cKanren

miniKanren with Constraints

Claire E. Alvis Jeremiah J. Willcock Kyle M. Carter
William E. Byrd Daniel P. Friedman

School of Informatics and Computing, Indiana University, Bloomington, IN 47405
{calvis,jewillco,kylcarte,webyrd,dfried}@cs.indiana.edu

Abstract

We present cKanren, a framework for constraint logic programming (CLP) in Scheme. cKanren subsumes miniKanren, a logic programming language embedded in Scheme. cKanren allows programmers to easily use, define, and combine different kinds of constraints. We provide two example constraint systems: one over finite domains and one over trees.

The cKanren framework is designed to encourage an especially pure style of logic programming in which goals can be reordered arbitrarily without affecting a program's semantics (with an important decidability-related caveat).

We develop the complete implementation of the cKanren framework, written in R^6RS Scheme extended with SRFI 39 parameters. We present the implementation of cKanren's finite domain and disequality constraint solvers, and we provide introductions to miniKanren, cKanren, and numerous example programs, including the Send More Money cryptarithmetic puzzle and N-Queens.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords CLP, CLP(FD), Scheme, miniKanren, logic programming, constraint solving, constraint logic programming

1. Introduction

We present a complete implementation of cKanren, a framework for constraint logic programming in Scheme. Traditional logic programming provides only a single constraint: equality over terms, which is implemented using unification. Constraint logic programming (Apt 2003; Jaffar and Maher 1994), also known as *CLP*, supports additional constraints, such as constraints over finite domains *CLP(FD)* and tree terms *CLP(Tree)*.

The cKanren framework is an extension of miniKanren, which embeds logic programming in Scheme (Friedman et al. 2005; Byrd and Friedman 2006; Byrd 2009). Unlike previous extensions to miniKanren, such as αKanren (Byrd and Friedman 2007), cKanren does not *directly* add user-level operators to miniKanren. Rather, cKanren allows programmers to use, create, and combine constraint systems. cKanren is implemented in R^6RS Scheme (Sperber et al. 2007), and uses the SRFI-39 parameter mechanism (Feeley 2002) supported by multiple Scheme implementations (Flatt and PLT 2010; Dybvig 2010).

In addition to describing the cKanren framework, we present constraints for finite domains and tree terms. Constraints over finite domains allow miniKanren programmers to declaratively reason about finite sets of values, based on a restricted set of relational arithmetic operators. Disequality constraints over tree terms allow miniKanren programmers to express a limited but useful form of negation: that two terms are not equal and can never be made equal.

Our paper makes the following contributions:

- We describe the two kinds of constraints implemented within our cKanren framework: $\leqslant^{fd}$, $+^{fd}$, $\not\equiv^{fd}$, and *alldiff*^{fd} constraints over finite domains (Section 2.1.1), and $\not\equiv$ disequality constraints over tree terms (Section 2.1.2).
- We provide examples and exercises (Section 2.1.3) using both kinds of constraints, including the famous Send More Money and N-Queens problems, along with $rember^o$, a program that demonstrates the need for disequality constraints. We present our solutions in Section 4.
- We present complete implementations of the cKanren constraint framework (Section 3.1), and constraints over finite domains (Section 3.3) and tree terms (Section 3.5).[1]
- We demonstrate how to create new kinds of constraints by composing existing cKanren constraint systems (Section 5).
- We describe the "miniKanren philosophy" that informed the development of cKanren, and discuss several important design and implementation goals (Section 6).
- We present cKanren's helper definitions and an updated implementation of core miniKanren in the appendices. These appendices along with the code in the body of the paper comprise a working implementation.

We begin with an overview of the core miniKanren language.

[1] The cKanren implementation can be downloaded from `github.com/calvis/cKanren`.

2. The Language

We present three languages: miniKanren, miniKanren extended with constraints over finite domains, and miniKanren extended with tree disequality. cKanren provides a framework for writing constraints and for using those kinds of constraints within miniKanren programs. We then show the finite domain and disequality goals in action, through examples such as Send More Money, N-Queens, and *rember*^{o}.

2.1 miniKanren

In this section we briefly review the miniKanren language; readers already familiar with miniKanren can safely skip to Section 2.1.1, while those wishing to learn more about the language should see Byrd and Friedman (2007) (from which the first part of this section has been adapted) and Friedman et al. (2005).

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. By our convention, names of relations end with a superscript o—for example *any*^{o}, which is entered as `anyo`. Relational operators do not follow this convention: ≡ (entered as `==`), **cond**^{e} (entered as `conde`), and **fresh** (formerly **exist**). Similarly, (**run**^{5} (*q*) *body*) and (**run*** (*q*) *body*) are entered as `(run 5 (q) body)` and `(run* (q) body)`, respectively.

miniKanren extends Scheme with three operators: ≡, **cond**^{e}, and **fresh**. There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

fresh, which syntactically looks like **lambda**, introduces into scope new lexical variables bound to new (logic) variables; ≡ unifies two terms. Thus

(**fresh** (*x y z*) (≡ *x z*) (≡ 3 *y*))

would associate *x* with *z* and *y* with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

(**run**^{1} (*q*) (**fresh** (*x y z*) (≡ *x z*) (≡ 3 *y*))) ⇒ ($_{\_0}$)

The value returned is a list containing the single value $_{\_0}$; we say that $_{\_0}$ is the *reified value* of the unbound variable *q* and thus can be any value. *q* also remains unbound in

(**run**^{1} (*q*) (**fresh** (*x y*) (≡ *x q*) (≡ 3 *y*))) ⇒ ($_{\_0}$)

We can get back other values, of course.

```
(run^1 (y)              (run^1 (q)              (run^1 (y)
  (fresh (x z)            (fresh (x z)            (fresh (x y)
    (≡ x z)                 (≡ x z)                 (≡ 4 x)
    (≡ 3 y)))               (≡ 3 z)                 (≡ x y))
                            (≡ q x)))             (≡ 3 y))
```

Each of these examples returns (3); in the rightmost example, the *y* introduced by **fresh** is different from the *y* introduced by **run**. A **run** expression can also evaluate to the empty list. This indicates that there does not exist any value of the variable bound by the **run** expression that can cause its body to succeed.

(**run**^{1} (*x*) (≡ 4 3)) ⇒ ()

We use **cond**^{e} to get several values. Syntactically, **cond**^{e} looks like **cond** but without ⇒ or **else**. For example,

```
(run^2 (q)
  (fresh (w x y)
    (cond^e
      ((≡ `(,x ,w ,x) q)
       (≡ y w))
      ((≡ `(,w ,x ,w) q)
       (≡ y w))))) ⇒ ((_0 _1 _0) (_0 _1 _0))
```

Although the two **cond**^{e} lines are different, the values returned are identical. This is because distinct reified unbound variables are assigned distinct subscripts, increasing from left to right—the numbering starts over again from zero within each value, which is why the reified value of *x* is $_{\_0}$ in the first value but $_{\_1}$ in the second value. The superscript 2 in **run** denotes the maximum length of the resultant list. If the superscript ∗ is used, then there is no maximum imposed. This can easily lead to infinite loops:

```
(run* (q)
  (let loop ()
    (cond^e
      ((≡ #f q))
      ((≡ #t q))
      ((loop)))))
```

Had ∗ been replaced by a natural number *n*, then an *n*-element list of alternating #f's and #t's would be returned. The **cond**^{e} succeeds while associating *q* with #f, which accounts for the first value. When getting the second value, the second **cond**^{e} line is tried, and the association made between *q* and #f is forgotten—we say that *q* has been *refreshed*. In the third **cond**^{e} line, *q* is refreshed again.

We now look at several interesting examples that rely on *any*^{o}, which tries *g* an unbounded number of times.

```
(define any^o
  (λ (g)
    (cond^e
      (g)
      ((any^o g)))))
```

Consider the first example,

```
(run* (q)
  (cond^e
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

which does not terminate because the call to *any*^{o} succeeds an unbounded number of times. If ∗ were replaced by 5, then we would get (#t #f #f #f #f). (The user should not be concerned with the order in which values are returned.)

Now consider

```
(run^10 (q)
  (any^o
    (cond^e
      ((≡ 1 q))
      ((≡ 2 q))
      ((≡ 3 q))))) ⇒ (1 2 3 1 2 3 1 2 3 1)
```

Here the values 1, 2, and 3 are interleaved; our use of *any*^{o} ensures that this sequence is repeated indefinitely.

Even if some **cond**^{e} lines loop indefinitely, other **cond**^{e} lines can contribute to the values returned by a **run** expression. However, we are not concerned with expressions looping indefinitely. For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns (1 2 3); replacing **run**^{3} with **run**^{4} would cause divergence since there are only three values and $never^o$ would loop indefinitely looking for the fourth.

2.1.1 miniKanren with Finite Domain Constraints

Finite domain constraints allow the user to assign a finite domain to a variable and to use mathematical relations between variables such as $\leqslant$, $<$, $\neq$, and $+$. Termination is guaranteed when programmers limit themselves to **fresh**, $\equiv$, **cond**^{e}, and the finite domain operators, since the domains are finite (Jaffar and Maher 1994). However, this guarantee no longer holds when using recursion.

We introduce five goals below, where n^* denotes a non-empty list of strictly increasing natural numbers, x denotes a variable, u, v, and w denote arbitrary values, and v^* denotes either a list of values or a variable eventually associated with a list of values. In finite domains, any non-variable value should be a natural number.

Names that end with *fd* and have no subscript like the ones below comprise the user interface: dom^{fd} and four constraints.

- $(dom^{fd}\ x\ n^*)$ (entered as `domfd`) constrains $x \in n^*$. If $n^* = (n)$, then $(dom^{fd}\ x\ n^*)$ is the same as $(\equiv\ x\ n)$.
- $(\leqslant^{fd}\ u\ v)$ (entered as `<=fd`) constrains $u \leqslant v$.
- $(+^{fd}\ u\ v\ w)$ (entered as `plusfd`) constrains $u + v = w$.
- $(\not\equiv^{fd}\ u\ v)$ (entered as `=/=fd`) constrains $u \neq v$.
- $(all\text{-}diff^{fd}\ v^*)$ (entered as `all-difffd`) ensures that all variables and values found within the flat list v^* are different from each other.

The following examples illustrate these constraints. In the first example, we state that q is never the same as 2. Then, when we state that q can only be a 1, 2, or 3, we exclude 2 from being a possible value of q.

```
(run* (q)
  (≠^fd q 2)
  (dom^fd q (1 2 3))) ⇒ (1 3)
```

A variable's domain can be defined at any time within the variable's scope. We use the derived goal (a goal that is defined in terms of other goals) **in**^{fd} which allows for assigning a domain to several different variables. Associating a variable with more than one domain, however, assigns the intersection of those domains to the variable.

