# cKanren

## miniKanren with Constraints

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

{calvis, jewillco, kylcarte, webyrd, dfried}@cs.indiana.edu

School of Informatics and Computing
Indiana University, Bloomington

October 23, 2011

# Overview

1. Introduction to Logic Programming/miniKanren
2. Introduction to Constraints
3. Examples
4. Implementation Overview

# miniKanren

Logic programming language extending Scheme

# miniKanren

Logic programming language extending Scheme

Three important operators: $\equiv$, *fresh*, and $cond^e$

# miniKanren

Logic programming language extending Scheme

Three important operators: $\equiv$, *fresh*, and $cond^e$

Intuition:
The *goal* `(== 5 5)` *succeeds* while `(== 5 6)` *fails*

# miniKanren

Logic programming language extending Scheme

Three important operators: $\equiv$, *fresh*, and *cond*$^e$

Intuition:
The *goal* `(== 5 5)` *succeeds* while `(== 5 6)` *fails*

```
(fresh (x)
  (conde
    ((== x 5))
    ((== x 6))))
```

*unifies* $x$ with 5 *or* 6

# miniKanren

Logic programming language extending Scheme

Uses *run* as an interface operator

```
(run1 (y)
  (fresh (x z)
    (== x z)
    (== 3 y)))
```

$\Rightarrow$ `(3)`

# miniKanren

Logic programming language extending Scheme

Uses *run* as an interface operator

```
(run1 (y)
  (fresh (x z)
    (== x z)
    (== 3 y)))
```

$\Rightarrow$ (3)

```
(run1 (y)
  (fresh (x z)
    (== x z)
    (== 3 z)
    (== y x)))
```

$\Rightarrow$ (3)

# miniKanren

Logic programming language extending Scheme

Uses *run* as an interface operator

```
(run1 (y)
  (fresh (x z)
    (== x z)
    (== 3 y)))
```

$\Rightarrow$ (3)

```
(run1 (y)
  (fresh (x z)
    (== x z)
    (== 3 z)
    (== y x)))
```

$\Rightarrow$ (3)

```
(run1 (y)
  (fresh (y)
    (conde
      ((== y 4))
      ((== y 5))))
  (== 3 y))
```

$\Rightarrow$ (3)

# Constraints

Imposing a certain restriction on a variable or set of variables
Find a solution such that every constraint is satisfied

# Constraints

Imposing a certain restriction on a variable or set of variables
Find a solution such that every constraint is satisfied

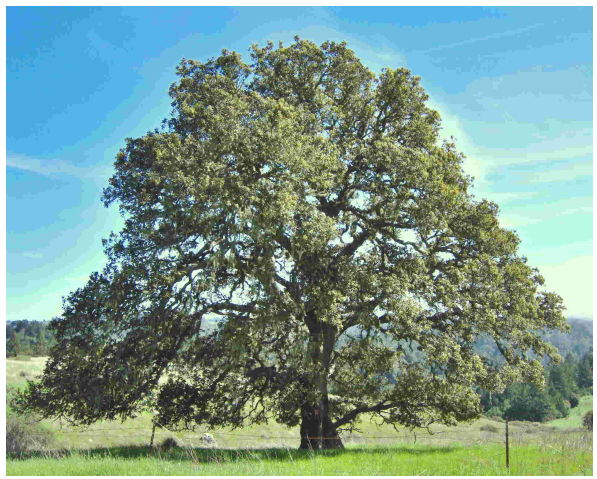
Examples: Set of equations

$$\begin{aligned} x + y + z &= h \\ h + 3 &= m - x \\ y - 7 &= h + z \end{aligned}$$

# Constraints

Imposing a certain restriction on a variable or set of variables
Find a solution such that every constraint is satisfied

