

Logic Programming

How to use logic as a programming language

Example languages

- Prolog
- Gödel (adds types)
- Mercury (adds functional programming)
- ...

The 5th generation project in Japan in the 1980s.

Rule-based systems.

Propositional logic (text 10.7.1) just truth and falsity, not very expressive.

Predicate calculus a.k.a. **First-order logic** (text 10.7.2) facts about some universe of discourse. Can use as basis for programming languages.

Acknowledgments

Some of the following material appears in “Truth, Deduction, and Computation” by Ruth E. Davis.

Some of the following material is from notes on lectures by Stuart Shieber on logic programming. Other material is copied verbatim from Shieber’s handouts.

Minor amounts of material are influenced by notes on lectures by Gerald Sachs on mathematical logic.

1

2

Semantics of Propositional Calculus

syntactic domains	semantic domains
variables V	{truth, falsity}
propositions P	$\phi \rightarrow$ {truth, falsity}

Grammar: $P ::= V \mid \neg P \mid P \rightarrow P \mid (P)$
 $V ::= p \mid q \mid r \mid s \mid \dots$

Valuation function $[[\cdot]]_\phi$ relative to interpretation ϕ
 ϕ : variables \rightarrow {truth, falsity}

V ranges over variables, P, P_1, P_2, Q range over propositions

$\mathcal{V}[[V]]_\phi$	$= \phi(V)$	
$\mathcal{P}[[V]]_\phi$	$= \mathcal{V}[[V]]_\phi$	
$\mathcal{P}[[\neg P]]_\phi$	$= \begin{cases} \text{truth if } \mathcal{P}[[P]]_\phi = \text{falsity} \\ \text{falsity if } \mathcal{P}[[P]]_\phi = \text{truth} \end{cases}$	“not”
$\mathcal{P}[[P_1 \rightarrow P_2]]_\phi$	$= \begin{cases} \text{truth if } \mathcal{P}[[P_2]]_\phi = \text{truth} \\ \text{truth if } \mathcal{P}[[P_1]]_\phi = \text{falsity} \\ \text{falsity otherwise} \end{cases}$	“implies”
$\mathcal{P}[[(P)]]_\phi$	$= \mathcal{P}[[P]]_\phi$	

Syntactic sugar:

abbreviation	called	expansion
$P \wedge Q$	“and”	$\neg(P \rightarrow (\neg Q))$
$P \vee Q$	“or”	$((\neg P) \rightarrow Q)$
$P \leftrightarrow Q$	“equivalent to”	$(P \rightarrow Q) \wedge (Q \rightarrow P)$

3

More semantics of Propositional Calc.

Add abbreviations to the language: give denotational definitions.

Claim \wedge really means “and”.

$\mathcal{P}[[P \wedge Q]]_\phi \equiv \mathcal{P}[[\neg(P \rightarrow (\neg Q))]]_\phi$	
by \neg	$= \begin{cases} \text{falsity if } \mathcal{P}[[P \rightarrow (\neg Q)]]_\phi = \text{truth} \\ \text{truth if } \mathcal{P}[[P \rightarrow (\neg Q)]]_\phi = \text{falsity} \end{cases}$
by \rightarrow	$= \begin{cases} \text{falsity if } \mathcal{P}[[\neg Q]]_\phi = \text{truth} \\ \text{falsity if } \mathcal{P}[[P]]_\phi = \text{falsity} \\ \text{truth otherwise} (\mathcal{P}[[P]]_\phi = \text{truth and } \mathcal{P}[[\neg Q]]_\phi = \text{falsity}) \end{cases}$
by \neg	$= \begin{cases} \text{falsity if } \mathcal{P}[[Q]]_\phi = \text{falsity} \\ \text{falsity if } \mathcal{P}[[P]]_\phi = \text{falsity} \\ \text{truth otherwise} (\mathcal{P}[[P]]_\phi = \text{truth and } \mathcal{P}[[Q]]_\phi = \text{truth}) \end{cases}$

Exercise: by same method, show

$\mathcal{P}[[(\neg P) \rightarrow Q]]_\phi = \text{truth if}$
 $\mathcal{P}[[P]]_\phi = \text{truth or } \mathcal{P}[[Q]]_\phi = \text{truth}$

More syntactic sugar: constants

$\text{true} \equiv P \rightarrow P$
 $\text{false} \equiv \neg(P \rightarrow P)$

Precedence: \neg , then \wedge , then \vee , then \rightarrow , then \leftrightarrow .

