementation of our framework in Section 5, and we provide the inference engine in 22 lines of code as an Appendix. The complete framework and the implementation of common miniKanren symbolic constraints comprise just over 100 lines – a marked decrease in line count over similarly featureful implementations. We also provide our full implementation at `github.com/jasonhemann/constraint-microKanren` alongside several syntactic extensions. To recapitulate, our contributions include:

- A macro-based framework for generating pure, functional, shallowly-embedded CLP languages in Racket
- The implementations of several symbolic constraints common to Kanren
- The implementations of several more suggestive and useful symbolic constraints
- A literate presentation of the complete framework and the implementation of our constraints.

2 Background

We do not expect any miniKanren background or experience of the reader; a logic programming background is enough. We briefly adumbrate here the necessary background material. We review some features and contrast miniKanren's syntax and behavior to that of Prolog. The interested reader should consult one of several references and tutorials for further details [10].

Although Kumar [24] has formally specified miniKanren's syntax and given several semantics, implementers have been encumbered by neither, and none of the more widely used implementation obey these specifications. Instead, miniKanren is better described as a family of related logic programming languages, traditionally shallowly embedded in a declarative host language, most of whose semantics are informally specified by direct appeal to their host languages' features. Our embedding inherits much of the syntax and structure of it's host. Implementations' concrete syntax varies from host to host because of the shallow embedding. Different miniKanren implementations often provide different extensions and operators, as happens with Prologs. The traditional Prolog definition of a naïve reverse (`nrev`) is syntactically analogous to the miniKanren version defined using pattern-matching syntax [20].

```
nrev([],[]).
nrev([H|T],L2) :- nrev(T,R), append(R,[H],L2).
```

```
(defmatche (nrev l₁ l₂)
  ((() ()))
  (((,h . ,t) ,l₂)
   (fresh (r)
     (nrev t r) (append r `(,h) l₂))))
```

Rather than using case to distinguish constants and variables, in miniKanren symbol in quoted (`'`) data are constants and are otherwise considered variables. In pattern matches, we precede variables by an unquote (`,`), and terms are otherwise considered constants. Pairs are destructured as `(,`α `. ,`β`)` rather than `[`α`|`β`]`. Predicates are defined at once as a collection of clauses, and we do not require the name of the predicate in every clause. Unlike in Prolog, the programmer must explicitly introduce auxiliary variables using the `fresh` operator.

Below is the translation of the `nrev` into microKanren.

```
(define-relation (nrev l₁ l₂)
  (disj (conj (== l₁ '())
              (== l₂ '()))
        (call/fresh (λ (h)
                      (call/fresh (λ (t)
                                    (conj (== `(,h . ,t) l₁)
                                          (call/fresh (λ (r)
                                                        (conj (nrev t r)
                                                              (append r `(,h) l₂))))))))))))
```

The operators `conj` and `disj` provide binary conjunction and disjunction; `call/fresh` introduces scope, relying on Racket's `λ` for lexical binding. The operator `==` is microKanren's first-order syntactic equality constraint. Finally, the operator `define-relation` defines predicates and plays a part in the interleaving. Syntactically, the language resembles Spivey and Seres's Haskell embedding of Prolog [28]. Modulo differences in syntax, both are essentially a *completed predicate*, à la Clark [5].

We can layer the miniKanren syntactic sugar over this somewhat verbose core syntax with about 45 lines of Racket macros. The microKanren approach is now the most common way to implement miniKanren.

The miniKanren equivalent of the `?-` prompt is the `run` operator. The `run` operator takes a number (here `1`), a variable name (here `q`), and a query to execute. The `run` operator returns a list of at most that many answers to the query, simplified and answer projected [19] with respect to the query variable. The following queries return similar answers.

```
?- findnsols(1,L2,nrev([a,b,c],L2),Q).
Q = [[c,b,a]] ?
```

```
> (run 1 (q) (nrev '(a b c) q))
'((c b a))
```

Prologs default to DFS, whereas Kanrens rely on an unguided, interleaving depth-first search, based on Kiselyov et. al's Logic monad [23], that is both complete and more useful in practice than are BFS or IDDFS. This style of search is not currently available in any logic language of which we are aware besides miniKanren. This complete, more efficient search makes a purely relational approach to programming more practical, as we can query pure, all-modes relations more effectively. This in turn reduces the need for both non-logical operators and predefined predicates.