```
(define-syntax in^fd
  (syntax-rules ()
    ((_ x_0 x ... e)
     (let ((n* e))
       (fresh () (dom^fd x_0 n*) (dom^fd x n*) ...)))))
```

In the next example below, when we write $(\equiv\ x\ y)$, we are stating that x is going to be the same as y in every answer, so $(\equiv\ q\ `(,x\ ,y\ ,z))$ could have been $(\equiv\ q\ `(,x\ ,x\ ,z))$ and $(\mathbf{in}^{fd}\ y\ '(3\ 4\ 5))$ is the same as $(\mathbf{in}^{fd}\ x\ '(3\ 4\ 5))$.

```
(run* (q)
  (fresh (x y z)
    (in^fd z '(1 3 5 6 7 8))
    (≡ x y)
    (in^fd y '(3 4 5))
    (≡ q `(,x ,y ,z))
    (in^fd z '(5 6 9))
    (in^fd x '(1 2 3)))) ⇒ ((3 3 5) (3 3 6))
```

Even if a variable is not bound to a constant by the end of a program (as is the case with z in the example above), the variable will be associated with any satisfiable value in the domain of that variable if it is included in the returned expression.

We introduce the derived goal $<^{fd}$ along with the very useful function *range*, which, given two natural numbers, returns a list of all the numbers between *lb* and *ub*, inclusively.

```
(define <^fd
  (λ (u v)
    (fresh () (≤^fd u v) (≠^fd u v))))

(define range
  (λ (lb ub)
    (cond
      ((< lb ub) (cons lb (range (+ lb 1) ub)))
      (else (cons lb '())))))
```

The following is a simple example of $<^{fd}$ and *range* in action.

```
(run* (x)
  (≤^fd x 7)
  (<^fd 2 x)
  (in^fd x (range 0 10))) ⇒ (3 4 5 6 7)
```

Variables used with finite domain constraints must have domains. As a result,

```
(run* (q)
  (fresh (x y)
    (<^fd x y)
    (<^fd y x)))
```

signals an error; an alternative would be to fail, which would be unfriendly to the user.

Unsatisfiable constraints, even when the variables are not referenced or associated with the run variable in any way, still result in failure.

```
(run* (q)
  (fresh (x y z)
    (in^fd x y z '(1 2))
    (all-diff^fd `(,x ,y ,z))
    (≡ q 5))) ⇒ ()
```

A variant of this example has a domain of three values.

```
(run* (q)
  (fresh (x y z)
    (in^fd x y z '(1 2 3))
    (all-diff^fd `(,x ,y ,z))
    (≡ q x))) ⇒ (1 2 3)
```

Here each element of x's domain shows up as a value. But, consider this expression.

```
(run* (q)
  (fresh (x y z)
    (in^fd x y z '(1 2 3))
    (all-diff^fd `(,x ,y ,z))
    (≡ q `(,x ,z)))) ⇒ ((1 2) (1 3) (2 1) (3 1) (2 3) (3 2))
```

Had $(\equiv q\ `(,x\ ,z))$ been $(\equiv q\ 5)$, the result would have been (5), because cKanren ignores domain variables that are not associated with the variable bound by the **run** expression.

Here is a simple example of *all-diff*^{fd}.

```
(run* (q)
  (in^fd q (range 3 6))
  (all-diff^fd `(2 3 ,q))) ⇒ (4 5 6)
```

We want values for q that satisfy the *all-diff*^{fd} goal. We observe that if we choose 3, we will have two occurrences of 3, so that value is not included in the list of answers associated with q. But, if we try either 4, 5, or 6, then we observe that none of them are the same as 2 or 3.

Now, consider this **run*** expression.

```
(run* (q)
  (fresh (x y z)
    (in^fd x y z (range 1 5))
    (<^fd z x)
    (+^fd y 2 z)
    (≡ q `(,x ,y ,z)))) ⇒ ((4 1 3) (5 1 3) (5 2 4))
```

2.1.2 miniKanren with Tree Disequality

We now introduce disequality constraints using ≠ (entered as `=/=`), which works on arbitrary values, using the same restrictions imposed on miniKanren.

```
(run* (q)
  (fresh (x y)
    (cond^e
      ((≡ x 1) (≡ y 1))
      ((≡ x 2) (≡ y 2))
      ((≡ x 1) (≡ y 2))
      ((≡ x 2) (≡ y 1)))
    (≠ x y)
    (≡ q `(,x ,y)))) ⇒ ((1 2) (2 1))
```

The next example relies on *all-diff*^{o}, a derived goal that uses ≠. *all-diff*^{o} takes a list and succeeds as long as all the values in the list are different at all points in the program.

```
(define all-diff^o
  (λ (l)
    (cond^e
      ((≡ l '()))
      ((fresh (a) (≡ l `(,a))))
      ((fresh (a ad dd)
         (≡ l `(,a ,ad . ,dd))
         (≠ a ad)
         (all-diff^o `(,a . ,dd))
         (all-diff^o `(,ad . ,dd)))))))
```

Our final example mimics the last *all-diff*^{fd} example but with an unbounded, instead of bounded, result.

```
(run^1 (q) (all-diff^o `(2 3 ,q)))
⇒ ((_0 : (≠ ((_0 . 2)) ((_0 . 3)))))
```

Thus, any value for q suffices, provided it is neither 2 nor 3. This is an unbounded number of answers. Because of the imposed bound required when using *all-diff*^{fd}, however, the number of answers is bounded.

2.1.3 Exercises

- Finite domains can be used to solve standard cryptarithmetic problems, such as finding the correct letter values to satisfy the following equation:

$$\begin{array}{rr} & S\ E\ N\ D \\ + & M\ O\ R\ E \\ \hline & M\ O\ N\ E\ Y \end{array}$$

  Each letter represents a different digit in the range 0 through 9, and the two leading digits, S and M, should be nonzero.

- n queens are placed on an $n \times n$ chessboard so that no two queens can attack one another. Of course, we want to find all the solutions.

- *The Reasoned Schemer* (Friedman et al. 2005) shows how to derive recursive relational programs from recursive functional programs. But, our approach sometimes fails to work! Consider,

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) '())
      (else
       (let ((a (car ls)) (d (cdr ls)))
         (let ((res (rember x d)))
           (cond
             ((equal? a x) res)
             (else `(,a . ,res)))))))))
```

  and its derived recursive relation.

```
(define rember^o
  (λ (x ls out)
    (cond^e
      ((≡ '() ls) (≡ '() out))
      ((fresh (a d res)
         (≡ `(,a . ,d) ls)
         (rember^o x d res)
         (cond^e
           ((≡ a x) (≡ res out))
           ((≡ `(,a . ,res) out))))))))

(run* (q) (rember^o 'a '(a b a c) q))
⇒ ((b c) (b a c) (a b c) (a b a c))
```

  Is this the correct answer? The first list removes all occurrences of a, so it seems so. But then the second list removes the first occurrence of a; the third list removes the second occurrence of a; and the last list removes no occurrences of a. So, we know the last three lists are wrong. And worse, how can the following example succeed?

```
(run* (q) (rember^o 'a '(a b c) '(a b c))) ⇒ (_0)
```

Solutions to the exercises can be found on page 12.

3. Implementation

In this section, we introduce the framework that we use to implement the constraint systems. Part of understanding the framework is to get an intuitive feel for how it works in the presence of any ordering of the goals. That's the point of Section 3.1.1. Following that, we show the specifics of each constraint system. Definitions that are neither in the body of the paper nor in R^6RS are defined in the appendices.

3.1 Framework

The cKanren framework has been designed to be as flexible as possible, so users can define their own constraints with ease. We implement operators for unification, constraint propagation, and satisfiability that can be extended with user-defined functions.

3.1.1 Core Data Structures

All information is kept in a package a, with three different stores. First, the substitution s contains all associations directly resulting from unification. Second, d is a store for all the domains of variables. This store is a list of (*var* . *domain*) pairs, where *domain* is a non-empty (ordered small to large, with no duplicates) list of natural numbers. Finally, the constraint store c contains only predefined constraints that have been encountered, in normalized form. In the first two stores, s and d, each association is a variable paired with an associated value. The constraint store is not an association list; instead, it contains a list of operator constraints (described below).

Although there are other approaches for organizing a package, we have chosen this way for simplicity and readability. Each kind of information is separated so the user can see organized output for debugging; retrieval functions have less information to sift through; and there is no need for dummy indicator values when variables are unassociated in s, do not have a domain in d, or are unconstrained in c.

We provide a package constructor, and an accessor that lexically binds variables to each different store.[2]

```
(define make-a
  (λ (s d c)
    (cons s (cons d c))))

(define-syntax λM
  (syntax-rules (:)
    ((_ (a : s d c) body)
     (λ (a)
       (let ((s (car a)) (d (cadr a)) (c (cddr a)))
         body)))
    ((_ (a) body) (λ (a) body))))
```

In addition, we define two related operators. When $identity_{\mathsf{M}}$ is passed as the second argument to $compose_{\mathsf{M}}$, then $(compose_{\mathsf{M}}\ f_{\mathsf{M}}\ \hat{f}_{\mathsf{M}}) = f_{\mathsf{M}}$, since for all a, (**and** a a) $= a$.

```
(define identity_M (λM (a) a))

(define compose_M
  (λ (f_M f̂_M)
    (λM (a)
      (let ((a (f_M a)))
        (and a (f̂_M a))))))
```

[2] Variations on λ_{M} could generate simpler code when s, d, or c is not free in *body*. We propose defining those macros as an exercise for the reader.

Each store is initially the empty list but can be extended using specialized functions. *ext-s* and *ext-d* simply extend an association list, whereas *ext-c* extends the constraint store provided the new constraint has at least one variable in it. This ensures that constraints strictly between constants are not in the constraint store.

Although what is contained within a constraint differs depending on which constraints are used, all constraints that reside in c look like '(,(op_{c} *arg* ...) $\mathsf{op_c}$,*arg* ...). Each such constraint has an operator name, $\mathsf{op_c}$, which is the name of the helper function called directly or indirectly from the associated constraint constructor. In programs, we use the lexical variable *oc* to refer to such an "*o*perator *c*onstraint." Here is the definition of *ext-c*.

```
(define ext-c
  (λ (oc c)
    (cond
      ((any/var? (oc→rands oc)) (cons oc c))
      (else c))))
```

The implementor's macro $\mathbf{build}_{\mathsf{oc}}$ (page 15) requires that *arg* ... appear twice. This imposes the same kind of **let** use that appears in **case**. We form hygienically-generated lexical variables, z ..., leading to

```
(let ((z arg) ...)
  '(,(op_c z ...) op_c ,z ...))
```

which is where we set up the code to invoke a helper function, op_{c}, and where we place the *name* of the helper function, $\mathsf{op_c}$. If the result is viewed as a dotted pair, then we get '(,(op_{c} z ...) . ($\mathsf{op_c}$,z ...)), whose *car*, a function call, has been invoked and whose *cdr*, a list, describes the function call.