4

Definitions and Theorems

Definitions

- if $\mathcal{P} \llbracket P \rrbracket_\phi = \text{truth}$ then we say “ ϕ **satisfies** P ” and write “ $\phi \models P$ ”.
- if there is *some* ϕ for which $\mathcal{P} \llbracket P \rrbracket_\phi = \text{truth}$ then we say “ P is **satisfiable**”.
- if for all ϕ , $\mathcal{P} \llbracket P \rrbracket_\phi = \text{truth}$ then we say “ P is **valid**” and write $\models P$
- if P is not satisfiable (i.e. there is no ϕ such that $\mathcal{P} \llbracket P \rrbracket_\phi = \text{truth}$) then we say “ P is **unsatisfiable**” sometimes written $\not\models P$

Trivial theorems

- P is valid if and only if $\neg P$ is unsatisfiable.
- If $P \rightarrow Q$ is valid and Q is unsatisfiable, then P is unsatisfiable.

Proofs: unwind definitions.

Predicate Calculus (First Order Logic)

add richer set of meanings: \mathcal{U} “**universe of discourse**”
 function symbols
 predicate symbols } collectively called “**functors**”
 quantification

syntactic domains		semantic domains
constant symbols c	a, b, c	Objects in \mathcal{U}
function symbols f_i	f, g, h	Functions: $\mathcal{U} \times \dots \times \mathcal{U} \rightarrow \mathcal{U}$
predicate symbols p_i	p, q	Predicates in $\mathcal{U} \times \dots \times \mathcal{U}$
variables v	x, y, z	Objects in \mathcal{U}
formulas Wff	P, Q	$\phi \rightarrow \{\text{truth, falsity}\}$

Grammar: Term ::= $c \mid v \mid f_i(\text{Term}_1, \dots, \text{Term}_i)$
 Wff ::= $p_i(\text{Term}_1, \dots, \text{Term}_i)$
 $\mid \neg \text{Wff} \mid \text{Wff} \rightarrow \text{Wff} \mid (\text{Wff})$
 $\mid (\forall v) \text{Wff}$

Syntactic sugar:

abbreviation	called	expansion
$P \wedge Q$	“and”	$\neg(P \rightarrow (\neg Q))$
$P \vee Q$	“or”	$((\neg P) \rightarrow Q)$
$P \leftrightarrow Q$	“equivalent to”	$(P \rightarrow Q) \wedge (Q \rightarrow P)$
$(\exists x)P$	“exists”	$\neg(\forall x)\neg P$

Precedence: \neg , then \wedge , then \vee , then \rightarrow , then \leftrightarrow , then \forall , \exists .

Predicate Calculus terminology

- a Wff is called a “**well-formed formula**”, “**formula**”, or “**sentence**”.
- $p_i(t_1, \dots, t_i)$ is called an “**atomic formula**”
- an atomic formula is also called a “**positive literal**”
- $\neg p_i(t_1, \dots, t_i)$ is called a “**negative literal**”
- positive literals and negative literals are both called “**literals**”
- variables, constant symbols, functions symbols, and predicate symbols are all called “**primitive symbols**”

Remembering relations

An n -ary relation is a set of n -tuples

- Example: a binary relation, the “**diagonal relation**” on the natural numbers: $\{(0,0), (1,1), (2,2), \dots\}$
- Example: a binary relation, “ \leq ” on the natural numbers $\{(0,0), (0,1), (1,1), (0,2), (1,2), (2,2), (0,3), \dots\}$
- Example: a unary relation $\{2, 3, 5, 7, 11, \dots\}$ the prime numbers.
- Example: a ternary relation “**addition**” $\{(0,0,0), (0,1,1), \dots (1,1,2), (1,2,3), \dots (2,2,4), \dots\}$

The **characteristic function** of an n -ary relation maps n -tuples to $\{\text{truth, falsity}\}$.

Example: the characteristic function for the \leq relation sends $(1,2)$ to truth, but $(3,2)$ to falsity.

Predicate calculus semantics is based on the characteristic functions of relations: the meaning of a predicate symbol p_k is a relation. The meaning of $p_k(t_1, \dots, t_k)$ is “is the k -tuple of meanings of terms in the relation p_k ?”

Predicate Calculus semantics

Valuation functions $\llbracket \cdot \rrbracket$ relative to an interpretation ϕ of primitive symbols (variables, constants, function symbols, predicate symbols).