For instance, Kanrens have no equivalent to Prolog's `is` operator. Instead, the Kanren arithmetic system is built from all-moded relations built from bit-adders up, without any use of non-logical operators [22]. The `add` used in the following query is 3-place addition relation and `log` is a four-place logarithm relation. miniKanren prints the answers as little-endian binary numbers. In the first, we return two answers, and indeed $0+0=0$ and $1+1=2$. In the second, we return a list of the only answer `(#f #t #t)`, as $14=2^3+6$.

```
> (run 2 (a b) (add a a b))
'((() ()) ((#t) (#f #t)))
> (run* (q) (log (#f #t #t #t) (#f #t) (#t #t) q))
'((#f #t #t))
```

These relations run in all modes, and with our interleaving search appropriately terminate. This declarative arithmetic is efficient enough to be useful in practice [2]. Kanrens do not currently support arithmetic over real or floating-point numbers, and heavily numeric computations are not Kanrens' strong suit. Instead, this all-moded logic programming technique inspires non-traditional sorts of problems. Such miniKanren programming examples include a typechecker that also behaves as a type inhabiter [26], an automated theorem prover that doubles as a proof assistant [3], and a programming-language interpreter that also serves as a quine generator [4]. Several such examples are available in browser at `tca.github.io/veneer/examples/editor`. Consider as a representative example `eval`, a relational interpreter for a Racket-like language. The relation holds between an expression `e`, an environment ρ, and a value `v` when the value of `e` in ρ is `v`. To search for a quine, we query `eval` for an expression that evaluates to its listings (source code). The result is a valid Racket quine.

```
> (run 1 (q) (eval q '() q)
'((((λ (_.0) (list _.0 (list 'quote _.0))) '(λ (_.0) (list _.0 (list 'quote _.0))))
   (=/= ((_.0 closure)))
   (sym _.0)))
```

miniKanren prints fresh variables as `_.`n in projected answers. The above answer is subject to several constraints: `_.0` must be a symbol other than `closure`. Historically, miniKanren programmers develop new constraints in response to challenges that arise in the course of solving such problems.

3 Kanren terms and constraint domain

The Kanren term language contains symbols, logic variables, Booleans, the empty list (`()`), and `cons` pairs of the above. We may imagine the term language has a single, implicit, uninterpreted functor tag, `cons/2`. In the interest of simplicity we reserve non-negative integers as logic variables. Our syntactic equality constraints, built with `==` are simply equations over the Herbrand domain. The additional common Kanren constraints are (binary) term disequality, written `=/=`, (binary) subterm discontainment, written `absento`, and the unary domain constraints `symbolo`, and `not-pairo`[1]. These last two declare the constrained term a symbol or a non-pair respectively. We demand that atomic constraints be applicable over the entire term language. Similarly, we expect that for all constraints c and term sequences $t_1 \ldots t_n$, if $c(t_1 \ldots t_n)$ holds, then for any σ, if $u_1\sigma = t_1 \wedge \ldots \wedge u_n\sigma = t_n$, then $c(u_1 \ldots u_n)$ also holds. That is, our disequality constraints correspond to the behavior of `dif/2` rather than `\==`, and other constraints behave analogously. These requirements carry important properties of the symbolic constraints in which we are interested and simplify the grammars of the generated languages.

In addition to the usual benefits, our constraints allow us to compress what would be multiple answers (potentially infinitely many) into single finite representations. Consider for instance, the `absento` constraint. An `absento` constraint holds between two terms `x` and `y` when `x` is neither equal to, nor a subterm of `y`. With just `=/=` constraints, we can in general only express this relationship in the limit, e.g. an infinite conjunction of disequalities between the fresh variable `y` and all possible terms `x` from which it is absent. With the `absento` constraint we can represent this relationship finitely.

4 Constraint framework restrictions

Our framework builds embedded constraint logic languages. The language designer gives the constraint names (constraint relation symbols), and the conditions the violate constraints via predicates to test for invalid sets of constraints. From these, the framework will generate the microKanren (and thus miniKanren) constraint operators and a constraint solver automatically. To specify these interactions via predicates in some sense *is* to define the constraints themselves. They are a declarative, functional specification of what it means to violate these constraints. Constraint-violation predicates, qua predicates, are by definition total functions. As such, the solver for the constraint system (`invalid?`, defined in Section 5) is also total.

We provide `==`, representing syntactic first-order equality, with every constraint system, and implement it via unification with the `occurs?` check. We fix this particular equational theory because of our intended use cases for the generated languages (e.g. sound type inference in a simply-typed language).

We require the resultant constraint solver to be *well-behaved* [17]. This means it is *logical*—that is, it gives the same answer for any representation of the same constraint information (i.e., regardless of order, redundancy, etc). It is also *monotonic*—that is, for any set of constraints, if the solver deems the set invalid, adding additional constraints cannot produce a valid set. Therefore, when adding a new constraint-violation predicate, a language designer is not required to modify older ones. Such a redesign may, however, clarify these violations. Presently, we do

[1] Implementations also often include a numeric domain constraint `numbero`. We omit numeric constants and also this constraint.

not check any of the preceding restrictions and requirements.