3.1.2 Watching cKanren Run

We now show how three simple examples run. In the expressions below, we show each step where s, d, or c changes. In the traces of these examples s contains associations to a natural number or to a variable; d contains associations to lists of natural numbers; and c contains the constraints. For ease of reading, the stored procedure, which would have been the first item in an *oc*, has been deleted. Whenever any changes occur while running the constraints in c, these constraints compute a fixpoint. The goal expressions within a **fresh** expression are presumed to be numbered, in the first example, from 1 to 5.

For our first example, we show three variants of the same program, each with the goals in a different order. By changing the order of goals on the same example, we indeed get the same answer, but more importantly, these examples demonstrate how the three stores conspire to produce the correct answer.

```
(run* (q)
  (fresh (x y z)
    (in^fd x z (range 3 5))
    (in^fd y (range 1 4))
    (<^fd x 5)
    (≡ x y)
    (≡ q '(,y ,z)))) ⇒ ((3 3) (4 3) (3 4) (4 4) (3 5) (4 5))
```

In the first two goals, we initialize the domains (see d_1 and d_2, below). In the third goal, we restrict $x < 5$, which is the same as restricting $x \leqslant 5$ and $x \neq 5$ [c_3]. This removes 5 from

x's domain [d_3] and revises the two constraints that comprise the derived constraint $<^{fd}$ (see $\hat{c}_3$). The next step unifies x with y, recording that information in the substitution s_4. Once x and y have been unified, their domains become their common elements [d_4]. (Once in the substitution, x loses its identity and, henceforth, it is only y. That is, each time we look up x, we find that it would be the same as looking up y. [c_4]) Then we form all possible pairs of values of y and z.

$d_1 \Rightarrow ((x\ .\ (3\ 4\ 5))\ (z\ .\ (3\ 4\ 5)))$

$d_2 \Rightarrow ((x\ .\ (3\ 4\ 5))\ (y\ .\ (1\ 2\ 3\ 4))\ (z\ .\ (3\ 4\ 5)))$

$c_3 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ x\ 5)\ (\not\equiv_{\mathsf{c}}^{\mathsf{fd}}\ x\ 5))$
$d_3 \Rightarrow ((x\ .\ (3\ 4))\ (y\ .\ (1\ 2\ 3\ 4))\ (z\ .\ (3\ 4\ 5)))$
$\hat{c}_3 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ x\ 5))$

$s_4 \Rightarrow ((x\ .\ y))$
$d_4 \Rightarrow ((x\ .\ (3\ 4))\ (y\ .\ (3\ 4))\ (z\ .\ (3\ 4\ 5)))$
$c_4 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5))$

Next, by swapping two goals, we get a slightly different view.

```
(run* (q)
  (fresh (x y z)
    (in^fd x z (range 3 5))
    (in^fd y (range 1 4))
    (≡ x y)
    (<^fd x 5)
    (≡ q `(,y ,z)))) ⇒ ((3 3) (4 3) (3 4) (4 4) (3 5) (4 5))
```

We start out with d_1 and d_2 being the same as in the previous example. When we unify x and y, we extend the substitution (see s_3), and shrink y to be those values that are common to x and y [d_3]. Doing so virtually allows us to forget about x. We install y in the constraint that is added, observing that neither x nor y can be 5. The $\not\equiv_{\mathsf{c}}^{\mathsf{fd}}$ constraint has only constants, so it never gets added to c [c_3]. Once again, we form all possible pairs of values of y and z.

$d_1 \Rightarrow ((x\ .\ (3\ 4\ 5))\ (z\ .\ (3\ 4\ 5)))$

$d_2 \Rightarrow ((x\ .\ (3\ 4\ 5))\ (y\ .\ (1\ 2\ 3\ 4))\ (z\ .\ (3\ 4\ 5)))$

$s_3 \Rightarrow ((x\ .\ y))$
$d_3 \Rightarrow ((x\ .\ (3\ 4))\ (y\ .\ (3\ 4))\ (z\ .\ (3\ 4\ 5)))$
$c_3 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5))$

Finally, we move the two $\mathbf{in}^{fd}$ goals.

```
(run* (q)
  (fresh (x y z)
    (≡ x y)
    (<^fd x 5)
    (in^fd z x (range 3 5))
    (in^fd y (range 1 4))
    (≡ q `(,y ,z)))) ⇒ ((3 3) (4 3) (3 4) (4 4) (3 5) (4 5))
```

We see that x is the same as y (see s_1, below). The next goal adds the two predefined constraints from the derived goal [c_2]. Thus, the constraints use y instead of x, since x *is* just y. Then the first $\mathbf{in}^{fd}$ goal runs d_3), placing values in the domain, but those domains must be run with respect to the constraints, which causes y to shrink [$\hat{d}_3$]. The second $\mathbf{in}^{fd}$ goal runs but makes no changes to the domain, since y is (3 4), a subset of (3 4 5). However, c changes by first removing ($\not\equiv_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5$), since every value in y's domain differs from 5 [c_4], and then changes c again, since every value in y's domain is less than or equal to 5 [$\hat{c}_4$]. As before, we form all possible pairs of y and z.

$s_1 \Rightarrow ((x\ .\ y))$

$c_2 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5)\ (\not\equiv_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5))$

$d_3 \Rightarrow ((y\ .\ (3\ 4\ 5))\ (z\ .\ (3\ 4\ 5)))$
$\hat{d}_3 \Rightarrow ((y\ .\ (3\ 4))\ (z\ .\ (3\ 4\ 5)))$

$c_4 \Rightarrow ((\leqslant_{\mathsf{c}}^{\mathsf{fd}}\ y\ 5))$
$\hat{c}_4 \Rightarrow ()$

We next consider how *all-diff*^{fd} and $+^{fd}$ work.

```
(run* (q)
  (fresh (w x y z)
    (in^fd w z (range 1 5))
    (all-diff^fd q)
    (≡ q `(,x ,y ,z))
    (≡ `(,x 2) `(1 ,y))
    (+^fd x y w)
    (+^fd w y z))) ⇒ ((1 2 5))
```

First we associate the two domains in d_1, below. Next, we run c_2, which creates a placeholder in the constraint store that acknowledges that we don't yet know what the variable q will be associated with. On the very next step, we discover that q is associated with the list `(,x ,y ,z). This is acknowledged by changing the kind of constraint (in c_3) to $\mathsf{all\text{-}diff}_{\mathsf{c}}^{\mathsf{fd}}$, since we now know that there are two lists: the unresolved variables and the found values. At this time, we have not found any values. Step 4 changes all three stores. First, the substitution grows by two associations. Once we see that x is 1 and y is 2, then we know that z cannot be either 1 or 2, so they are dropped from z's domain. At the same time we have found the values of x and y, so x and y are dropped from the first list in the constraint and the associated found values are placed, sorted into the second list. In step 5 we add the constraint that states $1 + 2 = w$ which vanishes when we find out that 3 is in w's domain. A similar event happens in step c_6 by adding $3 + 2 = z$. It too vanishes when we find out that 5 is in z's domain. Just before reification, we discover that 5 is not a member of the found list of values.

$d_1 \Rightarrow ((z\ .\ (1\ 2\ 3\ 4\ 5))\ (w\ .\ (1\ 2\ 3\ 4\ 5)))$
$c_2 \Rightarrow ((\mathsf{all\text{-}diff}_{\mathsf{c}}^{\mathsf{fd}}\ q))$

$s_3 \Rightarrow ((q\ .\ (x\ y\ z)))$
$c_3 \Rightarrow ((\mathsf{all\text{-}diff}_{/\mathsf{c}}^{\mathsf{fd}}\ (z\ y\ x)\ ()))$

$s_4 \Rightarrow ((x\ .\ 1)\ (y\ .\ 2)\ (q\ .\ (x\ y\ z))))$
$d_4 \Rightarrow ((z\ .\ (3\ 4\ 5))\ (w\ .\ (1\ 2\ 3\ 4\ 5)))$
$c_4 \Rightarrow ((\mathsf{all\text{-}diff}_{/\mathsf{c}}^{\mathsf{fd}}\ (z)\ (1\ 2)))$

$c_5 \Rightarrow ((+_{\mathsf{c}}^{\mathsf{fd}}\ 1\ 2\ w)\ (\mathsf{all\text{-}diff}_{/\mathsf{c}}^{\mathsf{fd}}\ (z)\ (1\ 2)))$

$c_6 \Rightarrow ((+_{\mathsf{c}}^{\mathsf{fd}}\ 3\ 2\ z)\ (\mathsf{all\text{-}diff}_{/\mathsf{c}}^{\mathsf{fd}}\ (z)\ (1\ 2)))$

$\hat{c}_6 \Rightarrow ((\mathsf{all\text{-}diff}_{/\mathsf{c}}^{\mathsf{fd}}\ (z)\ (1\ 2)))$

The last example shows how $\not\equiv$ works.

```
(run* (q)
  (fresh (w x z)
    (≠ `((1 ,x) ,q #f) `(,z x ,w))
    (≡ z `(1 ,x))
    (≡ w #f)
    (≡ q 'q))) ⇒ (q)
```

By unifying the two values in the empty substitution, we create a substitution that means that the three associations *cannot* all hold (see c_1 below). In s_2, we show that z is associated with $(1\ x)$. Thus, one of the three associations that must not hold now holds. If the remaining two hold, then this expression fails. For the next step, we associate w with #f. This removes the second of the three constraints that "must not hold." The final step associates q with the symbol q, which allows the constraint to be removed because q can never be the same as the symbol x. This accounts for the acceptable final substitution. Had the last goal expression been $(\equiv q\ 'x)$, however, then all three constraints would have held, leading to failure.

$c_1 \Rightarrow (\not\equiv_c^{neq}\ ((z\ .\ (1\ x))\ (y\ .\ x)\ (w\ .\ \#f)))$

$s_2 \Rightarrow ((z\ .\ (1\ x)))$
$c_2 \Rightarrow (\not\equiv_c^{neq}\ ((w\ .\ \#f)\ (q\ .\ x)))$

$s_3 \Rightarrow ((w\ .\ \#f)\ (z\ .\ (1\ x)))$
$c_3 \Rightarrow (\not\equiv_c^{neq}\ ((q\ .\ x)))$

$s_4 \Rightarrow ((q\ .\ q)\ (w\ .\ \#f)\ (z\ .\ (1\ x)))$

So far, we have sketched how substitutions, domains, and constraints work together. The remainder of this section fills in specifics.

3.2 Parameters

The functions within cKanren perform correctly when the three parameters[3] *process-prefix*, *enforce-constraints*, and *reify-constraints* are imported and given new values. Each parameter is initialized with a dummy value (page 15), so each kind of cKanren constraint must update each parameter with its own function. Informally, these functions should present the following interface:

process-prefix This function is sent a prefix of the substitution, consisting of all the associations newly added after a unification. In addition, it is sent the current constraints. This can be an opportunity to rerun constraints for the variables with new associations but different constraints.

enforce-constraints This function is run immediately before we reify the constraints and should accept the variable to be reified. Any checks for consistency or reorganization of the constraint store can be done here.

reify-constraints This function is run as part of the reifier. It is responsible for building a Scheme data structure that represents the information in the constraint store of a package.