ϕ : constants $\rightarrow \mathcal{U}$

ϕ : variables $\rightarrow \mathcal{U}$

ϕ : n-ary function symbols $\rightarrow (\mathcal{U} \times \dots \times \mathcal{U} \rightarrow \mathcal{U})$

ϕ : n-ary predicate symbols $\rightarrow \mathcal{U} \times \dots \times \mathcal{U}$

$\mathcal{T} \llbracket e \rrbracket_\phi = \phi(e)$ for e a primitive symbol

$\mathcal{T} \llbracket f(t_1, \dots, t_n) \rrbracket_\phi = \mathcal{T} \llbracket f \rrbracket_\phi(\mathcal{T} \llbracket t_1 \rrbracket_\phi, \dots, \mathcal{T} \llbracket t_n \rrbracket_\phi)$

function application in \mathcal{U}

$\llbracket P(t_1, \dots, t_n) \rrbracket_\phi = \begin{cases} \text{truth if } (\mathcal{T} \llbracket t_1 \rrbracket_\phi, \dots, \mathcal{T} \llbracket t_n \rrbracket_\phi) \in \mathcal{T} \llbracket P \rrbracket_\phi \\ \text{falsity otherwise} \end{cases}$

$\llbracket \neg P \rrbracket_\phi = \begin{cases} \text{truth if } \llbracket P \rrbracket_\phi = \text{falsity} \\ \text{falsity if } \llbracket P \rrbracket_\phi = \text{truth} \end{cases}$

$\llbracket P_1 \rightarrow P_2 \rrbracket_\phi = \begin{cases} \text{truth if } \llbracket P_2 \rrbracket_\phi = \text{truth} \\ \text{truth if } \llbracket P_1 \rrbracket_\phi = \text{falsity} \\ \text{falsity otherwise} \end{cases}$

$\llbracket (P) \rrbracket_\phi = \llbracket P \rrbracket_\phi$

$\llbracket (\forall x)P \rrbracket_\phi = \begin{cases} \text{truth if for all } \phi' \in I(\phi, x), \llbracket P \rrbracket_{\phi'} = \text{truth} \\ \text{falsity otherwise} \end{cases}$

$I(\phi, x)$ is the set of interpretations defined as

$I(\phi, x) = \{\psi \mid \psi(v) = \phi(v) \text{ for } v \neq x\}$ i.e. all

interpretations identical to ϕ except perhaps at x .

9

Example interpretation: $1+1=2$

$s(\mathbf{0}) + s(\mathbf{0}) = s(s(\mathbf{0}))$ or $= (+(s(\mathbf{0}), s(\mathbf{0})), s(s(\mathbf{0})))$

$\mathbf{0} \in c, \quad x, y, z \in v, \quad s \in f_1, \quad + \in f_2, \quad = \in p_2$

Select \mathcal{U} to be the non-negative integers, select interpretation ϕ_0 to be

$\phi_0 : = \mapsto id$ (the diagonal relation over \mathcal{U})

$s \mapsto succ$ (the successor function)

$+ \mapsto +$ (the addition function)

$\mathbf{0} \mapsto 0$

$x, y, z \mapsto 5$

$\llbracket =(+ (s(\mathbf{0}), s(\mathbf{0})), s(s(\mathbf{0}))) \rrbracket_{\phi_0}$

$= (\llbracket + (s(\mathbf{0}), s(\mathbf{0})) \rrbracket_{\phi_0}, \llbracket s(s(\mathbf{0})) \rrbracket_{\phi_0}) \in \llbracket = \rrbracket_{\phi_0}$

$= (\llbracket + (s(\mathbf{0}), s(\mathbf{0})) \rrbracket_{\phi_0}, \llbracket s(s(\mathbf{0})) \rrbracket_{\phi_0}) \in id$

$= \dots$

$= (\llbracket + \rrbracket_{\phi_0}(\llbracket s \rrbracket_{\phi_0}(\llbracket \mathbf{0} \rrbracket_{\phi_0}), \llbracket s \rrbracket_{\phi_0}(\llbracket \mathbf{0} \rrbracket_{\phi_0})), \llbracket s \rrbracket_{\phi_0}(\llbracket s \rrbracket_{\phi_0}(\llbracket \mathbf{0} \rrbracket_{\phi_0})) \in id$

$= \dots$

$= (1 + 1, 2) \in id$

$= (2, 2) \in id$

$= \text{truth}$

10

Another example interpretation

$(\forall x) \text{even}(x) \rightarrow \text{odd}(x + 1)$

$\mathbf{1} \in c, \quad x \in v, \quad + \in f_2, \quad \text{even}, \text{odd} \in p_1$