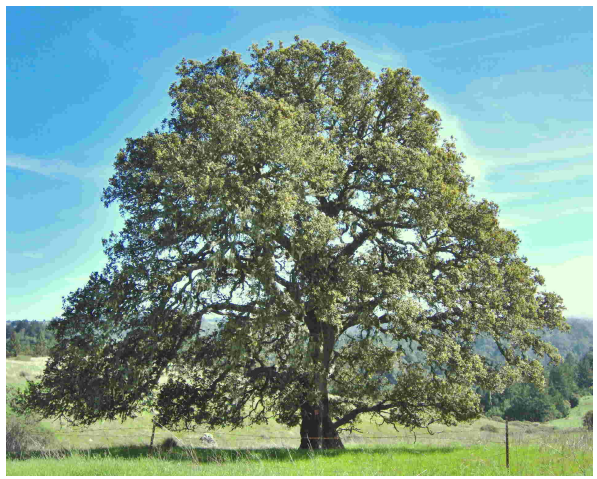
Examples: Set of equations, Tree Disequality



# Constraints

Imposing a certain restriction on a variable or set of variables
Find a solution such that every constraint is satisfied

Examples: Set of equations, Tree Disequality



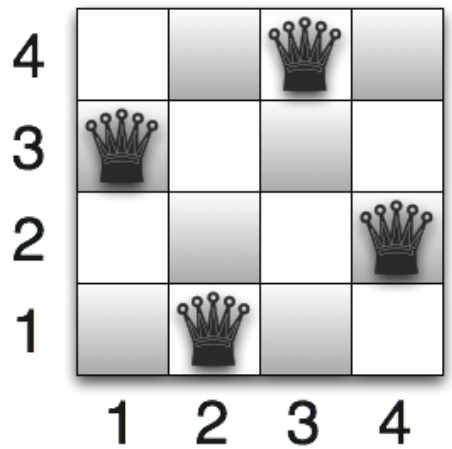
$$\text{'oak} \not\equiv \text{'pine}$$

$$\text{'}((1\ x)\ y\ 7) \not\equiv \text{'}(z\ 5\ w)$$

# Constraints

Imposing a certain restriction on a variable or set of variables
Find a solution such that every constraint is satisfied

Examples: Set of equations, Tree Disequality, N-Queens



# Send More Money

Find the correct letter values to satisfy the following equation:

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

Each letter represents a different digit in the range 0 through 9

# Motivation

# Motivation

- miniKanren does not use mathematical reasoning to rule out unrealizable values

# Motivation

- miniKanren does not use mathematical reasoning to rule out unrealizable values
- Performs very slowly on standard constraint problems

# Motivation

- miniKanren does not use mathematical reasoning to rule out unrealizable values
- Performs very slowly on standard constraint problems
- Extensions to miniKanren are incompatible with each other

# cKanren

- A framework for defining constraint systems on top of miniKanren

# cKanren

- A framework for defining constraint systems on top of miniKanren
- Retains all miniKanren functionality

# cKanren

- A framework for defining constraint systems on top of miniKanren
- Retains all miniKanren functionality
- Includes two constraint systems:
  *finite domains* and *tree disequality*

# cKanren

- A framework for defining constraint systems on top of miniKanren
- Retains all miniKanren functionality
- Includes two constraint systems:
  *finite domains* and *tree disequality*
- Easy to add or compose additional constraints systems

# Constraints Over Finite Domains

We can associate a *domain* with a variable $x$

# Constraints Over Finite Domains

We can associate a *domain* with a variable $x$

We consider only *finite domains* of natural numbers
such as $x \in \{1, 2, 3, 7, 8, 9\}$

... but there are others (interval domains, boolean domains, etc.)

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$
- $(+^{fd}\ u\ v\ w)$

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$
- $(+^{fd}\ u\ v\ w)$
- $(\neq^{fd}\ u\ v)$

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$
- $(+^{fd}\ u\ v\ w)$
- $(\neq^{fd}\ u\ v)$
- $(all\text{-}diff^{fd}\ v*)$

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$
- $(+^{fd}\ u\ v\ w)$
- $(\not\equiv^{fd}\ u\ v)$
- $(all\text{-}diff^{fd}\ v*)$

Derived goals:

- $in^{fd}$ to assign multiple variables a single initial domain

# Constraints Over Finite Domains

New operators:

- $(dom^{fd}\ x\ n*)$
- $(\leq^{fd}\ u\ v)$
- $(+^{fd}\ u\ v\ w)$
- $(\not\equiv^{fd}\ u\ v)$
- $(all\text{-}diff^{fd}\ v*)$

Derived goals:

- $in^{fd}$ to assign multiple variables a single initial domain
- $(<^{fd} u\ v)$

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    ...))
```

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    ...))
```

$x \in \{7, 8, 9, 10\}$

$y \in \{4, 5, 8, 9, 12\}$

$z \in \{1, 2, 12, 16\}$

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    (<=fd x y)
    ...))
```

$x \in \{7, 8, 9, 10\}$

$y \in \{8, 9, 12\}$

$z \in \{1, 2, 12, 16\}$

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    (<=fd x y)
    (+fd x y z)
    ...))
```

$x \in \{7, 8\}$

$y \in \{8, 9\}$

$z \in \{16\}$

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    (<=fd x y)
    (+fd x y z)
    (=/=fd x y)
    ...))
```

$x \in \{7\}$

$y \in \{9\}$

$z \in \{16\}$

# Example

```
(run* (q)
  (fresh (x y z)
    (domfd x '(7 8 9 10))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))
    (<=fd x y)
    (+fd x y z)
    (=/=fd x y)
    (== q `(,x ,y ,z))))
```

$\Rightarrow$ `((7 9 16))`

# Example

```
(run* (q)
  (fresh (x y z)
    (<=fd x y)
    (domfd x '(7 8 9 10))
    (+fd x y z)
    (=/=fd x y)
    (== q `(,x ,y ,z))
    (domfd y '(4 5 8 9 12))
    (domfd z '(1 2 12 16))))
```

$\Rightarrow$ `((7 9 16))`

# Disequality Over Trees

New operator $\not\equiv$ (more general than $\not\equiv^{fd}$)

# Disequality Over Trees

New operator $\not\equiv$ (more general than $\not\equiv^{fd}$)

```
(run* (q)
  (fresh (x y)
    (conde
      ((== x 1) (== y 1))
      ((== x 2) (== y 2))
      ((== x 1) (== y 2))
      ((== x 2) (== y 1)))
    (== q `(,x ,y))))
```

# Disequality Over Trees

New operator $\not\equiv$ (more general than $\not\equiv^{fd}$)

```
(run* (q)
  (fresh (x y)
    (conde
      ((== x 1) (== y 1))
      ((== x 2) (== y 2))
      ((== x 1) (== y 2))
      ((== x 2) (== y 1)))
    (== q `(,x ,y))))
```

$\Rightarrow$ `((1 1) (2 2) (1 2) (2 1))`

# Disequality Over Trees

New operator $\not\equiv$ (more general than $\not\equiv^{fd}$)

```
(run* (q)
  (fresh (x y)
    (conde
      ((== x 1) (== y 1))
      ((== x 2) (== y 2))
      ((== x 1) (== y 2))
      ((== x 2) (== y 1)))
    (=/= `(,x ,y) `(,y ,x))
    (== q `(,x ,y))))
```

# Disequality Over Trees

New operator $\not\equiv$ (more general than $\not\equiv^{fd}$)