[3] As seen in SRFI-39. The same functionality could be captured with global variables and side effects; however, parameters offer the cleanest solution. Prolog systems use modules and predicates with fixed names to allow customization of attribute hooks, rather than parameters.

Assuming these functions are defined prior to runtime, cKanren lays the framework for unification, running constraints, and running entire miniKanren functions.

3.2.1 *goal-construct*

Since most constraint operations should be deterministic, it is often necessary to wrap them in a goal that will succeed when a new package is returned successfully and fail otherwise. This function makes such wrapping easy. (See page 17 for the definition of λ_g, whose values are goals.)

```
(define goal-construct
  (λ (f_M)
    (λ_G (a)
      (cond
        ((f_M a) ⇒ unit_G)
        (else (mzero_G))))))
```

3.2.2 ≡

≡ unifies two arguments, u and v, and can result in three different scenarios. If *unify* (page 18) fails, the result is #f and causes the goal to fail. If the original substitution is returned unchanged, no additional action need occur. In the last case, we obtain new information from unification. That new information (contained in a prefix of the new substitution) is retrieved and passed to the function returned by (*process-prefix*) for further examination.

```
(define ≡
  (λ (u v)
    (goal-construct (≡_c u v))))

(define ≡_c
  (λ (u v)
    (λ_M (a : s d c)
      (cond
        ((unify `((,u . ,v)) s)
         ⇒ (λ (ŝ)
             (cond
               ((eq? s ŝ) a)
               (else
                (let ((p (prefix-s s ŝ))
                      (a (make-a ŝ d c)))
                  (((process-prefix) p c) a))))))
        (else #f)))))

(define prefix-s
  (λ (s ŝ)
    (cond
      ((null? s) ŝ)
      (else (let loop ((ŝ ŝ))
              (cond
                ((eq? ŝ s) '())
                (else (cons (car ŝ) (loop (cdr ŝ)))))))))))

(define #u (≡ #f #t))

(define #s (≡ #f #f))
```

3.2.3 *run-constraints*

When cKanren gets new information about a variable, such as an additional constraint or a more restricted domain, earlier constraints may be affected. Since we keep all constraint information in a store, we can recur through the store when such a change happens to reevaluate constraints.

A naïve approach is to rerun every constraint in the store indiscriminately until there are no more changes in the store (i.e., until a fixpoint is reached).

```
(define run-constraints0
  (λ (x*-ignored c)
    (cond
      ((null? c) identity_M)
      (else
       (compose_M
         (oc→proc (car c))
         (run-constraints0 x*-ignored (cdr c)))))))
```

run-constraints$_0$ performs well on small examples, but this approach is too costly when the constraint store grows. So, in a slightly less naïve approach, the procedure receives a list of variables to look for and runs only those constraints involving those variables. *any-relevant/var?* (page 16) searches the constraint's list of arguments for a variable in x^*.

```
(define run-constraints1
  (λ (x* c)
    (cond
      ((null? c) identity_M)
      ((any-relevant/var? (oc→rands (car c)) x*)
       (compose_M
         (oc→proc (car c))
         (run-constraints1 x* (cdr c))))
      (else (run-constraints1 x* (cdr c))))))
```

Unfortunately, *run-constraints*$_1$ still runs some constraints unnecessarily. Since we run a constraint each time before the recursive call, the constraint store being processed is not up to date. It is possible that a constraint in c no longer exists in the constraint store when the recursion reaches it.

In *run-constraints*, below, once we have found a constraint that contains one or more of the variables we are looking for, we check to make sure that we still need the constraint. The current constraint store is pulled in and, if the constraint is still contained within that store, it is removed and rerun. If the current constraint is not within the current constraint store, a previous constraint has rendered it unnecessary (and it should not run again).

```
(define run-constraints
  (λ (x* c)
    (cond
      ((null? c) identity_M)
      ((any-relevant/var? (oc→rands (car c)) x*)
       (compose_M
         (rem/run (car c))
         (run-constraints x* (cdr c))))
      (else (run-constraints x* (cdr c))))))

(define rem/run
  (λ (oc)
    (λ_M (a : s d c)
      (cond
        ((memq oc c)
         (let ((ĉ (remq oc c)))
           ((oc→proc oc) (make-a s d ĉ))))
        (else a))))))
```

After being run, the constraint might add itself back into the current constraint store, but only when it still has variable arguments. If all arguments are constants, the constraint is not reintroduced.

3.2.4 *reify*

reify takes a variable x and then runs the two goals within (**fresh** () ...). The first goal ensures that the most meaningful answers available are sent to the second goal. The second goal pulls in a package a, then returns the value associated with x in a (along with any relevant constraints), first replacing all variables with symbols representing those entities. A constraint (*proc name* . *rands*) is *relevant* if both *name* and *rands* appear in the value associated with x. We call this process of turning a cKanren value into a Scheme value *reification*.

The first **cond** line in the definition of *reify* below returns only the reified value v associated with x when the rename substitution is empty. If the rename substitution is not empty, it is used to rename the variables in v. Then, if there are no relevant constraints, the renamed value is returned. The **else** line returns both the reified value of x and the reified list of relevant constraints. (The use of *choice*$_G$ and *empty-f* in *reify* is a subtlety to allow `#f` as a value, for example in (**run*** (*q*) (≡ `#f` *q*)) ⇒ (`#f`).)

```
(define reify
  (λ (x)
    (fresh ()
      ((enforce-constraints) x)
      (λ_G (a : s d c)
        (choice_G
          (let* ((v (walk* x s))
                 (r (reify-s v empty-s)))
            (cond
              ((null? r) v)
              (else
               (let ((v (walk* v r)))
                 (cond
                   ((null? c) v)
                   (else
                    (((reify-constraints) v r) a)))))))
          empty-f))))))
```

3.3 Finite Domain Implementation

We created a domain interface powerful enough to allow users to switch domain representations without changing any of the constraint operations. Although we have chosen a simple domain—finite domains as sorted non-empty lists (with no duplicates) of natural numbers—we believe this framework would support a variety of other domains such as symbols, integers, or lists of sorted, nonoverlapping integer intervals. The user need only redefine basic functions such as intersection, difference, and the other simple functions found in Appendix A.

A variable must be associated with either a finite domain or a natural number before reification, but this association can happen at any point in a cKanren program.

While this restriction might cause some programs to be more verbose (for example, it might seem like the constraint ($\leqslant^{fd}$ *x* `5`) should automatically bind the domain (`0 1 2 3 4 5`) to x) such overloading of behavior forces a tradeoff between clean implementation and user convenience; we prefer the former.

3.3.1 The Finite Domain Parameters

We now consider the parameters as broadly described in Section 3.2. Recall that to access a parameter, it is necessary to invoke the parameter as a function of zero arguments. First we define the three parameters: *process-prefix*$_{FD}$, *enforce-*

constraints$_{\mathsf{FD}}$, and *reify-constraints*$_{\mathsf{FD}}$. Once these are defined, we proceed to the specific finite domain constraints.

process-prefix$_{\mathsf{FD}}$ reconsiders each association that was added as a result of unification. Consider an association $(x\ .\ v)$, where x is a variable and v is a constant or a variable. The domain of x is intersected with (the domain of) v and all constraints involving x are rerun. Although doing so does not directly rerun constraints of v, a singleton intersection triggers the constraints of v, as can be seen in the definition of *resolve-storable*$_\delta$, below.

```
(define process-prefix_FD
  (λ (p c)
    (cond
      ((null? p) identity_M)
      (else
        (let ((x (lhs (car p))) (v (rhs (car p))))
          (let ((t (compose_M
                     (run-constraints `(,x) c)
                     (process-prefix_FD (cdr p) c))))
            (λ_M (a : s d c)
              (cond
                ((get_δ x d)
                 ⇒ (λ (δ)
                     ((compose_M (process_δ v δ) t) a)))
                (else (t a)))))))))))
```

process$_\delta$ takes as arguments a value v and a domain δ. If v is a variable, all information is passed to *update-var*$_\delta$. If v is a domain value in δ, then we return a unchanged.

```
(define process_δ
  (λ (v δ)
    (λ_M (a)
      (cond
        ((var? v) ((update-var_δ v δ) a))
        ((memv?_δ v δ) a)
        (else #f)))))
```

In *update-var*$_\delta$ we intersect the two domains: the one associated with v in d and the domain passed into *process*$_\delta$. If the intersection is a singleton, we extend the substitution. Otherwise, we extend the domain with the intersection. If the two domains are disjoint, then we return false. (At this point, we have wrong information in d, but this is fine, since we look up variables in d only when they are *not* in s.)

```
(define update-var_δ
  (λ (x δ)
    (λ_M (a : s d c)
      (cond
        ((get_δ x d)
         ⇒ (λ (x_δ)
             (let ((i (intersection_δ x_δ δ)))
               (cond
                 ((null?_δ i) #f)
                 (else ((resolve-storable_δ i x) a))))))
        (else ((resolve-storable_δ δ x) a))))))

(define resolve-storable_δ
  (λ (δ x)
    (λ_M (a : s d c)
      (cond
        ((singleton?_δ δ)
         (let* ((n (singleton-element_δ δ))
                (a (make-a (ext-s x n s) d c)))
           ((run-constraints `(,x) c) a)))
        (else (make-a s (ext-d x δ d) c))))))
```

We have chosen to extend the substitution directly, rather than using $\equiv$ or *unify*. The presumption is that x has already been *walk*ed (page 18) in the substitution, so we can avoid that extra unification overhead.

enforce-constraints$_{\mathsf{FD}}$ has two purposes. The value associated with x is going to be returned as an answer, so it is desirable to associate x with a constant whenever possible. If x is unified with a variable that has domain information, each realizable domain value should be returned. It is always possible, however, that there are constrained variables that are not returned as part of the final answer. While we do not care about the exact value of every variable, we still need to be sure there exists at least one value for each variable such that all constraints are satisfied or the entire program should fail. *force-ans* recurs through the list of domain variables until it succeeds once; it is forced to stop by *once*^{o} (page 19) and reification can occur.

```
(define enforce-constraints_FD
  (λ (x)
    (fresh ()
      (force-ans x)
      (λ_G (a : s d c)
        (let ((bound-x* (map lhs d)))
          (verify-all-bound c bound-x*)
          ((once^o (force-ans bound-x*)) a)))))))
```

force-ans takes either a variable or a list of variables. If the variable is not associated with a constant in s but does have a domain, *map-sum* (page 16) attempts to associate that variable with everything in its domain. If the value is a pair, the *car* and *cdr* are searched for variables as well. Otherwise, it is already associated with a constant and it succeeds.

```
(define force-ans
  (λ (x)
    (λ_G (a : s d c)
      (let ((x (walk x s)))
        (cond
          ((and (var? x) (get_δ x d))
           ⇒ (map-sum (λ (v) (≡ x v))))
          ((pair? x)
           (fresh ()
             (force-ans (car x))
             (force-ans (cdr x))))
          (else #s))
        a)))))
```

reify-constraints$_{\mathsf{FD}}$, when invoked, produces an error, since there are variables in relevant constraints that are bound neither in s nor in d.

```
(define reify-constraints_FD
  (λ (m r)
    (error 'reify-constraints_FD "Unbound vars at end\n")))
```

We invoke (*use*$_{\mathsf{FD}}$) in order to use constraints over finite domains.

```
(define use_FD
  (λ ()
    (process-prefix process-prefix_FD)
    (enforce-constraints enforce-constraints_FD)
    (reify-constraints reify-constraints_FD)))
```

3.4 $\mathbf{let}_\delta$ and c-op

Here, we present two very useful macros: $\mathbf{let}_\delta$ and **c-op**.