Select \mathcal{U} to be the non-negative integers, select interpretation ϕ_0 to be

$\phi_0 : \mathbf{1} \mapsto \mathbf{1}$

$x \mapsto \mathbf{0}$

$+ \mapsto +$ (the addition function)

$\text{even} \mapsto \{0, 2, 4, \dots\}$ (a unary relation)

$\text{odd} \mapsto \{1, 3, 5, \dots\}$ (a unary relation)

$\llbracket (\forall x) \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi_0} = \text{truth}$ iff for all $\phi' \in I(\phi_0, x)$,

$\llbracket \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

Proof is by cases on ϕ' . Suppose that $\phi'(x) = i$ is an even number, show that the formula holds for all such ϕ' :

$\llbracket \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either $\llbracket \text{even}(x) \rrbracket_{\phi'} = \text{falsity}$ or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either $\llbracket x \rrbracket_{\phi'} \notin \llbracket \text{even} \rrbracket_{\phi'}$ or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either i is not an even number or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff $\llbracket x + 1 \rrbracket_{\phi'} \in \llbracket \text{odd} \rrbracket_{\phi'}$

iff $(i + 1) \in \llbracket \text{odd} \rrbracket_{\phi'}$

iff $i + 1$ is an odd number

11

Another example (continued)

Previous slide: holds for ϕ' such that $\phi'(x) = i$ is even since then $i + 1$ is odd.

Now consider the other possible ϕ' s: in these $\phi'(x)$ is odd:

$\llbracket \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either $\llbracket \text{even}(x) \rrbracket_{\phi'} = \text{falsity}$ or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either $\llbracket x \rrbracket_{\phi'} \notin \llbracket \text{even} \rrbracket_{\phi'}$ or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

iff either i is not an even number or $\llbracket \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$

Now, by hypothesis, i is not an even number, so again

$\llbracket \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$.

This exhausts all possible cases for ϕ' . In every case we have $\llbracket \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi'} = \text{truth}$ so

$\llbracket (\forall x) \text{even}(x) \rightarrow \text{odd}(x + 1) \rrbracket_{\phi_0} = \text{truth}$

12

Intended interpretations

Back to $P \equiv s(\underline{0}) + s(\underline{0}) = s(s(\underline{0}))$

take \mathcal{U} to be the same but ϕ_1 to be

$\phi_1 : \begin{array}{l} = \mapsto id \text{ (the diagonal relation over } \mathcal{U} \text{)} \\ + \mapsto + \text{ (the addition function)} \\ \underline{0} \mapsto 1 \\ s \mapsto \textit{square} \text{ (the squaring function)} \\ x, y, z \mapsto 5 \end{array}$

$\phi_0 \models P$ since $1 + 1 = 2$

$\phi_1 \not\models P$ since $1^2 + 1^2 \neq (1^2)^2$

To program in logic, would like to know intended interpretation.

Can not tell computer ϕ_0 since ϕ_0 is “real”, but computer manipulates symbols.

Next best thing: **theories** use more Wff’s to tell computer enough to get intended answer.

Notation: use Γ as a symbol for a theory.

13

Some theories

Theory of equality

$(\forall x)x=x$
 $(\forall x)(\forall y)x=y \rightarrow y=x$
 $(\forall x)(\forall y)(\forall z)x=y \wedge y=z \rightarrow x=z$

Theory of equality of addition given theory of equality

$(\forall x)\underline{0}+x=x$
 $(\forall x)(\forall y)s(x)+y=s(x+y)$

Note: this is not a theory of addition. Addition is a 2-ary function here and equality is the predicate. A theory of addition would be about the 3-ary addition predicate shown on the slide on relations.

Validity relative to a theory

We say $\Gamma \models P$ “ Γ validates P ” if for all ϕ such that $\phi \models \wedge \Gamma$, $\phi \models P$

if ϕ validates all of the sentences in the theory Γ , then ϕ validates the sentence P .

Equality, equality of addition $\models s(\underline{0}) + s(\underline{0}) = s(s(\underline{0}))$

14

Computing with logic

Question: How can we compute that $\Gamma \models P$?

- exhaustion
- deduction
- resolution refutation

Exhaustion:

Look at every interpretation ϕ . If $\phi \models \wedge \Gamma$ then check if $\phi \models P$.

This is method of **truth tables** in propositional calculus.

Does not generally work in first order logic: quantification over an infinite universe...