5 Constraints framework implementation

In this section we describe the implementation of our framework. We model the constraint store with a persistent hash table. To add a new type of constraint to the embedded language is to create a field in the hash table. We use the constraint's name as the key for that constraint's field in the store. We construct the initial state with something akin to `make-initial-state` below:

```
(define-syntax-rule (make-initial-state cid ...)
  (define S₀ (make-immutable-hasheqv '((==) (cid) ...))))
```

The Racket primitive `define-syntax-rule` builds a macro. This macro transforms an occurrence of the pattern, an expression beginning with `make-initial-state` followed by zero or more identifiers into an instantiation of the macro's template. This template creates the definition of `S₀` as an immutable hash table with `==` and each of the provided constraint identifiers as keys associated with empty lists (`()`). Since constraint identifiers are unique, each field will have a distinct key. One can also view these different fields as distinct stores for each type of constraint [29]. We use something like `make-initial-state` to construct the initial state when we make a constraint system. The hash table is immutable to allow structure sharing across different extensions of the same state, and we rely on the host language's garbage collection to free memory.

```
> (define == (make-constraint-goal-constructor '==))
...
```

In the embedding, we define globally each atomic constraint as the result of invoking `make-constraint-goal-constructor`. We use the name of the atomic constraint as its key in the store. The function `make-constraint-goal-constructor` takes a field in the store and returns a function accepting the correct number of term arguments. This is the definition of an atomic constraint in our embedding.

```
(define (((make-constraint-goal-constructor key) . terms) S/c)
  (let ((S (ext-S (car S/c) key terms)))
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))
```

Invoking this constraint with terms is a goal: a function expecting a state and returning a stream of states. To evaluate a constraint we extend the state and test for consistency. If the extended constraint store is consistent, we return a stream of a single state; if not, we return the empty stream. Once added, constraints are not removed from the store. This decision means the size of the constraint store and the cost of checking constraints grows each time we encounter a constraint in the execution of a program. In Section 9 we suggest improvements.

To constrain a term(s) during the execution of a program is to add the constrained term(s) to the corresponding field of the store. The `ext-S` function takes the store, the key, and a list of terms. The `ext-S` function adds those terms, as a data structure, to a list of such structures. By consing all of the terms together, `hash-update` creates the data structure.

```
(define (ext-S S key terms) (hash-update S key ((curry cons) (apply list* terms))))
```

We check consistency with `invalid?`. `make-invalid?` builds the definition of `invalid?`. The language designer provides `make-invalid?` a list of the names of atomic constraints (Racket identifiers). The designer also provides a sequence of predicates that check for constraint violations. Each predicate takes a substitution and returns true if it detects a violation. The constraint identifiers are free variables of the predicates; the expansion of `make-invalid?` will bind them. The result of `make-invalid?` is a predicate that tests if a store is invalid.

```
(define-syntax-rule (make-invalid? (cid ...) p ...)
  (λ (S) (let ((cid (hash-ref S 'cid)) ...)
           (cond ((valid-== (hash-ref S '==)) => (λ (s) (or (p s) ...)))
                 (else #t)))))
```

The first constraint we check is `==`. If this constraint is consistent, the result is a substitution. Assuming this field is valid, we pass the resulting substitution as an argument to the constraint-violation predicates.

Because our framework includes the implementation of the constraint `==` and provides `==` in every generated constraint system. The `==` constraint is special because when testing for the violation of other constraints, we treat terms of the language as classes quotiented by their meaning under the substitution. The `valid-==` function below and its associated help functions are also included with the framework. The `valid-==` function expects a list of cons pairs of terms to unify with each other. We provide `unify`'s definition in the Appendix.

```
(define (valid-== ==)
  (foldr (λ (pr s) (and s (unify (car pr) (cdr pr) s))) '() ==))
```

We used the phrase "something akin to" when describing `make-initial-state`. This is the main syntactic form for building constraint systems. We build the entire constraint system and embedded language with one invocation of `make-constraint-system`. This new syntactic form takes the same parameters as does `make-invalid?`. It builds `invalid?`, the initial state, and all the constraints themselves. The result is a constraint system; together with microKanren's control infrastructure (see Appendix) this yields a full implementation of microKanren-like CLP language. To construct a microKanren with just equality, the language designer invokes `make-constraint-system` with an empty list of constraint identifiers and no constraint-violation predicates.

```
> (make-constraint-system ())
```

