Relational Programming in Smalltalk

Massimo Nocentini

University of Florence, Italy

ESUG2018

```
outline
    ^ LinkedList new
        add: 'me and motivations';
        add: 'Refutation and Unification resolutions ';
        add: 'Microkanren in Smalltalk';
        add: 'Dyck paths and the McCulloch machine';
        yourself
```

Hi!

```
$ whoami
Massimo Nocentini
PhD student @ University of Florence
Mathematician (algebraic combinatorics, formal methods for algs)
Programmer (automated reasoning, logics and symbolic comp)
https://github.com/massimo-nocentini
$ clear
```

I believe *microkanren* is, first of all, an *educational beast*, concerning unification, lazy streams, backtracking and optimization; the abstract definition was shown by *Dan Friedman* and *Jason Hemann* at Scheme '13, Alexandria.

I repeat the exercise of writing it in:

- Python, native generators :) limits on recursive calls :(
- OCaml, algebraic datatypes :) hard to extend :(
- Smalltalk, simple, fast and clear :) many dispatching msgs :/

Main idea

In math a relation P is usually characterized by

$$\forall a, b, c.\, P(a, b, c) \leftrightarrow a + b = c \quad \text{entails} \quad P(1, 2, 3)$$

can be expressed using either the *imperative style*

```
a := 1.
b := 2.
c := a + b.
Object assert: [ c = 3 ].
```

or the *functional style*

```
Object assert: [
    ([ :a :b | a + b ] value: 1 value: 2) = 3 ]
```

or, finally, the *declarative style*

```
Object assert: [
    [ :a :b :c | a + b = c ] value: 1 value: 2 value: 3 ]
```

Resolution by *Refutation*

Let α be a sentence in *CNF* and $M(\alpha)$ the set of models that satisfy it, where a model is a set of assignments that make α true.

α is *valid* if it is true in *all* models; oth,
α is *satisfiable* if it is true in *some* model.

Let $\models$ and $\Rightarrow$ denote the *entail* and *imply* relations, respectively, in

$$\begin{gathered}\alpha \models \beta \leftrightarrow \\ M(\alpha) \subseteq M(\beta) \leftrightarrow \\ (\alpha \Rightarrow \beta) \text{ is valid } \leftrightarrow \\ \neg(\neg\alpha \vee \beta) \text{ is unsatisfiable;}\end{gathered}$$

therefore, to prove a sentence α reduces to decide

$$\neg\alpha \models \perp \quad \leftrightarrow \quad \alpha \text{ is valid,}$$

where $\perp$ denotes the empty clause, namely *falsehood*.

Resolution by *Refutation*

The *resolution rule* is a *complete* inference algorithm,

$$\frac{(l_0, \ldots, l_i, \ldots, l_{j-1}) \quad (m_0, \ldots, m_r, \ldots, m_{k-1}) \quad l_i = \neg m_r}{(l_0, \ldots, l_{i-1}, l_{i+1}, \ldots, l_{j-1}, m_0, \ldots, m_{r-1}, m_{r+1}, \ldots, m_{k-1})}$$

where $(l_0, \ldots, l_i, \ldots, l_{j-1}) = l_0 \vee \ldots \vee l_i \vee \ldots \vee l_{j-1}$,
for all $l_q, m_w \in \{0, 1\}$.

The *DPLL* algorithm is a recursive, depth-first enumeration of models using the resolution rule, paired with heuristics *early termination*, *pure symbol* and *unit clause* to speed up.

Resolution by *Unification*

Unification is the process of solving *equations among symbolic expressions*; a *solution* is denoted as a *substitution* θ, namely a mapping that assigns a symbolic values to free variables.

Let x and y be free variables, the set

$$\{\mathrm{cons}(x, \mathrm{cons}(x, \mathrm{nil})) = \mathrm{cons}(2, y)\}$$

has solution $\theta = \{x \mapsto 2, y \mapsto \mathrm{cons}(2, \mathrm{nil})\}$; moreover, the set

$$\{y = \mathrm{cons}(2, y)\}$$

has no *finite* solution; on the other hand,

$$\theta = \{y \mapsto \mathrm{cons}(2, \mathrm{cons}(2, \mathrm{cons}(2, \ldots)))\}$$

is a solution upto *bisimulation*.