15

Deduction

Use rules to prove P from Γ ($\Gamma \vdash P$)

Γ rewrites to P , using some rules.

desirable properties:

soundness if $\Gamma \vdash P$ then $\Gamma \models P$

completeness if $\Gamma \models P$ then $\Gamma \vdash P$

Some sound rules of inference (but not a complete set)

$$\frac{P \rightarrow Q, P}{Q}$$
 Modus ponens

$$\frac{P \rightarrow Q, \neg Q}{\neg P}$$
 Modus tollens

$$\frac{(\forall x)P}{P[a/x]}$$
 Universal instantiation

16

Deduction example

Theory of ancestry for the family of Bertrand Russel.

Axioms (1) and (2) constitute a general theory of ancestry, (1) being transitivity of ancestry, and (2) being a basis for ancestry. Axioms (3) and following constitute a specific theory of parentage for the Russel family.

1. $(\forall x)(\forall y)anc(x,y) \rightarrow ((\forall z)anc(y,z) \rightarrow anc(x,z))$
2. $(\forall x)(\forall y)parent(x,y) \rightarrow anc(x,y)$
3. $parent(bertrand,kate)$
4. $parent(amberly,bertrand)$
5. etc.

The intended interpretation is that $anc(x,y)$ holds if and only if x denotes an ancestor of y 's denotation. The atom $parent(x,y)$ holds if and only if x denotes a parent of y 's denotation. The constants $bertrand$, $amberly$, and $kate$ denote Bertrand Russell, his father Amberly, and his daughter Katherine Tait, respectively.

Use these theories plus Modus Ponens (MP) and Universal Instantiation (UI) to construct a deduction proof that Amberly Russell is an ancestor of Katherine Tait.

17

Deduction example II

1. $(\forall x)(\forall y)anc(x,y) \rightarrow ((\forall z)anc(y,z) \rightarrow anc(x,z))$ so
6. $anc(amb,bert) \rightarrow ((\forall z)anc(bert,z) \rightarrow anc(amb,z))$ by (1) and (UI)
2. $(\forall x)(\forall y)parent(x,y) \rightarrow anc(x,y)$ so
7. $parent(amb,bert) \rightarrow anc(amb,bert)$ by (2) and (UI)
4. $parent(amb,bert)$ so:
8. $anc(amb,bert)$ by (7), (4), and (MP)
9. $(\forall z)anc(bert,z) \rightarrow anc(amb,z)$ by (6), (8) and (MP)
10. $anc(bert,kate) \rightarrow anc(amb,kate)$ by (9) and (UI)
2. $(\forall x)(\forall y)parent(x,y) \rightarrow anc(x,y)$ so:
11. $parent(bert,kate) \rightarrow anc(bert,kate)$ by (2) and (UI)
3. $parent(bert,kate)$ so:
12. $anc(bert,kate)$ by (11), (3) and (MP) so:
13. $anc(amb,kate)$ by (8), (12) and (MP)

Q.E.D.

(Proof tree does not fit readably on slide)

18

Asides on deduction

Theorem (Gödel, 1930): **Completeness**

$\Gamma \models P$ if and only if $\Gamma \vdash P$

A theory validates a formula exactly when the formula can be proven using the theory and the axioms and rules of logical deduction.

Incompleteness is more famous, but not much use to us:

Theorem (Gödel, 1931): **Incompleteness**

If you can define an encoding of true sentences about \mathcal{U} and if the encodings are representable as elements of \mathcal{U} then there is some true sentence about \mathcal{U} that can not be deduced from the encoded sentences.

Deduction can be automated "Proof assistants" a.k.a. "Automated theorem provers"

There may be lots of proofs for a given formula: proof assistants need human guidance.

Typed λ -calculus can be used to encode the shape of proofs. Reduction in typed λ -calculus can be used to simplify proofs. This was original use of typed λ -calculus before programming languages

19

Resolution Refutation

(Resolution is an algorithm, refutation is what you are trying to use it for).

Observation: $\Gamma \models P$ if and only if $(\bigwedge \Gamma) \wedge \neg P$ is unsatisfiable.

The "only if" direction: Say $\Gamma = Q_1, \dots, Q_n$. Divide up all possible ϕ into (1) those that make all Q_i true and make P true and (2) those that make some Q_i false and make P false. (These are only two categories of ϕ s by def'n of Validity relative to a theory.) In the first group of ϕ s, $\neg P$ is false. In the second group of ϕ s, some Q_i false. So, in both cases (i.e. for all ϕ) $[(\bigwedge \Gamma) \wedge \neg P]_{\phi} = \text{falsity}$ – which is the definition of unsatisfiable.

