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
Logic programming language extending Scheme

Three important operators: \(\equiv\), \textit{fresh}, and \textit{cond}^e
Logic programming language extending Scheme

Three important operators: ≡, 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 $\text{conde}$

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

$(\text{fresh} \ (x))$
  $(\text{conde}$
    $(== \ x \ 5))$
    $(== \ x \ 6)))$

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))
⇒ (3)
```
Logic programming language extending Scheme

Uses \textit{run} as an interface operator

\begin{verbatim}
(run1 (y)
  (fresh (x z)
   (== x z)
   (== 3 y)))
⇒ (3)
\end{verbatim}
Logic programming language extending Scheme

Uses `run` as an interface operator

\[
\begin{align*}
\text{(run1} (y) & \quad \text{(run1} (y) \\
\text{  (fresh} (x \ z) & \quad \text{(fresh} (x \ z) \\
\text{   (==} x \ z & \quad \text{   (==} x \ z \\
\text{    (==} 3 \ y) & \quad \text{    (==} 3 \ z) \\
& \quad \text{    (==} y \ x) & \quad \text{    (==} y \ x) \\
\Rightarrow (3) & \quad \Rightarrow (3) & \quad \Rightarrow (3)
\end{align*}
\]
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

\[ x + y + z = h \]

\[ h + 3 = m - x \]

\[ y - 7 = h + z \]
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

\[ x + y + z = h \]
\[ h + 3 = m - x \]
\[ y - 7 = h + z \]
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

'oak \not\equiv 'pine

'((1\ x)\ y\ 7) \not\equiv '(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:

\[
\begin{align*}
S & 
\phantom{E}N & 
D \\
+ & 
M & 
O & 
R & 
E
\end{align*}
\]

\[
\overline{M O N E Y}
\]

Each letter represents a different digit in the range 0 through 9.
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
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 $x$. ... but there are others (interval domains, boolean domains, etc.).
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.)
New operators:
▶ \((\text{dom}^{fd} \times n^*)\)
New operators:
- \((\text{dom}^f d \times n^*)\)
- \((\leq^f d \ u \ v)\)
Constraints Over Finite Domains

New operators:

- \((\text{dom}^d x n^*)\)
- \(\leq^d u v\)
- \(+^d u v w\)
New operators:

- $(\text{dom}^{fd} \times n^*)$
- $(\leq^{fd} u \; v)$
- $(+^{fd} u \; v \; w)$
- $(\not=^{fd} u \; v)$

Derived goals:

- $\text{in}^{fd}$ to assign multiple variables a single initial domain
- $(\prec^{fd} u \; v)$
New operators:
- $(\text{dom}^\text{fd} \times n^*)$
- $(\leq^\text{fd} u v)$
- $(+^\text{fd} u v w)$
- $(\not=^\text{fd} u v)$
- $(\text{all-diff}^\text{fd} v^*)$
New operators:
- \((\text{dom}^{fd} \times n^*)\)
- \((\leq^{fd} u v)\)
- \((+^{fd} u v w)\)
- \((\not=^{fd} u v)\)
- \((\text{all-diff}^{fd} v^*)\)

Derived goals:
- \(in^{fd}\) to assign multiple variables a single initial domain
Constraints Over Finite Domains

New operators:
\[
\begin{align*}
&\text{▶ } (\text{dom}^{fd} \times n^*) \\
&\text{▶ } (\leq^{fd} u v) \\
&\text{▶ } (\text{+}^{fd} u v w) \\
&\text{▶ } (\neq^{fd} u v) \\
&\text{▶ } (\text{all-diff}^{fd} v^*)
\end{align*}
\]

Derived goals:
\[
\begin{align*}
&\text{▶ } in^{fd} \text{ to assign multiple variables a single initial domain} \\
&\text{▶ } (<^{fd} u v)
\end{align*}
\]
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

\[(\text{run}^*(q))\]

\[(\text{fresh } x \ y \ z)\]

\[(\text{domfd } x \ (7 \ 8 \ 9 \ 10)) \quad x \in \{7, 8, 9, 10\}\]

\[(\text{domfd } y \ (4 \ 5 \ 8 \ 9 \ 12)) \quad y \in \{4, 5, 8, 9, 12\}\]

\[(\text{domfd } z \ (1 \ 2 \ 12 \ 16)) \quad z \in \{1, 2, 12, 16\}\]

\[\ldots)\]