The definition below uses Racket's `syntax-parse` [7], a more sophisticated macro system. We pattern-match on the syntax argument, and the hash (`#`) begins the definition of the syntax template. We use `syntax-local-introduce` to introduce three new identifiers into lexical scope; the remaining constraint identifiers are already scoped.

```
(define-syntax (make-constraint-system stx)
  (syntax-parse stx
    [(_ (cid:id ...) p ...)
     (with-syntax ([invalid? (syntax-local-introduce #'invalid?)]
                   [S₀ (syntax-local-introduce #'S₀)]
                   [== (syntax-local-introduce #'==)])
       #'(begin (define invalid? (make-invalid? (cid ...) p ...))
                (define S₀ (make-immutable-hasheqv '((==) (cid) ...)))
                (define == (make-constraint-goal-constructor '==))
                (define cid (make-constraint-goal-constructor 'cid))
                ...))]))
```

This macro is the primary driver of our framework. The preceding code and the 22 line Appendix comprise the entire implementation.

6 Implementing a constraint system

Next, we make further the use of our framework. We implement a series of constraint-violation predicates with some associated help functions and use those predicates to generate a constraint system of common symbolic constraints. We develop these constraints and their predicates one at a time.

Beyond `==`, the typical Kanren contains four other constraints: `=/=`, `absento`, `symbolo`, and `not-pairo`. We discuss the predicates required to implement these constraints one at a time.

We first add a predicate to test for a violated `=/=` constraint. This predicate searches for an instance where, with respect to the current substitution, two terms under a `=/=` constraint already unify. In that case, the `=/=` constraint is deemed violated.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
    (λ (s) (ormap (λ (pr) (same-s? (car pr) (cdr pr) s)) =/=))
    ...)
```

We implement this predicate in terms of a help function `same-s?`. If the result of unifying two terms in the substitution is the same as the original substitution, then those terms were already equal relative to that substitution.

```
#| Term × Term × Subst → Bool |#
(define (same-s? u v s) (equal? (unify u v s) s))
```

The next predicate checks for violated `absento` constraints, using the auxiliary predicate `mem?`. The predicate searches for an instance where, with respect to the substitution, the first term of a pair already unifies with (a subterm of) the second term. In that case, we deem the `absento` constraint violated.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
    ...
    (λ (s) (ormap (λ (pr) (mem? (car pr) (cdr pr) s)) absento))
    ...)
```

The predicate `mem?` checks if a term `u` is already equivalent to any subterm of a term `v` under a substitution `s`. It makes use of `same-s?` in the check. If the result of unifying `u` and `v` is the same as the substitution `s` itself, then the two terms are equivalent.

```
#| Term × Term × Subst → Bool |#
(define (mem? u v s)
  (let ((v (walk v s)))
    (or (same-s? u v s) (and (pair? v) (or (mem? u (car v) s) (mem? u (cdr v) s))))))
```

We write a third constraint-violation predicate to search for a violated `symbolo` constraint. For each term under a `symbolo` constraint, we look if that term, relative to the substitution, is anything but a symbol or a variable. If so, that term violates the constraint. We define the function `walk` in the Appendix. The `not-pairo` violation predicate operates similarly. Their definitions complete our implementation of the common Kanren symbolic constraints.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
    ...
    (λ (s) (ormap (λ (y)
                    (let ((t (walk y s)))
                      (not (or (symbol? t) (var? t)))))
                  symbolo))
    (λ (s) (ormap (λ (n)
                    (let ((t (walk n s)))
                      (not (or (not (pair? t)) (var? t)))))
                  not-pairo)))
```

We show below the execution of an example microKanren program that uses all the typical Kanren constraints. The result of invoking this program is a stream containing a single state. We see that all the constraints are present in the constraint store, and we can read off each constraint. The `#hasheqv(...)` is the printed representation of the hash table, whose elements are the key/value pairs. For instance, the `=/=` field, `(=/= . ((c . 0) (0 . b)))`, contains the pairs `(c . 0)` and `(0 . b)`. These are the `=/=` constraints that have been added.