Technique Resolution theorem proving

1. **Construct** $(\bigwedge \Gamma) \wedge \neg P$
2. **Convert** to clausal form
3. **Resolve** literals in clauses to form more clauses (resolution)
4. **Generate** $L \wedge \neg L$ a contradiction (refutation)

20

Clausal form

A formula is in **prenex form** if every variable is quantified and all quantifiers precede a quantifier-free sentence.

All quantifiers \equiv the **prefix**

Quantifier-free sentence \equiv the **matrix**

A formula is in **clausal form** if

- it is in prenex form
- the only quantifier is \forall
- the matrix is in **conjunctive normal form**
 $(clause_1 \wedge \dots \wedge clause_n)$
 where each clause has the form
 $(literal_{i,1} \vee \dots \vee literal_{i,m_i})$
 and every literal has the form *atomic-formula* or \neg *atomic-formula*.

Conversion to clausal form

Step 1: Rewrite to use only $\wedge, \vee, \neg, \forall,$ and \exists

$$P \leftrightarrow Q \text{ rewrites to } (P \rightarrow Q) \wedge (Q \rightarrow P)$$

$$P \rightarrow Q \text{ rewrites to } \neg P \vee Q$$

Step 2: Rewrite to move quantifiers outwards

$$\neg(\exists x)P \text{ rewrites to } (\forall x)\neg P$$

$$\neg(\forall x)P \text{ rewrites to } (\exists x)\neg P$$

$$(\wedge \dots (\forall x)P \dots) \text{ rewrites to } (\forall x')(\wedge \dots P[x'/x] \dots)$$

$$(\vee \dots (\forall x)P \dots) \text{ rewrites to } (\forall x')(\vee \dots P[x'/x] \dots)$$

$$(\wedge \dots (\exists x)P \dots) \text{ rewrites to } (\exists x')(\wedge \dots P[x'/x] \dots)$$

$$(\vee \dots (\exists x)P \dots) \text{ rewrites to } (\exists x')(\vee \dots P[x'/x] \dots)$$

α -convert when needed to keep from capturing variables when expanding scope.

Can pick \forall and \exists in any order to move out of \wedge and \vee .

Example:

$$(\forall x)(\neg(x=0) \rightarrow (\exists y)(y+s(0)=x))$$

$$\Rightarrow (\forall x)(\neg(\neg(x=0)) \vee (\exists y)(y+s(0)=x))$$

$$\Rightarrow \underbrace{(\forall x)(\exists y)}_{\text{prefix}} \underbrace{(\neg(\neg(x=0)) \vee (y+s(0)=x))}_{\text{matrix}}$$

21

22

Conversion to clausal form 2

Converting matrix to C.N.F. using laws for rewriting quantifier-free logic.

push \neg in

$$\neg(\neg P) \text{ rewrites to } P$$

$$\neg(P \wedge Q) \text{ rewrites to } \neg P \vee \neg Q$$

$$\neg(P \vee Q) \text{ rewrites to } \neg P \wedge \neg Q$$

distribute or's into and's

$$P \vee (Q \wedge R) \text{ rewrites to } (P \vee Q) \wedge (P \vee R)$$

Example continued:

$$(\forall x)(\exists y) (\neg(\neg(x=0)) \vee (y+s(0)=x))$$

$$\Rightarrow (\forall x)(\exists y) (x=0 \vee (y+s(0)=x))$$

Conversion to clausal form 3

Getting rid of \exists (Skolemization)

Each \exists -bound variable is replaced by a function of the \forall -bound variables outside the \exists binding. If n \forall bindings, then a n -ary function. A 0-ary function is a constant.

Example continued:

$$(\forall x)(\exists y) (x=0 \vee (y+s(0)=x))$$

$$\Rightarrow (\forall x) (x=0 \vee (\text{pred}(x)+s(0)=x))$$

Notation: Clausal form vs. clause sets

Clausal form

$$(\forall x_1) \dots (\forall x_k) (I_{1,1} \vee \dots \vee I_{1,n_1})$$

$$\wedge (I_{2,1} \vee \dots \vee I_{2,n_2})$$

$$\vdots$$

$$\vdots$$

$$\wedge (I_{m,1} \vee \dots \vee I_{m,n_m})$$

If we can tell variables from constants, then don't need to write out \forall binders. Can just make a **set** of disjuncts (clauses):