Resolution by *Unification*

let G be a set of equations, unification rules are

delete $G \cup \{t = t\} \rightarrow G$

decompose $G \cup \{f(s_0, \ldots, s_k) = f(t_0, \ldots, t_k)\}$ entails

$$G \cup \{s_0 = t_0, \ldots, s_k = t_k\}$$

conflict if $f \neq g \vee k \neq m$ then

$$G \cup \{f(s_0, \ldots, s_k) = g(t_0, \ldots, t_m)\} \rightarrow \bot$$

eliminate if $x \notin vars(t)$ and $x \in vars(G)$ then

$$G \cup \{x = t\} \rightarrow G\{x \mapsto t\} \cup \left\{x \triangleq t\right\}$$

occur check if $x \in vars(f(s_0, \ldots, s_k))$ then

$$G \cup \{x = t(s_0, \ldots, s_k)\} \rightarrow \bot$$

microkanren

Let *solution*, *substitution* and *state* be synonyms; so, μ-kanren

- is a *DSL* for relational programming written in Scheme
- is a *purely functional* core of *miniKanren*
- provides *explicit streams* of satisfying states
- encodes math rel using a *goal-based* approach
- uses resolution by unification via *structural induction*

A *goal* is an object that responds to the `#onState:` selector, it receives a *substitution* and returns a `Chain` object of substitutions.

Chain hierarchy

We model a (possibly infinite) space of objects with the `Chain` hierarchy, which has `Bottom` and `Knot` as subclasses which denote the empty and a populated set, respectively.

Although Pharo provides the `Generator` class, we write our version of *lazy* enumeration, which is purely functional (neither clever uses of `thisContext` nor reentrant blocks).

Encodes the two *monadic* operations `mplus` and `bind`, which allow us to merge two `Chains` and to combine a `Chain` obj to yield an extended `Chain` obj, respectively.

Dispatch over two strategies `Sequential` and `Interleaved` in order to *enumerate* solution spaces.

Chain subclass: #Bottom

```
BlockClosure>>links: anObj
    ^ Chain item: anObj linker: self

Chain class>>bottom
    ^ Bottom new

Chain class>>item: anObj linker: aBlockClosure
    ^ Knot new
      item: anObj;
      linker: aBlockClosure;
      yourself

bind: aGoal interleaved: anInterleaved
    ^ self

mplus: anotherChain interleaved: anInterleaved
    ^ anotherChain value

atMost: anInteger
    ^ self

mature
    ^ LinkedList new
```

Chain subclass: #Knot

```
bind: aGoal interleaved: anInterleaved
    | alpha beta |
    alpha := aGoal onState: item.
    beta := [ self next bind: aGoal interleaved: anInterleaved ].
    ^ alpha mplus: beta interleaved: anInterleaved

mplus: anotherChain interleaved: anInterleaved
    ^ [ :_ | anotherChain value
                mplus: [ self next ]
                interleaved: anInterleaved ] links: item

next
    ^ linker value: item

atMost: n
    ^ n isZero
        ifTrue: [ Chain bottom ]
        ifFalse: [ [ :_ | self next value atMost: n - 1 ] links: item ]

mature
    ^ self next mature
        addFirst: item;
        yourself
```

ChainTest

```
ints: i
    ^ [ :a | self ints: a + 1 ] links: i

fib: m fib: n
    ^ [ :_ | self fib: n fib: m + n ] links: m

collatz: o
    ^ [ :_ | o even
                ifTrue: [ self collatz: o / 2 ]
                ifFalse: [ self collatz: 3 * o + 1 ] ] links: o

testNumbers
    self
        assert: (self nats atMost: 10) mature
        equals: (0 to: 9).
    self
        assert: (self fibs atMost: 10) mature
        equals: {0 . 1 . 1 . 2 . 3 . 5 . 8 . 13 . 21 . 34}.
    self
        assert: ((self collatz: 10) atMost: 10) mature
        equals: {10 . 5 . 16 . 8 . 4 . 2 . 1 . 4 . 2 . 1}.
```

Goal hierarchy

microkanren represents math rels using the `Goal` hierarchy

- `Succeed` it *is* satisfied by *each* sub;
- `Fail` it *is not* satisfied by *any* sub;
- `Or` it is satisfied if at least one obj it consumes can be satisfied;
- `And` it is satisfied if both objs it consumes can be satisfied;
- `Fresh` it introduces logic vars into the goal it combines;
- `Unify` it is satisfied if the two objs it consumes can be unified.

