

Resolution the miniKanren Way

Joe Near

Resolution

- Proof method for propositional and first-order logic
- Input must be in clausal form
- Can be refined to improve efficiency
- Refinements may be combined without loss of completeness

Propositional Logic

- Propositional symbols p, q, \dots
- Logical operators $\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow$
- Each propositional symbol stands for a complete proposition
- No quantification

First-order Logic

- Variables x, y, \dots
- Relations P, Q, \dots
- Functions f, g, \dots
- Logical operators $\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow$
- Quantifiers \forall, \exists

Conjunctive Normal Form

- We transform arbitrary first-order logic formulas into a set of clauses of the form:

$$(l_1 \vee l_2 \vee \dots) \wedge \dots \wedge (l_m \vee l_n \vee \dots)$$

- Each l_n is called a literal; each literal is a term composed of functions, predicates, and variables, or such a term that has been negated
- All variables are universally quantified
- Each clause is a finite multiset of literals
- In Scheme, we write the set of clauses as a list of lists:

$$((l_1 l_2 \dots) \dots (l_m l_n \dots))$$

Skolemization

- Removes existential quantification
- For a formula $\exists x M$:
 1. Find the free variables of $\exists x M$
 2. Generate a Skolem Function f that takes the free variables of M
 3. Substitute a call to f for each occurrence of x in M

CNF in Scheme

- Macros for quantification, so λ does substitution for us:

(define-syntax A

(syntax-rules ()

((A *var body*) '(forall *var* ,(lambda (*var*) 'body))))))

(define-syntax E

(syntax-rules ()

((E *var body*) '(ex *var* ,(lambda (*var*) 'body))))))

Finding Free Variables

```
(define fv
  (lambda (fml)
    (pmatch fml
      [,x (var? x) (list x)]
      [(not ,x) () (fv x)]
      [(,op ,x ,y) (member op '(and or => <=>)) (append (fv x) (fv y))]
      [(forall ,x ,t) () (remq x (fv t))]
      [(exists ,x ,t) () (remq x (fv t))]
      [(,f . ,args) () (apply append (map fvt args))]
      [else '()])))
```

```
(define fvt
  (lambda (fml)
    (pmatch fml
      [,x (var? x) (list x)]
      [(,f . ,args) () (apply append (map fvt args))]
      [else '()])))
```

Negation Normal Form

```
(define nnf
  (lambda (fml)
    (pmatch fml
      ((not (not ,a)) () (nnf a))
      ((not (forall ,var ,gfml)) () (nnf '(ex ,var ,(lambda (x) '(not ,(gfml x))))))
      ((not (ex ,var ,gfml)) () (nnf '(forall ,var ,(lambda (x) '(not ,(gfml x))))))
      ((not (and . ,fmls)) () (nnf '(or ,@(map (lambda (x) '(not ,x)) fmls)))
      ((not (or . ,fmls)) () (nnf '(and ,@(map (lambda (x) '(not ,x)) fmls)))
      ((=> ,a ,b) () (nnf '(or (not ,a) ,b)))
      ((not (=> ,a ,b)) () (nnf '(and ,a (not ,b))))
      ((<=> ,a ,b) () (nnf '(or (and ,a ,b) (and (not ,a) (not ,b))))
      ((not (<=> ,a ,b)) () (nnf '(or (and (not ,a) ,b) (and ,a (not ,b))))
      ((forall ,varx ,gfml) () (nnf (gfml (var varx))))
      ((and . ,fmls) () '(and ,@(map (lambda (x) (nnf x)) fmls)))
      ((or . ,fmls) () '(or ,@(map (lambda (x) (nnf x)) fmls)))
      ((ex ,v ,gfml) () (let* ((fvars (rem-dups (fv (subst-fm '(ex ,v ,gfml))))))
                          (fml-ex '(,(gensym) . ,fvars))
                          (nnf (gfml fml-ex))))
      (else fml))))
```

CNF

```
(define distrib2
  (lambda (s1 s2)
    (cond
      [(null? s1) '()]
      [else (append (map (lambda (x) (append (car s1) x)) s2)
                    (distrib2 (cdr s1) s2))])))
```

```
(define purecnf
  (lambda (fm)
    (pmatch fm
      [() () '()]
      [(or ,p ,q) () (distrib2 (purecnf p) (purecnf q))]
      [(and ,p ,q) () (append (purecnf p) (purecnf q))]
      [else (list (list fm))])))
```

```
(define clausal
  (lambda (fml)
    (filter nontautology? (map rem-dups (purecnf (nnf fml)))))
```


Resolution

● **Resolution:** two clauses of the form

● $(A) \cup C_1$

● $(B) \cup C_2$

for which there exists a substitution θ such that $A\theta = \neg B\theta$ can be resolved to form the resolvent clause $C_1\theta \cup C_2\theta$

● **Factorization:** a clause of the form

● $(A, B) \cup C$

for which there exists a substitution θ such that $A\theta = B\theta$ can be factored to form the clause $(A\theta) \cup C\theta$