$$(I_{1,1} \vee \dots \vee I_{1,n_1})$$

$$(I_{2,1} \vee \dots \vee I_{2,n_2})$$

$$\vdots$$

$$(I_{m,1} \vee \dots \vee I_{m,n_m})$$

23

24

Conversion to clausal form: summary

$(\forall x)(\neg(x=0) \rightarrow (\exists y)(y+s(0)=x))$ **original**
 $\Rightarrow (\forall x)(\neg(\neg(x=0)) \vee (\exists y)(y+s(0)=x))$ **rewrite implications**
 $\Rightarrow (\forall x)(\exists y)(\neg(\neg(x=0)) \vee (y+s(0)=x))$ **move quantifiers out**
 $\Rightarrow (\forall x)(\exists y)(x=0) \vee (y+s(0)=x)$ **push \neg in, distribute \wedge over \vee**
 $\Rightarrow (\forall x)(x=0) \vee (\text{pred}(x)+s(0)=x)$ **Skolemize clausal form**
Clause set corresponding to clausal form
 $(x=0) \vee (\text{pred}(x)+s(0)=x)$ **clause set**

1. Construct $(\wedge \Gamma) \wedge \neg P$
2. Convert to clausal form **done**
3. Resolve literals **next**
4. Generate \square

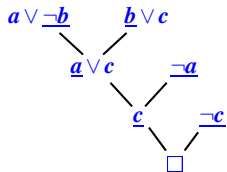
25

Example

Let a, b, c , be constants.

Show that $b \rightarrow a, \neg c \rightarrow b, \neg a \models c$

1. formula to refute: $(b \rightarrow a) \wedge (\neg c \rightarrow b) \wedge (\neg a) \wedge (\neg c)$
2. clausal form: $(a \vee \neg b) \wedge (b \vee c) \wedge (\neg a) \wedge (\neg c)$
3. resolve clauses:



4. got \square ?
Yes, so formula refuted, so the theory
 $b \rightarrow a, \neg c \rightarrow b, \neg a$ validates c .

Time for an exercise?

27

Resolution

Consider two clauses containing complementary literals: $L_1 \vee P$ and $\neg L_1 \vee Q$. Their **resolvent** is $P \vee Q$.

Write

$$\frac{L_1 \vee P \quad \neg L_1 \vee Q}{P \vee Q}$$

Repeat until

$$\frac{L_n \quad \neg L_n}{\square}$$

Read \square as “the empty clause”. Since there is nothing in the empty clause that can be made true, it is just a constant with the denotation *falsity*.

Why should resolution work? If there is an unsatisfiable clause set containing $L_1 \vee P$ and $\neg L_1 \vee Q$ then we want to claim that the clause set with $L_1 \vee P$ and $\neg L_1 \vee Q$ removed and $P \vee Q$ added is unsatisfiable.

Generalization of Modus ponens, which says if A and $B \vee \neg A$ then B . For proof see Davis “Truth Deduction and Computation”.

Resolution preserves unsatisfiability

26

Handling variables

Problem: We need to resolve on complementary literals.

Can we resolve

$$\frac{p(x,c) \vee P \quad \neg p(a,y) \vee Q}{P \vee Q}$$

$p(x,c)$ and $p(a,y)$ are not the same literal: constant c is not variable y , constant a is not variable x .

But we are trying to show formula unsatisfiable: if **no** ϕ makes formula true, then a ϕ such that $\phi(x) = \phi(a)$ and $\phi(y) = \phi(c)$ will not make the formula true.

That is not enough, but read backwards: all variables are universally quantified, so if a ϕ such that $\phi(x) = \phi(a)$ and $\phi(y) = \phi(c)$ makes the formula true, then some ϕ makes the formula true. Therefore, it was not unsatisfiable after all.

Solution: solve problem mechanically by building a **unifier** $\sigma = \{(x,a), (y,c)\}$. Extend $\sigma : \text{Variable} \mapsto \text{Term}$ to $\hat{\sigma} : \text{Term} \mapsto \text{Term}$, which applies σ to every *Variable* in the *Term*.

Unification preserves unsatisfiability

28

Handling Variables II

Problem: in trying resolution refutation to show

$anc(amb, kate)$

We want to resolve

$$\frac{\frac{\neg anc(amb, bert) \quad anc(x, y) \vee \neg parent(x, y)}{\neg parent(amb, bert)}}{\neg parent(amb, bert)}$$

and

$$\frac{\frac{\neg anc(bert, kate) \quad anc(x, y) \vee \neg parent(x, y)}{\neg parent(bert, kate)}}{\neg parent(bert, kate)}$$

but under substitution $\{(x, amb), (y, bert)\}$ there are no complementary literals in the second case. Under substitution $\{(x, bert), (y, kate)\}$ there are no complementary literals in the first case.