Moreover, a substitution (aka, a set of assignments) is represented by a `Dictionary` obj, wrapped by `State` obj to count the number of logic vars introduced by `Fresh` goals.

Our substitutions are *triangular* in the sense that if

$$\theta = \{x \mapsto y, y \mapsto z, z \mapsto 3\}$$

then $x \mapsto 3$ is subsumed by θ, this is implemented in `State>>#walk`.

State

```
State>>walk: anObj
    | k |
    k := anObj.
    [ k := substitution at: k ifAbsent: [ ^ k ] ] repeat
```

A substitution is extended by

```
State>>at: aVar put: aValue
    | s |
    s := substitution copy.
    s
        at: aVar
        ifPresent: [ :v |
            aValue = v
                ifFalse: [ UnificationError signal ] ]
        ifAbsent: [ s at: aVar put: aValue ].
    ^ self class new
        birthdate: birthdate;
        substitution: s;
        yourself
```

Goal subclass: #[Succeed | Fail | Disj | Conj]

In parallel, `true` and `false` have logical brothers

```
Succeed>>onState: aState
    ^ Chain with: aState

Fail>>onState: aState
    ^ Chain bottom
```

respectively; btw, for conjuction and disjunction we have

```
Disj>>onState: aState
    ^ interleaving of: ((either onState: aState)
      mplus: [ or onState: aState ])

Conj>>onState: aState
    ^ interleaving of: ((both onState: aState) bind: and)
```

Goal subclass: #Fresh

```
Fresh>>onState: aState
    ^ aState collectVars: (1 to: receiver numArgs) forFresh: self

State>>collectVars: aCollection forFresh: aFresh
    | nextState vars |
    nextState := self class new
        substitution: substitution;
        birthdate: birthdate + aCollection size;
        yourself.
    vars := aCollection collect: [ :i | Var id: i ].
    ^ aFresh onState: nextState withVars: vars

Fresh>>onState: aState withVars: aCollection
    | g |
    vars := aCollection.
    g := receiver valueWithArguments: vars.
    ^ g onState: aState

BlockClosure>>fresh
    ^ Goal fresh: self
```

Goal subclass: #Unify

```
Object>>unifyWith: another
    ^ Goal unify: self with: another

Unify>>onState: aState
    ^ [ | extended_state |
        extended_state := Unifier new
                            unify: this with: that onState: aState.
        Goal succeed onState: extended_state ]
            on: UnificationError
            do: [ Goal fail onState: aState ]

Unifier>>unify: anObj with: anotherObj onState: aState
    | aWalkedObj anotherWalkedObj |
    aWalkedObj := aState walk: anObj.
    anotherWalkedObj := aState walk: anotherObj.
    ^ aWalkedObj unifyWith: anotherWalkedObj
                 usingUnifier: self
                 onState: aState
```

Unifier

```
unifyObject: anObj withObject: anotherObj onState: aState
    ^ anObj = anotherObj
        ifTrue: [ aState ]
        ifFalse: [ UnificationError signal ]

unifyVar: aVar withObject: anObject onState: aState
    ^ aState at: aVar put: anObject

unifyVar: aVar withVar: anotherVar onState: aState
    ^ aVar = anotherVar
        ifTrue: [ aState ]
        ifFalse: [
            self unifyVar: aVar withObject: anotherVar onState: aState ]

unifyLinkedList: c withLinkedList: d onState: aState
    ^ c size = d size
        ifTrue: [ (c zip: d)
            inject: aState
            into: [ :s :p | self unify: p key with: p value onState: s ] ]
        ifFalse: [ UnificationError signal ]
```

Goal subclass: #Cond

```
Cond>>if: ifGoal then: thenGoal
    clauses add: ifGoal -> thenGoal

Cond>>ifPure: aStrategy
    if := [ :c :o |
        IfPure new
            question: c key answer: c value otherwise: o;
            streamCombinationStrategy: aStrategy;
            yourself ]

Cond>>e
    self ifPure: Sequential new

Cond>>i
    self ifPure: Interleaved new

Cond>>onState: aState
    | g |
    else ifNil: [ self else: false asGoal ].
    g := clauses copy
            add: else;
            reduceRight: if.
    ^ g onState: aState
```