```
> (call/initial-state 1
    (call/fresh (λ (x)
                  (conj (== 'a x)
                        (conj (=/= x 'b)
                              (conj (absento 'b `(,x))
                                    (conj (not-pairo x)
                                          (conj (symbolo x)
                                                (=/= 'c x)))))))))
'((#hasheqv((== . ((a . 0))) (=/= . ((c . 0) (0 . b))) (absento . ((b 0)))
            (symbolo . (0)) (not-pairo . (0)))
   . 1))
```

7 Adding new constraints

In addition to clarifying existing implementations, our framework also simplifies describing more complicated symbolic constraints new to Kanren: `booleano` and `listo`. The first mandates that the constrained term be a Boolean, and the second a proper list. These constraints have more complex interactions than do the previous ones. As a result, we need several new predicates to support the implementation of each of these constraints.

We suggest these new constraints, both because of their additional complexity, and also their utility. With them, we can improve the implementations of relational interpreters, an archetypal miniKanren programming example. Consider the partially-completed miniKanren definition of the relational interpreter `eval` below.

```
(defmatche (eval e ρ v)
  ((,e ,ρ ,v) (fresh () (symbolo e) (lookup e ρ v)))
  ((,e ,ρ ,v) (fresh () (booleano e) (listo ρ)))
  ...)
```

If `e` is a variable, `v` is its value in the environment. We define `lookup` recursively as a three-place relation. When the variable is found in the environment, we return its value. In prior implementations of relational interpreters, the remainder of the environment remains unconstrained. Without the `listo` constraint, the only way to ensure our environments are proper

lists requires a recursive relation. This amounts to enumerating proper lists of all given lengths. Instead, we can now express infinitely many answers with a single `listo` constraint. We have also more tightly constrained the implementation of `lookup`, which results in more precise answers.

```
(defmatche (lookup x ρ o)
  ((,x ((,x . ,o) . ,d) o) (listo d))
  ((,x ((,aa . ,da) . ,d) o) (fresh () (=/= aa x) (lookup x d o))))
```

In prior definitions of `eval`, rather than using a `booleano` constraint, we equated the term first with `#t`, and then separately with `#f`. This generates near-duplicate programs that differ in their placement of `#t` and `#f`. By instead "compressing" the Booleans into one, we ensure the programs we generate have a more interesting variety.

7.1 Implementing `booleano`

Checking `booleano` involves more work than does checking the prior domain constraints, since there are precisely two Boolean values. The first predicate if we have forbid a term from being either of the constants `#t` and `#f` while demanding that it be a Boolean. We also need a predicate to check for a `booleano`-constrained term that is a non-variable, non-Boolean. Finally since the `booleano` domain constraint is incompatible with `symbolo`, the last predicate checks for terms constrained by both.

```
> (make-constraint-system (=/= absento symbolo not-pairo booleano)
    ...
    (let ((not-b (λ (s) (or (ormap (λ (pr) (same-s? (car pr) (cdr pr) s)) =/=)
                            (ormap (λ (pr) (mem? (car pr) (cdr pr) s)) absento)))))
      (λ (s) (ormap (λ (b) (let ((s₁ (unify b #t s)) (s₂ (unify b #t s)))
                             (and s₁ s₂ (not-b s₁) (not-b s₂))))
                booleano)))
    (λ (s) (ormap (λ (b) (let ((b (walk b s)))
                           (not (or (var? b) (boolean? b)))))
              booleano))
    (λ (s) (ormap (λ (b) (ormap (λ (y) (same-s? y b s)) symbolo))
              booleano)))
```

The following is an example of its use.

```
> (call/initial-state 1
    (call/fresh (λ (x)
                  (conj (=/= #f x)
                        (conj (=/= #t x)
                              (booleano x))))))
'()
```

7.2 Implementing `listo`

Checking `listo` is more complicated still. Consequently some of the constraint-violation predicates are also quite complex. We add four independent predicates to properly implement `listo`.

In the first of these, we look for an instance in which the end of a term labeled a proper list `l` is required to be a symbol. The function `walk-to-end` recursively walks the `cdr` of a term `x` in a substitution `s` and returns the final `cdr` of `x` relative to `s`. We use it in constraint-violation predicates related to the `listo` constraint.

```
#| Term × Subst → Bool |#
(define (walk-to-end x s)
  (let ((x (walk x s)))
    (if (pair? x) (walk-to-end (cdr x) s) x)))
```

The second predicate resembles the first, except it checks for a Boolean instead.

```
> (make-constraint-system (=/= absento symbolo not-pairo booleano listo)
    ...
    (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                           (ormap (λ (y) (same-s? y end s)) symbolo)))
             listo))
    (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                           (ormap (λ (b) (same-s? b end s)) booleano)))
             listo))
    (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                           (let ((s^ (unify end '() s)))
                             (and s^
                                  (ormap (λ (n) (same-s? end n s)) not-pairo)
                                  (or (ormap (λ (pr) (same-s? (car pr) (cdr pr) s^)) =/=)
                                      (ormap (λ (pr) (mem? (car pr) (cdr pr) s^)) absento))))))
             listo))
    (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                           (ormap (λ (pr) (and (null? (walk (car pr) s))
                                               (mem? end (cdr pr) s)))
                             absento)))
             listo))
    ...)
```

In the third, we check for a proper list that must have a definite fixed last cdr (the `end`) under the substitution. This means either `end` already is `()`, or a `not-pairo` constrains `end`. If, in addition, either `=/=` or `absento` constraints forbid `end` from being `()`, then that is a violation. The following example demonstrates this behavior.