```
(run* (q)
  (fresh (x y)
    (conde
      ((== x 1) (== y 1))
      ((== x 2) (== y 2))
      ((== x 1) (== y 2))
      ((== x 2) (== y 1)))
    (=/= `(,x ,y) `(,y ,x))
    (== q `(,x ,y))))
```

$\Rightarrow$ `((1 2) (2 1))`

# Implementation Overview

# Data Structures

cKanren uses a package to store information

# Data Structures

cKanren uses a package to store information

Substitution
Example: $((x\ .\ 1)\ (y\ .\ \#\mathrm{t})\ (z\ .\ x))$

# Data Structures

cKanren uses a package to store information

Substitution
Example: $((x\ .\ 1)\ (y\ .\ \#\mathrm{t})\ (z\ .\ x))$

Domain store
Example: $((x\ .\ (7\ 8\ 9))\ (y\ .\ (2\ 3\ 4\ 5)))$

# Data Structures

cKanren uses a package to store information

Substitution
Example: $((x\ .\ 1)\ (y\ .\ \#\mathrm{t})\ (z\ .\ x))$

Domain store
Example: $((x\ .\ (7\ 8\ 9))\ (y\ .\ (2\ 3\ 4\ 5)))$

Constraint store
Example: $((\texttt{proc}\ \leq^{fd}\ y\ x)\ (\texttt{proc}\ \textit{all-diff}^{fd}\ '(x\ z\ h\ 7)))$

# Framework

1. $\equiv$
2. Fixpoint algorithm
3. Consistency checks
4. *reify*

# Equivalence

$\equiv$

- Only constraint that is not kept in the constraint store

# Equivalence

$\equiv$

- Only constraint that is not kept in the constraint store
- Uses miniKanren unification

# Fixpoint Algorithm

No constraints directly interact with one another

# Fixpoint Algorithm

No constraints directly interact with one another

A framework function reruns constraints on newly ground variables

# Fixpoint Algorithm

No constraints directly interact with one another

A framework function reruns constraints on newly ground variables

Example:

```
(run* (q)
  (fresh (x)
    (infd x q '(1 2 3))
    (+fd x 1 q)
    ...
    (== x 2)
    ...))
```

# Fixpoint Algorithm

1. Receives variables $x*$
   For example, $x$ from previous slide, after being unified with 2

# Fixpoint Algorithm

1. Receives variables $x*$
   For example, $x$ from previous slide, after being unified with 2
2. Grabs current constraint store
   Constraint store `(...  (proc +fd x 1 q) ...)`

# Fixpoint Algorithm

1. Receives variables $x*$
   For example, $x$ from previous slide, after being unified with 2
2. Grabs current constraint store
   Constraint store `(...  (proc +fd x 1 q) ...)`
3. Run every constraint involving any variables in $x*$ again
   ... but only if the constraint is still in the store

   Reruns $+^{fd}$ constraint with new information that $x$ is 2.

# Consistency

Programs with irrelevant but unsatisfiable constraints will fail

# Consistency

Programs with irrelevant but unsatisfiable constraints will fail

```
(run* (q)
  (fresh (x y z)
    (infd x y z '(1 2))
    (all-difffd `(,x ,y ,z))
    (== q 5)))
```

$\Rightarrow$ ()

# Consistency

Programs with irrelevant but unsatisfiable constraints will fail

```
(run* (q)
  (fresh (x y z)
    (infd x y z '(1 2))
    (all-difffd `(,x ,y ,z))
    (== q 5)))
```

$\Rightarrow$ ()

Before returning anything to the user, each variable with finite domain constraints is re-evaluated, to guarantee that there is *at least one* acceptable value for each constrained variable.

# Reification

*reify*

- Produces the final result returned to the user

# Reification

*reify*

- Produces the final result returned to the user
- Constraint store may need consolidation or reformatting

# Reification

*reify*

- Produces the final result returned to the user
- Constraint store may need consolidation or reformatting

```
(run* (q) (=/= q 5))
```

$\Rightarrow$ `(( _0 : (=/= ((_0 . 5))))) `

# Parameters

# Parameters

*process-prefix*

Can rerun constraints for the variables with new associations

# Parameters

*process-prefix*
Can rerun constraints for the variables with new associations

*enforce-constraints*
Consistency checks before reification

# Parameters

*process-prefix*
Can rerun constraints for the variables with new associations

*enforce-constraints*
Consistency checks before reification

*reify-constraints*
Builds a Scheme data structure that packages the constraint information in a way that is readable to the user

# Composition

Having multiple constraint systems in the same session is tricky, as parameter definitions will overwrite each other

# Composition

Having multiple constraint systems in the same session is tricky, as parameter definitions will overwrite each other

```
(let ((ls (run* (q) (n-queens q 8))))
  (run* (s) (all-diffo ls)))
```

Implementor must define parameters in a way that makes sense

# Future Work

- Performance
- Specialized operators
- Adding $\alpha$Kanren
- Using different domains? Simulaneously?

# Questions?