$\mathbf{let}_\delta$, an implementor's convenience macro, lexically binds u to each argument's current value in the substitution and u_δ to each associated domain, respectively.

```
(define-syntax let_δ
  (syntax-rules (:)
    ((_ (s d) ((u : u_δ) ...) body)
     (let ((u (walk u s)) ...)
       (let ((u_δ (cond
                    ((var? u) (get_δ u d))
                    (else (make_δ '(,u)))))
         ...)
         body)))))
```

c-op is a macro for defining constraint operations that uses $\mathbf{let}_\delta$. $\mathbf{build}_{\mathsf{oc}}$ (page 15) creates a storable representation of the constraint, and the constraint store is extended. The body is run *when* each argument u has a domain.

```
(define-syntax c-op
  (syntax-rules (:)
    ((_ op ((u : u_δ) ...) body)
     (λ_M (a : s d c)
       (let_δ (s d) ((u : u_δ) ...)
         (let* ((c (ext-c (build_oc op u ...) c))
                (a (make-a s d c)))
           (cond
             ((and u_δ ...) (body a))
             (else a))))))))
```

3.4.1 The Goal Constructors

There are five constraint constructors: dom^{fd}, $\leqslant^{fd}$, $+^{fd}$, $\not\equiv^{fd}$, and $all\text{-}diff^{fd}$.

- The goal $(dom^{fd}\ x\ n^*)$ constrains $x \in n^*$. The real work of dom^{fd} is done by $process_\delta$ (page 9).

```
(define dom^fd
  (λ (x n*)
    (goal-construct (dom^fd_c x n*))))

(define dom^fd_c
  (λ (x n*)
    (λ_M (a : s d c)
      ((process_δ (walk x s) (make_δ n*)) a))))
```

- The goal $(\leqslant^{fd}\ u\ v)$ implies that $\min(u) \leqslant \max(v)$, filtering out ineligible domain elements from u and v.

```
(define ⩽^fd
  (λ (u v)
    (goal-construct (⩽^fd_c u v))))


(define ⩽^fd_c
  (λ (u v)
    (c-op ⩽^fd_c ((u : u_δ) (v : v_δ))
      (let ((u_min (min_δ u_δ))
            (v_max (max_δ v_δ)))
        (compose_M
          (process_δ u
            (copy-before (λ (u) (< v_max u)) u_δ))
          (process_δ v
            (drop-before (λ (v) (⩽ u_min v)) v_δ))))))))
```

- The goal $(+^{fd}\ u\ v\ w)$ has the meaning $u + v = w$. Thus we get two relations:

$$\min(u) + \min(v) \leqslant \max(w)$$
$$\min(w) \leqslant \max(u) + \max(v)$$

```
(define +^fd
  (λ (u v w)
    (goal-construct (+^fd_c u v w))))


(define +^fd_c
  (λ (u v w)
    (c-op +^fd_c ((u : u_δ) (v : v_δ) (w : w_δ))
      (let ((u_min (min_δ u_δ)) (u_max (max_δ u_δ))
            (v_min (min_δ v_δ)) (v_max (max_δ v_δ))
            (w_min (min_δ w_δ)) (w_max (max_δ w_δ)))
        (compose_M
          (process_δ w
            (range (+ u_min v_min) (+ u_max v_max)))
          (compose_M
            (process_δ u
              (range
                (- w_min v_max) (- w_max v_min)))
            (process_δ v
              (range
                (- w_min u_max) (- w_max u_min)))))))))
```

- The goal $(\not\equiv^{fd}\ u\ v)$ constrains u and v from having the same values. If the domains of u and v are disjoint, the constraint can never be violated and so it is ignored. Otherwise, the constraint is kept around until the domains are equal (which violates the constraint, returning false) or one domain is reduced to a singleton. Then the single element is removed from the other argument's domain and the constraint is dropped.

```
(define ≢^fd
  (λ (u v)
    (goal-construct (≢^fd_c u v))))


(define ≢^fd_c
  (λ (u v)
    (λ_M (a : s d c)
      (let_δ (s d) ((u : u_δ) (v : v_δ))
        (cond
          ((or (not u_δ) (not v_δ))
           (make-a s d (ext-c (build_oc ≢^fd_c u v) c)))
          ((and (singleton?_δ u_δ)
                (singleton?_δ v_δ)
                (= (singleton-element_δ u_δ)
                   (singleton-element_δ v_δ)))
           #f)
          ((disjoint?_δ u_δ v_δ) a)
          (else
           (let* ((ĉ (ext-c (build_oc ≢^fd_c u v) c))
                  (a (make-a s d ĉ)))
             (cond
               ((singleton?_δ u_δ)
                ((process_δ v (diff_δ v_δ u_δ)) a))
               ((singleton?_δ v_δ)
                ((process_δ u (diff_δ u_δ v_δ)) a))
               (else a)))))))))
```

- The goal ($all\text{-}diff^{fd}$ v^*) ensures all variables and values in v^* differ. $all\text{-}diff^{fd}$ could have been written as a derived constraint of $\not\equiv^{fd}$. Instead, this constraint is between a list of values (either variables or constants), where the constants are recursively excluded from each variable's domain or the argument to $all\text{-}diff^{fd}$ might be a single variable, which eventually becomes such a list of values.

```
(define all-diff^fd
  (λ (v*)
    (goal-construct (all-diff_c^fd v*))))
```

In the first of $all\text{-}diff_c^{fd}$'s two **cond** lines, we acknowledge that we have a variable (eventually to become a list) and we include in c a constraint which means "Keep trying until the variable has an association in the substitution." In the second **cond** line, we must have a list, so we partition the list into two pieces: unresolved variables and the found values, making sure along the way that the found values are all different. If they are not different, $all\text{-}diff_c^{fd}$ fails.

```
(define all-diff_c^fd
  (λ (v*)
    (λ_M (a : s d c)
      (let ((v* (walk v* s)))
        (cond
          ((var? v*)
           (let* ((oc (build_oc all-diff_c^fd v*)))
             (make-a s d (ext-c oc c))))
          (else
           (let-values (((x* n*) (partition var? v*)))
             (let ((n* (list-sort < n*)))
               (cond
                 ((list-sorted? < n*)
                  ((all-diff/_c^fd x* n*) a))
                 (else #f)))))))))))
```

In $all\text{-}diff/_c^{fd}$, we are dealing with the unresolved variables y^* and the found values n^*. We move a variable y of y^* to x^* unless y is associated with a value in the substitution, presumably to a single valid value. Then, we don't move y to x^* and instead move y to n^*, which must stay sorted. When each variable has been processed, we *exclude* the final n^* *from* the possible choices remaining from each of the domains of x^*. $exclude\text{-}from_\delta$ (defined below) calls $process_\delta$ to refine d, and potentially call *run-constraints*.

```
(define all-diff/_c^fd
  (λ (y* n*)
    (λ_M (a : s d c)
      (let loop ((y* y*) (n* n*) (x* '()))
        (cond
          ((null? y*)
           (let* ((oc (build_oc all-diff/_c^fd x* n*))
                  (a (make-a s d (ext-c oc c))))
             ((exclude-from_δ (make_δ n*) d x*) a)))
          (else
           (let ((y (walk (car y*) s)))
             (cond
               ((var? y) (loop (cdr y*) n* (cons y x*)))
               ((memv?_δ y n*) #f)
               (else (let ((n* (list-insert < y n*)))
                       (loop (cdr y*) n* x*)))))))))))))
```

```
(define exclude-from_δ
  (λ (δ_1 d x*)
    (let loop ((x* x*))
      (cond
        ((null? x*) identity_M)
        ((get_δ (car x*) d)
         ⇒ (λ (δ_2)
             (compose_M
               (process_δ (car x*) (diff_δ δ_2 δ_1))
               (loop (cdr x*)))))
        (else (loop (cdr x*)))))))
```

3.5 Disequality Constraints Implementation

The version of disequality defined on finite domains is not powerful enough to operate on lists, even when those lists contain finite domain values and variables alone. So, we describe a more general version of disequality constraints.

This implementation uses unification to uncover exactly which associations must never be made. For example, the constraint ($\not\equiv$ (x 1) (2 y)) means that x cannot be 2 when y is 1. Conveniently, unification of (x 1) and (2 y) produces exactly the normalized associations that should never be true simultaneously: ((x . 2) (y . 1)). This idea has been formalized by Hubert Comon (1991), who has described this and other approaches to disequality.

3.5.1 The Disequality Constraints Parameters

We define the same constraint parameters as in Section 3.3.1; however, the disequality constraint parameters are much simpler than those of finite domains.

$process\text{-}prefix_{NEQ}$ is quite simple, as no domain information is stored. The constraints of every variable within the prefix p are reexamined using *run-constraints*.

```
(define process-prefix_NEQ
  (λ (p c)
    (run-constraints (recover/vars p) c)))
```

Next we consider $enforce\text{-}constraints_{NEQ}$. Disequality constraints fail immediately if they are unsatisfiable. It is not necessary to check the store again before reification, so *enforce-constraints* is merely $unit_G$.

```
(define enforce-constraints_NEQ (λ (x) unit_G))
```

$reify\text{-}constraints_{NEQ}$ reifies every relevant constraint in c, since they are included as part of the reified value; we have arbitrarily chosen the colon ':' to separate the reified value from the list of reified constraints.

```
(define reify-constraints_NEQ
  (λ (m r)
    (λ_G (a : s d c)
      (let* ((c (walk* c r))
             (p* (remp any/var? (map oc→prefix c))))
        (cond
          ((null? p*) m)
          (else `(,m : . ((≢ . ,p*)))))))))
```

We invoke (use_{NEQ}) in order to use disequality constraints over term trees,

```
(define use_NEQ
  (λ ()
    (process-prefix process-prefix_NEQ)
    (enforce-constraints enforce-constraints_NEQ)
    (reify-constraints reify-constraints_NEQ)))
```

3.5.2 The Disequality Goal

We define $\not\equiv$ for disequality, which takes two arguments that must be different. If not, the goal fails.

```
(define ≠
  (λ (u v)
    (goal-construct (≠c u v))))

(define ≠c
  (λ (u v)
    (λM (a : s d c)
      (cond
        ((unify '((,u . ,v)) s)
         ⇒ (λ (ŝ) ((≠c^neq (prefix-s s ŝ)) a)))
        (else a)))))
```