Example

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

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

\(x \in \{7\}\)
\(y \in \{9\}\)
\(z \in \{16\}\)
(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)))

⇒ ((7 9 16))
(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))))
⇒ ((7 9 16))
Disequality Over Trees

New operator $\neq$ (more general than $\neq^{fd}$)
Disequality Over Trees

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

\[
\text{run* (q)} \\
\quad\text{(fresh (x y)} \\
\quad\quad\text{(conde)} \\
\quad\quad\quad\text{((== x 1) (== y 1))} \\
\quad\quad\quad\text{((== x 2) (== y 2))} \\
\quad\quad\quad\text{((== x 1) (== y 2))} \\
\quad\quad\quad\text{((== x 2) (== y 1))} \\
\quad\quad\quad\text{(== q '(,x ,y)))}
\]
Disequality Over Trees

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

$$\text{(run* (q)}$$

$$\quad\text{(fresh (x y)}$$

$$\quad\text{(conde}$$

$$\quad\quad\text{((== x 1) (== y 1))}$$

$$\quad\quad\text{((== x 2) (== y 2))}$$

$$\quad\quad\text{((== x 1) (== y 2))}$$

$$\quad\quad\text{((== x 2) (== y 1))}$$

$$\quad\quad\text{(== q \{\text{'(,x ,y)'}\})}$$

$$\Rightarrow\quad\text{((1 1) (2 2) (1 2) (2 1))}$$
Disequality Over Trees

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

$$(\text{run}\ast (q))$$

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

\[
\text{(run* (q)} \\
\quad \text{(fresh (x y)} \\
\quad \quad \text{(conde)} \\
\quad \quad \quad \quad \text{((== x 1) (== y 1))} \\
\quad \quad \quad \quad \text{((== x 2) (== y 2))} \\
\quad \quad \quad \quad \text{((== x 1) (== y 2))} \\
\quad \quad \quad \quad \text{((== x 2) (== y 1)))} \\
\quad \quad \quad \quad (=/= \, (\text{x } \text{y}) \, (\text{,y } \text{x})) \\
\quad \quad \quad \quad (== \, \text{q} \, (\text{,x } \text{y})) \\
\quad \quad \Rightarrow \, ((\text{1 2}) \, (\text{2 1})))
\]
Implementation Overview
cKanren uses a package to store information
Data Structures

cKanren uses a package to store information

Substitution
Example: ((x . 1) (y . #t) (z . x))
Data Structures

cKanren uses a package to store information

Substitution
Example: \(((x \ . \ 1) \ (y \ . \ #t) \ (z \ . \ x))\)

Domain store
Example: \(((x \ . \ (7 \ 8 \ 9)) \ (y \ . \ (2 \ 3 \ 4 \ 5)))\)
cKanren uses a package to store information

Substitution
Example: ((x . 1) (y . #t) (z . x))

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

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

1. ≡
2. Fixpoint algorithm
3. Consistency checks
4. reify
Equivalence

- Only constraint that is not kept in the constraint store
Equivalence

- 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 (\ldots \ (\text{proc +fd } x \ 1 \ q) \ldots )
Fixpoint Algorithm

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

2. Grabs current constraint store
   Constraint store ($\ldots$ (proc $+fd$ $x$ 1 $q$) $\ldots$)

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-diffld '(',x ,y ,z))
     (== q 5)))
⇒ ()
```
Programs with irrelevant but unsatisfiable constraints will fail

\begin{verbatim}
(run* (q)
  (fresh (x y z)
    (infd x y z '(1 2))
    (all-diffffd '(@(x, y, z))
      (== q 5)))
⇒ ()
\end{verbatim}

Before returning anything to the user, each variable with finite domain constraints is re-evaluated, to guarantee that there is \textit{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

\[(\text{run* (q) (=/= q 5)})\]

\[\Rightarrow ((\_0 : (=/= ((\_0 . 5))))))\]
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
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? Simultaneously?
Questions?