Dyck paths

Let $\mathcal{D}$ be the set of *Dyck paths* and let $\rightsquigarrow$ be the *CFG*

$$\rightsquigarrow = \varepsilon \mid (\rightsquigarrow) \rightsquigarrow$$

where ε is the empty string; so, *enumerate* $\mathcal{D}$ *using* $\rightsquigarrow$.

```
dycko: alpha
    ^ Goal cond e
      if: alpha nilo then: true asGoal;
      else: [ :beta :gamma |
          (sexpTheory let: alpha
                      be: ($( cons: beta)
                      append: ($) cons: gamma)) &
          ([ self dycko: beta ] eta &
           [ self dycko: gamma ] eta) ] fresh
```

Dyck paths

```
testDycko
    | g |

    "enumeration"
    g := [ :alpha | combTheory dycko: alpha ] fresh.
    self
        assert: (g solutions atMost: 20)
        equals:
            ({nil . '()' . '(())' . '()()' . '(()())' . '()(())' .
            '(())()' . '()()()' . '(()()())'.  '()(()())' .
            '(())(())' . '()()(())' . '((()))' . '()(())()' .
            '(())()()' . '()()()()'.  '(()()()())' . '()(()()())' .
            '(())(()())' . '()()(()())'} collect: #asCons).

    "an invalid Dyck path"
    g := [ :alpha | combTheory dycko: '(()(())()(' asCons ] fresh.
    self assert: g solutions all equals: {}
```

McCulloch's machine and the MC lock puzzle

Let X and Y be natural numbers in machine

$$\mathcal{C} = \left\{ \frac{}{2X \overset{\circ}{\rightarrow} X}, \frac{X \overset{\circ}{\rightarrow} Y}{3X \overset{\circ}{\rightarrow} Y2Y} \right\}$$

question: *does exist a number* α *such that* $\alpha \overset{\circ}{\rightarrow} \alpha$?

```
consumes: two_alpha produces: alpha machine: aMachine
    ^ two_alpha unifyWith: (2 cons: alpha)

consumes: three_alpha produces: alpha_two_alpha machine: aMachine
    ^ [ :beta :gamma |
        (three_alpha unifyWith: (3 cons: beta)) &
        ((self associate: gamma is: alpha_two_alpha machine: aMachine) &
        (aMachine proves: beta relates: gamma)) ] fresh
```

McCulloch's machine and the MC lock puzzle

```
InductiveRelationsTheory>>proves: anObj relates: anotherObj
    | g |
    g := Goal cond i.
    rules do: [ :r | g if: (r consumes: anObj
                               produces: anotherObj
                               machine: self)
                       then: true asGoal ].
    ^ g

testFirstMachine
    | g |

    "McCulloch's first machine"
    g := [ :a | self mcculloch proves: a relates: a ] fresh.
    self assert: (g solutions atMost: 1) equals: {#(3 2 3) asCons}.

    "Montecarlo lock"
    g := [ :a | self mclock proves: a relates: a ] fresh.
    self
        assert: ((g solutions atMost: 1) collect: #asLinkedList)
        equals: {#(5 4 6 4 2 5 4 6 4 2)}
```

End

A quick check:

$$\overline{323 \overset{\circ}{\rightarrow} \qquad}$$

End

A quick check:

$$\dfrac{\overline{\;23 \overset{\circ}{\rightarrow}\;}}{323 \overset{\circ}{\rightarrow}}$$

End

A quick check:

$$\frac{\overline{23 \overset{\circ}{\rightarrow} 3}}{323 \overset{\circ}{\rightarrow}}$$

End

A quick check:

$$\frac{\overline{23 \xrightarrow{\circ} 3}}{323 \xrightarrow{\circ} 323}$$

End

A quick check:

$$\cfrac{\overline{23 \overset{\circ}{\rightarrow} 3}}{323 \overset{\circ}{\rightarrow} 323}$$

Future directions:

- unification is just a *constraint*...
- ...so add *disequality*, *type checks* and other constraints
- *impure* operators, such as Prolog's *cut* (!)
- automatic message dispatching for unifications

Thanks!