$\not\equiv_{\mathsf{c}}^{neq}$ operates on a substitution p. If every association in p is in s, the disequality constraint has been violated. Otherwise, we get a new non-empty prefix to be stored. When the initial substitution is empty, $\hat{s}$ is the prefix.

```
(define ≠c^neq
  (λ (p)
    (λM (a : s d c)
      (cond
        ((unify p s)
         ⇒ (λ (ŝ)
             (let ((p (prefix-s s ŝ)))
               (cond
                 ((null? p) #f)
                 (else ((normalize-store p) a))))))
        (else a)))))
```

If *unify* does not return false, there is still a possibility the disequality constraint can be violated. Now $\hat{s}$ contains the prefix p that must *not* become realized.

normalize-store removes any superfluous disequality constraints in c. If p is subsumed by any prefix in c, or if p subsumes any prefix in c, there is redundancy. After the constraint store has been examined, either a new package with potentially fewer constraints or the original package with at most one more constraint is returned.

```
(define normalize-store
  (λ (p)
    (λM (a : s d c)
      (let loop ((c c) (ĉ '()))
        (cond
          ((null? c)
           (let ((ĉ (ext-c (build_oc ≠c^neq p) ĉ)))
             (make-a s d ĉ)))
          ((eq? (oc→rator (car c)) '≠c^neq)
           (let* ((oc (car c))
                  (p̂ (oc→prefix oc)))
             (cond
               ((subsumes? p̂ p) a)
               ((subsumes? p p̂) (loop (cdr c) ĉ))
               (else (loop (cdr c) (cons oc ĉ))))))
          (else (loop (cdr c) (cons (car c) ĉ))))))))
```

The substitution p subsumes (page 17) s if unifying p in the substitution s does not extend s. If any prefix in c subsumes p, then c is not extended. Furthermore, because p will be part of the new c, those prefixes in c that p subsumes are dropped.

4. Solutions to Exercises

Recall from Section 2.1.3 that we left unsolved three problems: Send More Money, N-Queens, and *rember*^{o}. We start with the solution to Send More Money.

4.1 *send-more-money*^{o} : Solution

In *send-more-money*^{o} below, we must be certain that each letter in the puzzle is associated with a different value. Since we know that s and m cannot be 0 (the leading digit of a natural number is never 0), their domains start at 1. The domains of the remaining *letters* include 0. There are also three carry variables. (There should have been four, one for each column, but we know that m must be 1, since the sum of two digits, even with a *carry-in* of 1, cannot exceed 19.) Thus, we use m as the fourth *carry* variable.). The rest of *send-more-money*^{o} does the long addition.

add-digits^o performs one step of long addition, adding two digits together considering a possible *carry-in*, and returning the sum with a possible *carry-out*.

```
(define send-more-money^o
  (λ (letters)
    (fresh (s e n d m o r y carry0 carry1 carry2)
      (≡ letters '(,s ,e ,n ,d ,m ,o ,r ,y))
      (all-diff^fd letters)
      (in^fd s m (range 1 9))
      (in^fd e n d o r y (range 0 9))
      (in^fd carry0 carry1 carry2 (range 0 1))
      (add-digits^o s m carry2 m o)
      (add-digits^o e o carry1 carry2 n)
      (add-digits^o n r carry0 carry1 e)
      (add-digits^o d e 0 carry0 y))))

(define add-digits^o
  (λ (augend addend carry-in carry-out digit)
    (fresh (partial-sum sum)
      (in^fd partial-sum (range 0 18))
      (in^fd sum (range 0 19))
      (+^fd augend addend partial-sum)
      (+^fd partial-sum carry-in sum)
      (cond^e
        ((<^fd 9 sum) (≡ carry-out 1) (+^fd digit 10 sum))
        ((≤^fd sum 9) (≡ carry-out 0) (≡ digit sum))))))
```

```
(run* (q) (send-more-money^o q)) ⇒ ((9 5 6 7 1 0 8 2))
```

This answer corresponds to the variable assignments:

$$S = 9 \quad E = 5 \quad N = 6 \quad D = 7$$
$$M = 1 \quad O = 0 \quad R = 8 \quad Y = 2$$

which indeed satisfy the Send More Money' puzzle.

There are two interesting facets to this code. First, careful inspection of *send-more-money*^{o} and *add-digits*^{o} reveals that the solution requires no explicit recursion. Second, the goals of *send-more-money*^{o} are ordered so that the first and second arguments to the four goals using *add-digits*^{o} when read from top to bottom spell $s\ e\ n\ d$ and $m\ o\ r\ e$, respectively.

4.2 *n-queens*^{o} : Solution

Here we solve N-Queens puzzle. Recall that in chess, a queen is attacking a piece if they are on the same diagonal, the same row, or the same column, with no interference.

We define *n-queens*^{o} recursively. A loop executes n times, creating n variables, one for each row. When the loop reaches its base case, the list l contains the newly initialized variables. Each variable is in a different position in the list l (since no two queens can occupy the same row); the values that will be associated with these variables must be all different (since no two queens can occupy the same column); and no two queens can share the same diagonal. *all-diff*^{fd} and *diagonals*^{o} enforce these last two restrictions.

```
(define n-queens^o
  (λ (q* n)
    (let loop ((i n) (l '()))
      (cond
        ((zero? i)
         (fresh ()
           (all-diff^fd l)
           (diagonals^o n l)
           (≡ q* l)))
        (else (fresh (x)
                (in^fd x (range 1 n))
                (loop (- i 1) (cons x l))))))))

(define diagonals^o
  (λ (n r)
    (let loop ((r r) (i 0) (s (cdr r)) (j 1))
      (cond
        ((or (null? r) (null? (cdr r))) #s)
        ((null? s) (loop (cdr r) (+ i 1) (cddr r) (+ i 2)))
        (else
         (let ((q_i (car r)) (q_j (car s)))
           (fresh ()
             (diag^o q_i q_j (- j i) (range 0 (* 2 n)))
             (loop r i (cdr s) (+ j 1)))))))))
```

In *diagonals*^{o}, i keeps track of the position in the list of queens that we are on and j keeps track of the position in the list of queens somewhere to the right of i. q_i and q_j are two elements of n. Effectively, we check every possible position for two queens exactly once, as q_i will always be less than q_j. For each combination, *diag*^{o} is called.

As long as the following equations hold (Schrijvers et al. 2009), the queens will not be attacking each other and *diag*^{o} will succeed.

$$q_i + d \neq q_j$$

$$q_j + d \neq q_i$$

```
(define diag^o
  (λ (q_i q_j d rng)
    (fresh (q_i+d q_j+d)
      (in^fd q_i+d q_j+d rng)
      (+^fd q_i d q_i+d)
      (≠^fd q_i+d q_j)
      (+^fd q_j d q_j+d)
      (≠^fd q_j+d q_i))))
```

The following expression produces the solution when $n = 8$. The **run*** expression evaluates to the list of 92 solutions.

```
(length (run* (q) (n-queens^o q 8))) ⇒ 92
```

4.3 *rember*^{o} : Solution

Recall that *rember*^{o} does not give us the results we were expecting. A simple fix is that once we have found an a, we demand that we won't get any more answers from that a as is shown in the last line of the definition of *rember*^{o}, below. This concept becomes particularly important when using a representation of environments where shadowing exists, say, in a type inferencer being used as a type inhabiter for demonstrating the Curry-Howard Isomorphism.

```
(define rember^o
  (λ (x ls out)
    (cond^e
      ((≡ '() ls) (≡ '() out))
      ((fresh (a d res)
         (≡ `(,a . ,d) ls)
         (rember^o x d res)
         (cond^e
           ((≡ a x) (≡ res out))
           ((≠ a x) (≡ `(,a . ,res) out))))))))

(run* (q) (rember^o 'a '(a b a c) q)) ⇒ ((b c))

(run* (q) (rember^o 'a '(a b c) '(a b c))) ⇒ ()
```

5. Composition

In order to compose different kinds of constraints, it is enough to redefine each parameter according to the semantics desired by the user. For instance, we can combine the finite domain and disequality constraints by allowing $\neq$ to call $\neq^{fd}$ in the case of natural number arguments, or using domain information to eliminate parts of the prefix in $\neq^{neq}_{c}$. The composite library imports the function definitions from the finite domain and disequality libraries, and defines the composite parameters in terms of the domain-specific functions.

With the composite library, examples such as the following can be solved.

```
(run* (q)
  (in^fd q (2 3 4))
  (all-diff^o `(apple 3 ,q))) ⇒ (2 4)
```

Although the two kinds of constraints do not explicitly communicate, q's domain information can be used in conjunction with *all-diff*^{o}. Before reification, q will be unified with all remaining values in its domain. q's only realizable values are 2 and 4, since *all-diff*^{o} fails when 3 is tried. (Using *all-diff*^{fd} instead of *all-diff*^{o} would result in an error, since apple is a symbol rather than a natural number.)

Conflicting constraints cannot be present in the same call to **run***. Having two kinds of constraints in any environment would cause conflicting parameter definitions. For example, assume we wish to verify that the 92 answers returned by (**run*** (q) (*n-queens*^{o} q 8)) are actually all unique answers. We cannot use *all-diff*^{fd}, since the result is a list of *lists*, but finite domain constraints only work on a list of *natural numbers*, so, we must use *all-diff*^{o}.

```
(define answers (run* (q) (n-queens^o q 8)))

(run* (q) (all-diff^o answers)) ⇒ (_0)
```

These calls to **run*** leave no opportunity to redefine parameters, so we must be sure that the definitions for all parameters work correctly when either finite domain or disequality constraints are used.

Combining different kinds of constraints that use different domain representations is more complex. Conveniently, the user has the power to redefine, extend, or ignore any part of the existing libraries when making a composition. A tagged domain store could be implemented simply by redefining

get$_\delta$ and *ext-d*. Basic parts of the implementation framework, like *reify*, are general and powerful enough to handle a new kind of constraint, provided there is a *reify-constraints* parameter to determine what should be returned.

Each library should have a function devoted to defining the parameters properly. For finite domains it is use_{FD}, for disequality constraints it is use_{NEQ}, and for their composition it might be use_{FDNEQ}. Thus, it is possible to import libraries without worrying about the values of the parameters: simply invoke the correct thunk and the definitions will be there.

6. The miniKanren Philosophy

Our development of cKanren has been informed by a design and implementation philosophy that arose from our work on miniKanren. The central tenet of this philosophy is that in a purely declarative miniKanren relation, the order of goals is unimportant. That is, swapping two conjuncts (or two disjuncts) should not affect the semantics of the program. This is true only to a point: a miniKanren query that has no answers may diverge instead of failing in finite time. For a query that produces answers, however, reordering subgoals should not affect the set of possible answers returned.[4]

6.0.1 Design Philosophy

A consequence of our philosophy is that the programmer can specify constraints in any order. For example, the finite domain constraints presented in Section 2.1.1 can be applied to a variable, even before the variable has been associated with a finite domain or has been bound in the substitution.