Resolution Proving

- Repeatedly apply the resolution and factorization rules to a set of clauses, adding each new clause to the set, until the empty clause is derived
- Deriving the empty clause means that the original formula is **unsatisfiable**
- If we negate the original theorem and then prove the negation unsatisfiable, then the original theorem is **valid**

A Simple Prover

```
(define factor
  (lambda (cl fact-cl)
    (fresh (copy lit)
      (copy-termo cl copy)
      (rembero lit copy fact-cl)
      (membero lit fact-cl))))

(define resolve
  (lambda (cl1 cl2 res-cl)
    (fresh (ccl1 ccl2 res-cl1 res-cl2 neg lit1 lit2)
      (copy-termo cl1 ccl1)
      (copy-termo cl2 ccl2)
      (rembero lit1 ccl1 res-cl1)
      (rembero lit2 ccl2 res-cl2)
      (negateo lit2 neg)
      (≡ lit1 neg)
      (appendo res-cl1 res-cl2 res-cl))))
```

```
(define res-step
  (lambda (cls out)
    (fresh (cl1 cl2 res-cl)
      (conde
        [(membero cl1 cls)
         (membero cl2 cls)
         (resolve cl1 cl2 res-cl)
         (≡ '(,res-cl . ,cls) out)]
        [(membero cl1 cls)
         (factor cl1 res-cl)
         (≡ '(,res-cl . ,cls) out)]))))

(define proof-loop
  (lambda (cls)
    (fresh (new-cls)
      (res-step cls new-cls)
      (conde
        [(membero '() new-cls)]
        [(proof-loop new-cls)]))))
```

Negation

```
(define negateo
  (lambda (tm neg)
    (fresh (na nd)
      (conde
        [(≡ '(not ,neg) tm)]
        [(never-pairo tm) (≡ '(not ,tm) neg)]
        [(≡ '(,na . ,nd) tm)
         (never-equalo na 'not)
         (≡ '(not ,tm) neg)]))))))
```

Running the Prover

```
(define prove
  (lambda (axioms thm)
    (let ((thm (clausal '(not ,thm)))
          (axioms (clausal (build-and axioms))))
      (printf "num clauses: ~s\n" (+ (length axioms) (length thm)))
      (pretty-print thm)
      (pretty-print axioms)
      (printf "-\n")
      (let ((result (run 1 (q)
                        (proof-loop (append axioms thm)))))
        (if (null? result)
            (error 'prove "failed to prove")
            (pretty-print result))))))
```

Problems

- No proof produced
- Slow, due to duplicated work

Linear Resolution

- Start with one “given” clause from the set of clauses
- At each step, either:
 - Resolve the given clause with another clause
 - Factor the given clauseIn either case, the resulting clause becomes the new given clause
- Proof tree is easy to construct, since it is linear
- Duplication of work is (mostly) avoided

Linear Resolution

```
(define res-loop
  (lambda (given cls proof out)
    (fresh (cl new-cl a d)
      (conde
        [(membero cl cls)
         (resolve given cl new-cl)
         (conde
           [( $\equiv$  '() new-cl) ( $\equiv$  '(() ,cl . ,proof) out)]
           [( $\equiv$  '(,a . ,d) new-cl)
            (res-loop new-cl '(,given . ,cls) '(,new-cl ,cl . ,proof) out)]])
        [(factor given new-cl)
         (res-loop new-cl '(,given . ,cls) '(,new-cl . ,proof) out)])))))
```

```
(define proof-loop
  (lambda (cls proof)
    (fresh (given clsp)
      (rembero given cls clsp)
      (res-loop given clsp '(,given) proof))))
```

Positive Resolution

- In each application of the resolution rule, at least one clause resolved upon must contain only positive literals
- Another way to reduce duplication of work
- Despite the strength of the restriction, completeness is retained

Positive Resolution

```
(define positive-clauseo
  (lambda (cl)
    (fresh (a d res)
      (conde
        [(≡ '() cl)]
        [(≡ '(,a . ,d) cl)
          (conde
            [(never-pairo a)]
            [(fresh (aa dd)
              (≡ '(,aa . ,dd) a)
              (never-equalo aa 'not))])
          (positive-clauseo d)]))))))
```

```
(define resolve
  (lambda (cl1 cl2 res-cl)
    (fresh (ccl1 ccl2 res-cl1 res-cl2 neg lit1 lit2)
      (conde
        [(positive-clauseo cl1)]
        [(positive-clauseo cl2)])
      (copy-termo cl1 ccl1)
      (copy-termo cl2 ccl2)
      (rebero lit1 ccl1 res-cl1)
      (rebero lit2 ccl2 res-cl2)
      (negateo lit2 neg)
      (≡ lit1 neg)
      (appendo res-cl1 res-cl2 res-cl))))))
```