```
> (call/initial-state 1
    (call/fresh (λ (x)
                  (conj (listo x)
                        (conj (not-pairo x)
                              (disj (=/= '() x)
                                    (absento x '()))))))))
'()
```

In the last predicate we require to correctly implement `listo`, `end` can be a proper list of unknown length. An `absento` constraint forbidding `()` from occurring in a term containing `end`, however, causes a violation. The constraint must precisely forbid `()` from occurring in a term containing `end` to cause the violation.

```
> (call/initial-state 1
    (call/fresh (λ (x)
                  (conj (listo x)
                        (absento '() x)))))
'()
```

These constraint-violation predicates are somewhat involved—of necessity. We have ensured that constraint violations can each be treated independently and that they comprise the entirety of the constraint domain knowledge required. Furthermore, by requiring that our solver be monotonic and logical, we have ensured that adding new constraints never requires the language designer to modify existing predicates.

8 Related work

The modern development of CLP languages begins in the mid 1980s by groups in Melbourne, Marseilles, and the ECRC. The CLP scheme [16] is an important development from this era. The CLP scheme separates the inference mechanism from the constraint handling and satisfaction. It subsumes many individual logic programming extensions and provides a theoretical foundation for disparate CLP languages.

Schrijvers et al. offer different a motivation for separating constraint solving and search [27]. They implement different advanced search strategies via monad transformers over basic search monads. It's not yet clear where miniKanren's interleaving DFS search fits into their framework, although this is a topic we are currently investigating.

There exists a close connection between microKanren (and thus also miniKanren) and pure Prolog. Spivey and Seres's present embed a similar subset of Prolog work on a Haskell embedding of Prolog [28], Kiselyov's "Taste of Logic Programming" [21], and of course Ralf Hinze's extensive work on implementations of Prolog-style backtracking [13, 14] are all closely related to our microKanren as well.

There exists a different sort of CLP paradigm based on research in constraint satisfaction problems using constraint propagation to reduce the search space. cKanren, an earlier miniKanren for CLP, takes this different approach and uses domain restriction and constraint propagation [1]. Alvis et al. take as their primary example finite domains. cKanren returns as answers ground instances that satisfy the program's constraints. Unlike languages generated by our framework, they provide constraint minimization and an answer-formatter in their implementation. We presented a preliminary (non-archival) draft version of the current work at Scheme Workshop 2015 [11].

9 Conclusion

We have presented a framework for developing microKanren-like CLP languages in an instance of the CLP scheme. Decoupling the constraint management from the inference, control, and variable management has helped to clarify the behavior of microKanren. We implement the customary miniKanren constraints as well as interesting and useful new ones.

In our implementation we deliberately reject certain common optimizations and features that would have complicated our implementation. We generate black-box constraint solvers, rather than simplifying solvers. These generated constraint solvers are not at all specialized for incremental constraint solving, to the point of not even checking for duplicate constraints. We do not minimize even the answer constraint, nor do we perform any answer projection.

We do not intend to generate efficient, state-of-the-art CLP languages, and on that front we have surely succeeded. Instead of efficiency, our aim is a simple, general framework for implementing constraints in microKanren. We envision our framework as a lightweight tool

for rapidly prototyping constraint sets. Language designers can explore and test constraint definitions and interactions without building or modifying a complicated and efficient dedicated solver. We also imagine it as an educational artifact that provides functional programmers a minimal executable instance of the CLP scheme.

Although we have preferred simplicity over performance here, we hope to investigate the performance impacts of various simple optimizations including incremental constraint solving, early projection [9], attributed variables [15], or calling out to an appropriate dedicated constraint solver. We are especially interested in developing these optimizations as series of correctness-preserving transformations.

In future work we also hope to build an extensible, generic constraint simplification framework analogous to our framework for building constraint solvers. The language designer should have to write only the individual constraint simplification predicates for the framework to produce a simplifier. Ideally this framework will infer an efficient order in which to execute these minimization functions based on an abstract interpretation. We also want to formalize the meaning of a "kind" of constraint-violation. Defining precisely what violations a single predicate should check will clarify the language designer's precise responsibilities.

As it exists ours is a clear, simple framework for generating miniKanren languages with constraints and serves as a test-bed for developing constraint systems and an artifact of study. Further, it serves as a foundation for continued future work in designing constraint systems.

Acknowledgements

We thank Will Byrd, Chung-chieh Shan, and Oleg Kiselyov for early discussions of constraints in miniKanren. We thank Ryan Culpepper for his improvements to the framework macros. We also thank our anonymous reviewers for their suggestions and improvements.

References

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