As pointed out on page 8, provided a variable is associated with either a domain or a natural number before reification, the program works correctly. Another consequence of the miniKanren philosophy is that "extra-logical" operators, such as Prolog's `var/1`, `is`, or "cut" (`!`) operators, are not allowed in user-level code. Although not enforced by cKanren, we hope that implementors of additional constraint libraries will adhere to this philosophy.

As a result, our solution to Send More Money in Section 4.1 does not make assumptions about when variables become associated with domains or values. Although a less pure implementation of Send More Money might run faster, we are convinced that the benefits of declarativeness are too important to abandon.

6.0.2 Implementation Philosophy

The miniKanren philosophy also guides our implementation of cKanren. For example, we decided to store a closure indicating progress along with a first-order representation of constraints in the constraint store. Keeping only a closure would have severely reduced our ability to recover information from the constraint store. A user trying to debug would not have been able to look at a textual version of the constraints. This would make it impossible to see which constraints were actually being stored. Also, an implementor trying to reify the constraints would not have been able to include meaningful information about the constraints on a variable. Instead, we have included both a symbol representing the constraint and its operands, thus giving us the best of both worlds.

cKanren's constraint store is a list of operator constraints. Another way to design the constraint store would be to associate a variable with its relevant constraints. However, a constraint would then appear once for each variable it references, resulting in redundancy and complicating the fixpoint algorithm. With our cKanren framework we can decide whether a constraint is still relevant with a single call to *memq*; the alternative approach would require more lookups to determine the same information.

We had originally implemented cKanren using attributed variables (Holzbaur 1992). An attributed variable is one which calls a user-defined customization point (a "hook") when unified, either with a value or another variable. A key reason we have avoided attributed variables is the change required to unification. Attribute hooks are called whenever an attributed variable is unified with anything, requiring the ability to call cKanren code from $\equiv$, possibly leading to further unifications and backtracking. Although Prolog systems with attributed variables do allow backtracking from attribute hooks, we have chosen to not use attributed variables to simplify cKanren's implementation.

7. Conclusion

The field of Constraint Logic Programming is well developed and its history, along with the more general area of Constraint Programming, is described in (Marriott and Stuckey 1998) and (Apt 2003). Extensions to Prolog for Constraint Logic Programming, such as Prolog II (Colmerauer 1985), presented disequality constraints over trees. The first theoretical work on *CLP(X)*, for different Xs, appears in Jaffar and Lassez (1987).

The CLP language CHIP (Dincbas et al. 1988; Van Hentenryck 1989) introduced finite domain constraints. Jaffar (1992) described constraints over real numbers (*CLP(R)*). CHIP led to ECL^{i}PS^e (Wallace et al. 1997). In the theory of *CLP(X)*, the substitution store associated with logic programming becomes a set of constraints (Jaffar and Lassez 1987). Jaffar and Maher (1994) survey the development of Constraint Logic Programming.

We have presented cKanren, a simple, concise, and easily extendable framework for constraint logic programming in Scheme. cKanren allows programmers to use and define constraint libraries that extend miniKanren, which itself embeds logic programming in Scheme. We have also presented the implementation of two libraries of constraints: one over finite domains, and the other over tree terms. We hope others will create cKanren libraries of their own, increasing the expressive power of miniKanren.

Acknowledgments

We are grateful to our readers: Adam Foltzer, Lindsey Kuper, Karissa McKelvey, David Nolen, Zach Sparks, and Cameron Swords. The early work on miniKanren would not have been successful without the insights of Mitchell Wand and Steve Ganz. Oleg Kiselyov made major improvements to every aspect of miniKanren, and we are grateful to Chung-Chieh Shan for his work on the final implementation of miniKanren. We continue to appreciate LaTeX and its creator, Dorai Sitaram.

[4] See (Byrd 2009) for a detailed discussion of these issues and of the miniKanren design philosophy.

References

Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001. URL `citeseer.ist.psu.edu/baader99unification.html`.

William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, 2009.

William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In Robby Findler, editor, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago Technical Report TR-2006-06, pages 105–117, 2006.

William E. Byrd and Daniel P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Universite Laval Technical Report DIUL-RT-0701*, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

Alain Colmerauer. Prolog in 10 figures. *Commun. ACM*, 28 (12):1296–1310, 1985. ISSN 0001-0782.

Hubert Comon. Disunification: A survey. In *Computational Logic – Essays in Honor of Alan Robinson*, pages 322–359, 1991. URL `citeseer.ist.psu.edu/comon91disunification.html`.

Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Franoise Berthier. The constraint logic programming language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

Marc Feeley. SRFI 39: Parameter objects. `http://srfi.schemers.org/srfi-39/`, 2002.

Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, Montreal, Canada, September 18–21, 2000*, pages 186–197. ACM Press, 2000.

Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268, London, UK, 1992. Springer-Verlag. ISBN 3-540-55844-6. URL `http://portal.acm.org/citation.cfm?id=646448.692447`.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program*, 19/20:503–581, 1994.

Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Trans. Program. Lang. Syst.*, 14:339–395, May 1992. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/129393.129398`.

David B. MacQueen, Philip Wadler, and Walid Taha. How to add laziness to a strict language without even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, September 1998. Baltimore, MD.

Kim Marriott and Peter J. Stuckey. *Programming with Constraints. An Introduction*. The MIT Press, 1998.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *ACM Computing Surveys*, 19 (6):663–697, 2009.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. URL `http://www.r6rs.org/`.

Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud, editor, *Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Nancy, France, September 16–19, 1985. Springer-Verlag.

Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January, 1992. ACM Press.

Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, May 1997.

A. cKanren Implementation Helpers

```
(define process-prefix (make-parameter 'dummy))
(define enforce-constraints (make-parameter 'dummy))
(define reify-constraints (make-parameter 'dummy))
(define empty-d '())
(define empty-c '())
(define empty-a (make-a empty-s empty-d empty-c))
(define ext-d (λ (x fd d) (cons `(,x . ,fd) d)))
(define make_δ (λ (n*) n*))
(define oc→proc (λ (oc) (car oc)))
(define oc→rator (λ (oc) (car (cdr oc))))
(define oc→rands (λ (oc) (cdr (cdr oc))))
(define oc→prefix (λ (oc) (car (oc→rands oc))))

(define-syntax build_oc
  (syntax-rules ()
    ((_ op arg ...)
     (build-aux_oc op (arg ...) () (arg ...)))))

(define-syntax build-aux_oc
  (syntax-rules ()
    ((_ op () (z ...) (arg ...))
     (let ((z arg) ...) `(,(op z ...) . (op ,z ...))))
    ((_ op (arg0 arg ...) (z ...) args)
     (build-aux_oc op (arg ...) (z ... q) args))))
```

```
(define list-sorted?
  (λ (pred ls)
    (cond
      ((or (null? ls) (null? (cdr ls))) #t)
      ((pred (car ls) (cadr ls)) (list-sorted? pred (cdr ls)))
      (else #f))))

(define list-insert
  (λ (pred x ls)
    (cond
      ((null? ls) (cons x '()))
      ((pred x (car ls)) (cons x ls))
      (else (cons (car ls) (list-insert pred x (cdr ls)))))))

(define copy-before
  (λ (pred δ)
    (cond
      ((null? δ) '())
      ((pred (car δ)) '())
      (else (cons (car δ) (copy-before pred (cdr δ)))))))

(define drop-before
  (λ (pred δ)
    (cond
      ((null? δ) '())
      ((pred (car δ)) δ)
      (else (drop-before pred (cdr δ))))))

(define map-sum
  (λ (f)
    (letrec
      ((loop
         (λ (ls)
           (cond
             ((null? ls) #u)
             (else
              (cond^e
                ((f (car ls)))
                ((loop (cdr ls)))))))))
      loop)))
```

get_δ looks up a variable's current domain in d. If a variable does not currently have a domain, this function returns false. In order to distinguish between variables without domains, and values that can never have domains (such as `#t` or negative numbers), the argument to get_δ must be a variable.

```
(define get_δ
  (λ (x d)
    (cond
      ((assq x d) ⇒ rhs)
      (else #f))))
```

Applying conventional operations to domains is efficient when the lists representing the domains are sorted.

```
(define value?_δ (λ (v) (and (integer? v) (⩽ 0 v))))
(define memv?_δ (λ (v δ) (and (value?_δ v) (memv v δ))))
(define null?_δ (λ (δ) (null? δ)))
(define singleton?_δ (λ (δ) (null? (cdr δ))))
(define singleton-element_δ (λ (δ) (car δ)))
(define min_δ (λ (δ) (car δ)))

(define max_δ
  (λ (δ)
    (cond
      ((null? (cdr δ)) (car δ))
      (else (max_δ (cdr δ))))))
```

```
(define disjoint?_δ
  (λ (δ_1 δ_2)
    (cond
      ((or (null? δ_1) (null? δ_2)) #t)
      ((= (car δ_1) (car δ_2)) #f)
      ((< (car δ_1) (car δ_2))
       (disjoint?_δ (cdr δ_1) δ_2))
      (else (disjoint?_δ δ_1 (cdr δ_2))))))

(define diff_δ
  (λ (δ_1 δ_2)
    (cond
      ((or (null? δ_1) (null? δ_2)) δ_1)
      ((= (car δ_1) (car δ_2)) (diff_δ (cdr δ_1) (cdr δ_2)))
      ((< (car δ_1) (car δ_2))
       (cons (car δ_1) (diff_δ (cdr δ_1) δ_2)))
      (else (diff_δ δ_1 (cdr δ_2))))))

(define intersection_δ
  (λ (δ_1 δ_2)
    (cond
      ((or (null? δ_1) (null? δ_2)) '())
      ((= (car δ_1) (car δ_2))
       (cons (car δ_1)
         (intersection_δ (cdr δ_1) (cdr δ_2))))
      ((< (car δ_1) (car δ_2))
       (intersection_δ (cdr δ_1) δ_2))
      (else (intersection_δ δ_1 (cdr δ_2))))))
```

Since x^* is a list of variables, which is constructed using *var*, which is itself, constructed using *vector*, we must use *memq*, when checking for membership in the list. In a purely functional setting, we would need to build variables in a different way, probably relying on a monotonically increasing non-negative integer variable.

```
(define any/var?
  (λ (t)
    (cond
      ((var? t) #t)
      ((pair? t)
       (or (any/var? (car t)) (any/var? (cdr t))))
      (else #f))))

(define any-relevant/var?
  (λ (t x*)
    (cond
      ((var? t) (memq t x*))
      ((pair? t) (or (any-relevant/var? (car t) x*)
                     (any-relevant/var? (cdr t) x*)))
      (else #f))))

(define recover/vars
  (λ (p)
    (cond
      ((null? p) '())
      (else
        (let ((x (lhs (car p)))
              (v (rhs (car p)))
              (r (recover/vars (cdr p))))
          (cond
            ((var? v) (ext/vars v (ext/vars x r)))
            (else (ext/vars x r)))))))))

(define ext/vars
  (λ (x r)
    (cond
      ((memq x r) r)
      (else (cons x r)))))
```

```
(define verify-all-bound
  (λ (c bound-x*)
    (unless (null? c)
      (cond
        ((find (λ (x) (not (memq x bound-x*)))
               (filter var? (oc→rands (car c))))
         ⇒ (λ (x)
             (error 'verify-all-bound
               "Constrained variable ~s without domain"
               x)))
        (else (verify-all-bound (cdr c) bound-x*))))))

(define subsumes?
  (λ (p s)
    (cond
      ((unify p s)
       ⇒ (λ (ŝ) (eq? s ŝ)))
      (else #f))))
```

B. miniKanren Implementation

Our miniKanren implementation comprises three kinds of operators: the interface operator **run**; goal constructors $\equiv$, **cond**^{e}, and **fresh**, which take a package *implicitly*; and functions such as $\equiv_{\mathsf{c}}$ (page 7), which take a package *explicitly*.