Set of Support

- Split the set of clauses into two subsets: the set of support, and the unsupported set
- The unsupported set must be satisfiable
- In each resolution, one of the clauses resolved upon must be a member of the set of support
- Choose the set of support to be the set of clauses that contain no positive literals
- This ensures that the unsupported set is satisfiable, but a larger unsupported set could be chosen using knowledge of the specific problem
- The larger the unsupported set, the more dramatic the improvement in performance

Finding the Set of Support

```
(define negative-clause?
  (lambda (cl)
    (pmatch cl
      [() () #t]
      [((not ,x) . ,rest) () (negative-clause? rest)]
      [else #f])))
```

```
(define set-of-support
  (lambda (cls)
    (filter negative-clause? cls)))
```

Using the Set in the Prover

```
(define res-step
  (lambda (cls setup new-setup out)
    (fresh (cl1 cl2 res-cl)
      (conde
        [(membero cl1 cls)
         (membero cl2 cls)
         (conde
           [(membero cl1 setup)]
           [(membero cl2 setup)])
         (resolve cl1 cl2 res-cl)
         (≡ '(,res-cl . ,setup) new-setup)
         (≡ '(,res-cl . ,cls) out)]
        [(membero cl1 cls)
         (factor cl1 res-cl)
         (≡ '(,res-cl . ,setup) new-setup)
         (≡ '(,res-cl . ,cls) out)]))))
```

```
(define proof-loop
  (lambda (cls setup)
    (fresh (new-cls new-setup)
      (res-step cls setup new-setup new-cls)
      (conde
        [(membero '() new-cls)]
        [(proof-loop new-cls new-setup)]))))
```

Tautology Elimination

Tautologous clauses never help us towards a proof, so we can shorten the set of clauses we have to deal with by avoiding them

(define *resolve*

(lambda (*cl1 cl2 res-cl*)

(fresh (*ccl1 ccl2 res-cl1 res-cl2 neg lit1 lit2*)

(copy-termo cl1 ccl1)

(copy-termo cl2 ccl2)

(rembero lit1 ccl1 res-cl1)

(rembero lit2 ccl2 res-cl2)

(negateo lit2 neg)

(≡ lit1 neg)

(appendo res-cl1 res-cl2 res-cl)

(cond^a

[(tautologyo res-cl) fail]

[succeed]]))

(define *tautologyo*

(lambda (*cl*)

(fresh (*lit1 lit2 neg*)

(membero lit1 cl)

(membero lit2 cl)

(negateo lit1 neg)

(≡ neg lit2))

Subsumption

- Many generated clauses don't help us towards a proof, and we would like to throw these out
- A clause C subsumes a clause D if C logically implies D
- We want to avoid generating clauses that are subsumed by another clause already present in our set
- When we generate a clause that subsumes a clause already in our set, we would like to remove the subsumed clause from the set
- In propositional logic, C subsumes D if $C \subseteq D$
- In first-order logic, C subsumes D if there exists a θ such that $C\theta \subseteq D$

Subsumption

```
(define instantiate-termo
  (lambda (v1 v2)
    (project (v1 v2)
      (lambdag@ (p)
        (( $\equiv$  v2 (reify v1 p)) p))))))
```

```
(define subsumes
  (lambda (x y)
    (fresh (a d copy inst)
      (conde
        [( $\equiv$  '() x)]
        [( $\equiv$  '(,a . ,d) x)
          (copy-termo a copy)
          (instantiate-termo y inst)
          (membero copy inst)
          (subsumes d y)]))))))
```

Subsumption

```
(define remove-subsumptions
  (lambda (res-cl cls clsp)
    (fresh (a d res c1 c2)
      (conde
        [(≡ cls '()) (≡ clsp '())]
        [(≡ '(,a . ,d) cls)
         (conda
           [(fresh ()
                (copy-termo res-cl c1)
                (copy-termo a c2)
                (subsumes c1 c2))
            (remove-subsumptions res-cl d clsp)]
           [(≡ '(,a . ,res) clsp)
            (remove-subsumptions res-cl d res)]])]))))
```

Subsumption

```
(define res-step
  (lambda (cls out)
    (fresh (cl1 cl2 res-cl scl new-cls)
      (conde
        [(membero cl1 cls)
         (membero cl2 cls)
         (resolve cl1 cl2 res-cl)
         (conda
           [(fresh () (membero scl cls) (subsumes scl res-cl)) fail]
           [succeed])
         (remove-subsumptions res-cl cls new-cls)
         (≡ '(,res-cl . ,new-cls) out)]
        [(membero cl1 cls)
         (factor cl1 res-cl)
         (≡ '(,res-cl . ,cls) out))])))
```

Conclusion

- Simple resolution provers are easy to write and (hopefully) easy to understand
- Many performance-enhancing refinements are also easy to implement, and can be combined
- *copy-term* and several uses of **cond**^a are (so far) impossible to avoid
- A lot of work is still duplicated, even with refinements
- Eliminating all duplication of work requires giving up the simple structure of the prover