[2] Bernd Braßel, Sebastian Fischer & Frank Huch (2008): *Declaring Numbers. Electronic Notes in Theoretical Computer Science* 216, pp. 111–124, doi:10.1016/j.entcs.2008.06.037. Available at `http://dx.doi.org/10.1016/j.entcs.2008.06.037`.

[3] William E. Byrd & Daniel P. Friedman (2007): *αKanren: A Fresh Name in Nominal Logic Programming*. In: *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701*, pp. 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*).

[4] William E. Byrd, Eric Holk & Daniel P. Friedman (2012): *miniKanren, live and untagged*. In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming - Scheme '12*, Association for Computing Machinery (ACM), doi:10.1145/2661103.2661105. Available at `http://dx.doi.org/10.1145/2661103.2661105`.

[5] Keith L. Clark (1978): *Negation as Failure*. In: *Logic and Data Bases*, Springer Science + Business Media, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11. Available at `http://dx.doi.org/10.1007/978-1-4684-3384-5_11`.

[6] Hubert Comon (1994): *Constraints in Term Algebras (Short Survey)*. In: *Algebraic Methodology and Software Technology (AMAST'93)*, Springer Science + Business Media, pp. 97–108, doi:10.1007/978-1-4471-3227-1_9. Available at http://dx.doi.org/10.1007/978-1-4471-3227-1_9.

[7] RYAN CULPEPPER (2012): *Fortifying macros*. *J. Funct. Prog.* 22(4-5), pp. 439–476, doi:10.1017/s0956796812000275. Available at http://dx.doi.org/10.1017/s0956796812000275.

[8] Matthew Flatt & PLT (2010): *Reference: Racket*. Technical Report PLT-TR-2010-1, PLT Design Inc. http://racket-lang.org/tr1/.

[9] Andreas Fordan (1999): *Projection in Constraint Logic Programming*. Ios Press.

[10] Daniel Friedman, William E. Byrd & Oleg Kiselyov (2005): *The Reasoned Schemer*. MIT Press, Cambridge, Mass.

[11] Jason Hemann & Daniel P. Friedman (2015): *A Framework for Extending microKanren with Constraints*. In: *16th Workshop on Scheme and Functional Programming*. Forthcoming.

[12] Jason Hemann, Daniel P Friedman, William E Byrd & Matthew Might (2016): *A Small Embedding of Logic Programming with a Simple Complete Search*. In: *Proceedings of the 12th Symposium on Dynamic Languages*, ACM.

[13] Ralf Hinze (2000): *Deriving backtracking monad transformers*. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP '00*, Association for Computing Machinery (ACM), doi:10.1145/351240.351258. Available at http://dx.doi.org/10.1145/351240.351258.

[14] Ralf Hinze (2001): *Prolog's control constructs in a functional setting Axioms and implementation*. *International Journal of Foundations of Computer Science* 12(02), pp. 125–170, doi:10.1142/S0129054101000436. Available at http://www.worldscientific.com/doi/abs/10.1142/S0129054101000436.

[15] Serge Le Huitouze (1990): *A new data structure for implementing extensions to Prolog*. In: *Programming Language Implementation and Logic Programming*, Springer Science + Business Media, pp. 136–150, doi:10.1007/bfb0024181. Available at http://dx.doi.org/10.1007/bfb0024181.

[16] J. Jaffar & J.-L. Lassez (1987): *Constraint logic programming*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*, Association for Computing Machinery (ACM), doi:10.1145/41625.41635. Available at http://dx.doi.org/10.1145/41625.41635.

[17] Joxan Jaffar, Michael Maher, Kim Marriott & Peter J Stuckey (1998): *The semantics of constraint logic programs*. *The Journal of Logic Programming* 37(1), pp. 1–46.

[18] Joxan Jaffar & Michael J. Maher (1994): *Constraint logic programming: a survey*. *The Journal of Logic Programming* 19-20, pp. 503–581, doi:10.1016/0743-1066(94)90033-7. Available at http://dx.doi.org/10.1016/0743-1066(94)90033-7.

[19] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey & Roland H. C. Yap (1993): *Projecting CLP(ℛ) constraints*. *New Gener Comput* 11(3-4), pp. 449–469, doi:10.1007/bf03037187. Available at http://dx.doi.org/10.1007/bf03037187.

[20] Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd & Daniel P. Friedman (2009): *A Pattern-matcher for miniKanren -or- How to Get into Trouble with CPS Macros*. In: *In Proceedings of the 2009 Workshop on Scheme and Functional Programming, Cal Poly Technical Report CPSLO-CSC-09-03*, pp. 37–45.

[21] Oleg Kiselyov (2006): *The taste of logic programming*. Available at http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[22] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman & Chung chieh Shan: *Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)*. In: *Functional and Logic Programming*, Springer Science + Business Media, pp. 64–80, doi:10.1007/978-3-540-78969-7_7. Available at http://dx.doi.org/10.1007/978-3-540-78969-7_7.

[23] Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, interleaving, and terminating monad transformers*. 40, Association for Computing Machinery (ACM), p. 192, doi:10.1145/1090189.1086390. Available at http://dx.doi.org/10.1145/1090189.1086390.

[24] Ramana Kumar (2010): *Mechanising Aspects of miniKanren in HOL*. Australian National University. Bachelors thesis.

[25] J.W. Lloyd & J.C. Shepherdson (1991): *Partial evaluation in logic programming*. *The Journal of Logic Programming* 11(3-4), pp. 217–242, doi:10.1016/0743-1066(91)90027-m. Available at http://dx.doi.org/10.1016/0743-1066(91)90027-m.

[26] Joseph P. Near, William E. Byrd & Daniel P. Friedman: *αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic*. In: *Logic Programming*, Springer Science + Business Media, pp. 238–252, doi:10.1007/978-3-540-89982-2_26. Available at http://dx.doi.org/10.1007/978-3-540-89982-2_26.

[27] Tom Schrijvers, Peter Stuckey & Philip Wadler (2009): *Monadic constraint programming*. *J. Funct. Prog.* 19(06), p. 663, doi:10.1017/s0956796809990086. Available at http://dx.doi.org/10.1017/s0956796809990086.

[28] JM Spivey & Silvija Seres (1999): *Embedding Prolog in Haskell*. In: *Proceedings of Haskell Workshop*, 99, pp. 1999–28.

[29] Mark Wallace (2002): *Constraint Logic Programming*. In: *Computational Logic: Logic Programming and Beyond*, Springer Science + Business Media, pp. 512–532, doi:10.1007/3-540-45628-7_19. Available at http://dx.doi.org/10.1007/3-540-45628-7_19.

Appendix: microKanren