A goal g is a function that maps a package a to an ordered sequence a^{∞} of zero or more packages. (For clarity, we notate λ as λ_{G} when creating such a function g.)

```
(define-syntax λG
  (syntax-rules (:)
    ((_ (a : s d c) body)
     (λ (a)
       (let ((s (car a)) (d (cadr a)) (c (cddr a)))
         body)))
    ((_ (a) body) (λ (a) body))))
```

Because a sequence of packages may be infinite, we represent it not as a list but as an a^{∞}, a special kind of stream that can contain either zero, one, or more packages (Hinze 2000; Wadler 1985). We use #f to represent the empty stream of packages. If a is a package, then a itself represents the stream containing just a.

```
(define mzero_G (λ () #f))
(define unit_G (λG (a) a))
(define choice_G (λ (a f) (cons a f)))
```

To represent a stream containing multiple packages, we use ($choice_{\mathsf{G}}$ a f), where a is the first package in the stream, and where f is a thunk that, when invoked, produces the remainder of the stream. (For clarity, we notate λ as λ_{F} when creating such a function f.) To represent an incomplete stream, we use (**inc** e), where e is an *expression* that evaluates to an a^{∞}—thus **inc** creates an f.

```
(define-syntax λF
  (syntax-rules () ((_ () e) (λ () e))))

(define-syntax inc
  (syntax-rules () ((_ e) (λF () e))))

(define empty-f (λF () (mzero_G)))
```

A singleton stream a is the same as ($choice_{\mathsf{G}}$ a *empty-f*). For goals that return only a single package, however, using this special representation of a singleton stream avoids the cost of unnecessarily building and taking apart pairs, and creating and invoking thunks.

To ensure that the values produced by these four kinds of a^{∞}'s can be distinguished, we assume that a package is never #f, a function, or a pair whose *cdr* is a function. To discriminate among these four cases, we define **case**$^{\infty}$.

```
(define-syntax case∞
  (syntax-rules ()
    ((_ e (() e0) ((f̂) e1) ((â) e2) ((a f) e3))
     (let ((a∞ e))
       (cond
         ((not a∞) e0)
         ((procedure? a∞) (let ((f̂ a∞)) e1))
         ((not (and (pair? a∞)
                    (procedure? (cdr a∞))))
          (let ((â a∞)) e2))
         (else (let ((a (car a∞)) (f (cdr a∞)))
                 e3)))))))
```

If the first argument to *take* is #f, then *take* returns the entire stream of reified values as a list, thereby providing the behavior of **run***. The **and** expressions within *take* detect this #f case.

```
(define take
  (λ (n f)
    (cond
      ((and n (zero? n)) '())
      (else
       (case∞ (f)
         (() '())
         ((f) (take n f))
         ((a) (cons a '()))
         ((a f) (cons a (take (and n (- n 1)) f))))))))
```

The interface operator **run** uses *take* to convert an f to an *even* stream (MacQueen et al. 1998). The definition of **run** places an artificial goal at the tail of g_0 g ... This artificial goal invokes *reify* (Section 3.2.4) on the variable x using the final package a produced by running all the goals in the empty package *empty-a* (page 15).

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (λF ()
         ((fresh (x) g0 g ... (reify x))
          empty-a))))))

(define-syntax run*
  (syntax-rules ()
    ((_ (x) g0 g ...) (run #f (x) g0 g ...))))
```

B.1 Goal Constructors

The simplest goal constructors are those expanded from *goal-construct* (page 7); the goals they create return either a singleton stream or an empty stream. To take the conjunction of goals, we define **fresh**, a goal constructor that first lexically binds variables built by *var* and then combines successive goals using **bind**$^{*}_{\mathsf{G}}$.

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (a)
       (inc
         (let ((x (var 'x)) ...)
           (bind* G (g0 a) g ...)))))))
```

bind$^*_{\mathsf{G}}$ is short-circuiting: since the empty stream is represented by **#f**. **bind$^*_{\mathsf{G}}$** relies on *bind*$_{\mathsf{G}}$ (Moggi 1991; Wadler 1992), which applies the goal g to each element in the stream a^∞. The resulting a^∞'s are then merged using *mplus*$_{\mathsf{G}}$, which combines an a^∞ and an f to yield a single a^∞.

```
(define-syntax bind*_G
  (syntax-rules ()
    ((_ e) e)
    ((_ e g0 g ...) (bind*_G (bind_G e g0) g ...))))

(define bind_G
  (λ (a∞ g)
    (case∞ a∞
      (() (mzero_G))
      ((f) (inc (bind_G (f) g)))
      ((a) (g a))
      ((a f) (mplus_G (g a) (λ_F () (bind_G (f) g)))))))

(define mplus_G
  (λ (a∞ f)
    (case∞ a∞
      (() (f))
      ((f̂) (inc (mplus_G (f) f̂)))
      ((a) (choice_G a f))
      ((a f̂) (choice_G a (λ_F () (mplus_G (f) f̂)))))))
```

To take the disjunction of goals we define **cond^e**, a goal constructor that combines successive **cond^e** lines using **mplus$^*_{\mathsf{G}}$**, which in turn relies on *mplus*$_{\mathsf{G}}$. We use the same implicit package a for each **cond^e** line. To avoid unwanted divergence, we treat the **cond^e** lines as a single **inc** stream.

```
(define-syntax cond^e
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λ_G (a)
       (inc (mplus*_G (bind*_G (g0 a) g ...)
                      (bind*_G (g1 a) ĝ ...)
                      ...))))))

(define-syntax mplus*_G
  (syntax-rules ()
    ((_ e) e)
    ((_ e0 e ...) (mplus_G e0 (λ_F () (mplus*_G e ...))))))
```

B.2 miniKanren Helpers

```
(define var (λ (dummy) (vector dummy)))
(define var? (λ (x) (vector? x)))
(define empty-s '())
(define ext-s (λ (x v s) (cons `(,x . ,v) s)))
(define lhs (λ (pr) (car pr)))
(define rhs (λ (pr) (cdr pr)))

(define walk
  (λ (u s)
    (cond
      ((not (var? u)) u)
      ((assq u s) ⇒ (λ (pr) (walk (rhs pr) s)))
      (else u))))

(define walk*
  (λ (w s)
    (let ((v (walk w s)))
      (cond
        ((var? v) v)
        ((pair? v)
         (cons (walk* (car v) s) (walk* (cdr v) s)))
        (else v)))))
```

Below is *unify* (Marriott and Stuckey 1998) but with triangular (Baader and Snyder 2001) instead of idempotent substitutions. In the two-pairs case the substitution does not grow; the equation that had a pair in both sides is dropped and replaced by two new equations: one that equates their respective *car*s (done now) and one that equates their respective *cdr*s (done later), so e cannot be empty. Otherwise, there is a recursive call to *unify*, where either zero or one new association is added to the substitution. Of course, unification can fail.

```
(define unify
  (λ (e s)
    (cond
      ((null? e) s)
      (else
       (let loop ((u (caar e)) (v (cdar e)) (e (cdr e)))
         (let ((u (walk u s)) (v (walk v s)))
           (cond
             ((eq? u v) (unify e s))
             ((var? u)
              (and (not (occurs✓ u v s))
                   (unify e (ext-s u v s))))
             ((var? v)
              (and (not (occurs✓ v u s))
                   (unify e (ext-s v u s))))
             ((and (pair? u) (pair? v))
              (loop (car u) (car v)
                `((,(cdr u) . ,(cdr v)) . ,e)))
             ((equal? u v) (unify e s))
             (else #f)))))))))

(define occurs✓
  (λ (x v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (eq-var? v x))
        ((pair? v)
         (or (occurs✓ x (car v) s) (occurs✓ x (cdr v) s)))
        (else #f)))))
```

reify-s is the heart of the reifier. *reify-s* takes an arbitrary value v, and returns a substitution that maps every distinct variable in v to a unique symbol. The trick to maintaining left-to-right ordering of the subscripts on these symbols is to process v from left to right, as can be seen in the *pair?* **cond** line, below. When *reify-s* encounters a variable, it determines if we already have a mapping for that entity. If not, *reify-s* extends the substitution with an association between the variable and a new, appropriately subscripted symbol built using *reify-n*.

```
(define reify-s
  (λ (v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (ext-s v (reify-n (size-s s)) s))
        ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
        (else s)))))

(define reify-n
  (λ (n)
    (string→symbol
      (string-append "_" "." (number→string n)))))

(define size-s (λ (x) (length x)))
```

B.3 Impure Control Operators

For completeness, we define three additional miniKanren goal constructors: **project**, which can be used to access the values of variables, and **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program. The examples from *Thin Ice* of *The Reasoned Schemer* (Friedman et al. 2005) demonstrate how **cond**^{a} and **cond**^{u} can be useful and the pitfalls that await the unsuspecting reader. Also, we have included an additional operator *once*^{o}, defined in terms of **cond**^{u}, which forces the input goal to succeed at most once.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (a : s d c)
       (let ((x (walk* x s)) ...)
         ((fresh () g0 g ...) a))))))

(define-syntax cond^a
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (a)
       (inc (if^a ((g0 a) g ...) ((g1 a) ĝ ...) ...))))))

(define-syntax if^a
  (syntax-rules ()
    ((_) (mzero_G))
    ((_ (e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (() (if^a b ...))
         ((f) (inc (loop (f))))
         ((a) (bind*_G a^∞ g ...))
         ((a f) (bind*_G a^∞ g ...)))))))

(define-syntax cond^u
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (a)
       (inc (if^u ((g0 a) g ...) ((g1 a) ĝ ...) ...))))))

(define-syntax if^u
  (syntax-rules ()
    ((_) (mzero_G))
    ((_ (e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (() (if^u b ...))
         ((f) (inc (loop (f))))
         ((a) (bind*_G a^∞ g ...))
         ((a f) (bind*_G a g ...)))))))

(define once^o (λ (g) (cond^u (g))))
```