```
(define (var n) n)
(define (var? n) (number? n))
(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond ((var? v) (eqv? x v))
          ((pair? v) (or (occurs? x (car v) s)
                         (occurs? x (cdr v) s)))
          (else #f))))
(define (ext-s x v s)
  (cond ((occurs? x v s) #f)
        (else `((,x . ,v) . ,s))))
(define (walk u s)
  (let ((pr (assv u s)))
    (if pr (walk (cdr pr) s) u)))
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond ((eqv? u v) s)
          ((var? u) (ext-s u v s))
          ((var? v) (ext-s v u s))
          ((and (pair? u) (pair? v))
           (let ((s (unify (car u) (car v) s)))
             (and s (unify (cdr u) (cdr v) s))))
          (else #f))))
(define ((call/fresh f) S/c)
  (let ((S (car S/c)) (c (cdr S/c)))
    ((f (var c)) `(,S . ,(+ 1 c)))))
```

```
(define ($append $₁ $₂)
  (cond ((null? $₁) $₂)
        ((promise? $₁)
         (delay/name ($append $₂ (force $₁))))
        (else
          (cons (car $₁) ($append (cdr $₁) $₂)))))
(define ($append-map g $)
  (cond ((null? $) `())
        ((promise? $)
         (delay/name ($append-map g (force $))))
        (else ($append (g (car $))
                       ($append-map g (cdr $))))))
(define ((disj g₁ g₂) S/c) ($append (g₁ S/c) (g₂ S/c)))
(define ((conj g₁ g₂) S/c) ($append-map g₂ (g₁ S/c)))
(define (pull $) (if (promise? $) (pull (force $)) $))
(define (take n $)
  (cond ((null? $) '())
        ((and n (zero? (- n 1))) (list (car $)))
        (else (cons (car $)
                (take (and n (- n 1))
                      (pull (cdr $)))))))
(define (call/initial-state n g)
  (take n (pull (g `(,S₀ . 0)))))
(define-syntax-rule (define-relation (rid . args) g)
  (define ((rid . args) S/c) (delay/name (g S/c))))
```