Solution: Standardize apart. Before using any clause with variables, apply a substitution that replaces the variables with fresh variables.

$$\theta_1 = \{(x, x'), (y, y')\}, \quad \theta_2 = \{(x, x''), (y, y'')\}, \\ \sigma_1 = \{(x', amb), (y', bert)\}, \quad \sigma_2 = \{(x'', bert), (y'', kate)\}$$

In resolution theorem proving, we want a single subst for variables in goal clause, but standardize apart variables in theory.

Resolution with unification

To resolve $P = P_1 \vee P_2 \vee \dots \vee L \vee \dots \vee P_n$ with

$Q = Q_1 \vee Q_2 \vee \dots \vee \neg L' \vee \dots \vee Q_m$, where $L \equiv P_i$ and $\neg L' \equiv Q_j$

1. **standardize apart**, creating some substitution θ for variables in Q .
2. **find complementary literals**, up to variables, L in P and $\hat{\theta}(\neg L')$ in $\hat{\theta}(Q)$ and create substitution $\sigma =$ most general unifier of L and $\hat{\theta}(\neg L')$.
3. **resolve** to generate $\hat{\sigma}(P-L) \vee \hat{\sigma}\hat{\theta}(Q-\neg L')$
 $\equiv \hat{\sigma}(P_1) \vee \dots \vee \hat{\sigma}(P_{i-1}) \vee \hat{\sigma}(P_{i+1}) \vee \dots \vee \hat{\sigma}(P_n) \vee$
 $\hat{\sigma}\hat{\theta}(Q_1) \vee \dots \vee \hat{\sigma}\hat{\theta}(Q_{j-1}) \vee \hat{\sigma}\hat{\theta}(Q_{j+1}) \vee \dots \vee \hat{\sigma}\hat{\theta}(Q_m)$

In resolution theorem proving we know that we are going to standardize apart, so we often reuse variable names between different clauses in a clause set – which would change the meaning of the theory if it were actually in clausal form with explicit quantifiers in the matrix: See reuse of x, y on next slide.

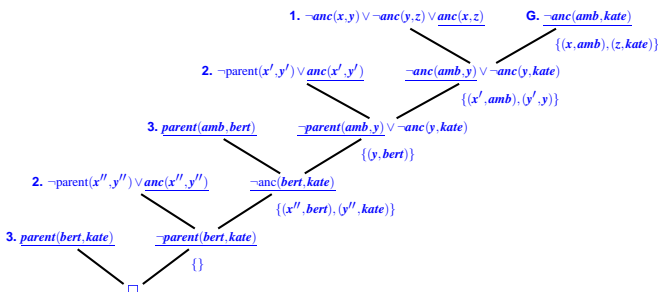
29

30

Resolution proof of $anc(amb, kate)$

Theory of ancestry as clause set:

1. $\neg anc(x, y) \vee \neg anc(y, z) \vee anc(x, z)$ (transitivity of ancestry)
2. $\neg parent(x, y) \vee ancestor(x, y)$ (basis for ancestry)
3. $parent(bertrand, kate)$
4. $parent(amberly, bertrand)$

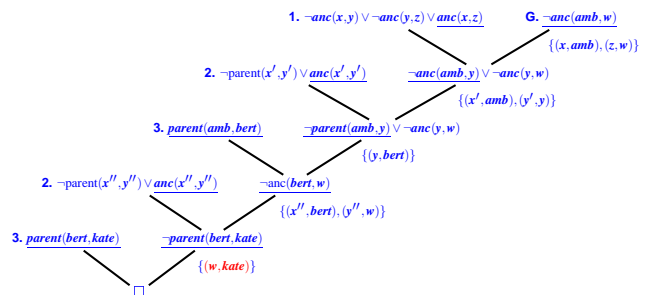


All this for yes/no answer?

Unification allows constructive proof of existentials.

Take as goal $(\exists w)anc(amb, w)$.

Negated goal is $(\forall w)\neg anc(amb, w)$. Run through similar proof:



Resolution refutation has provided a substitution $\{(w, kate)\}$ as the answer to the question “who is Amberly the ancestor of”. A similar resolution refutation would find that Amberly is the ancestor of Bertram.

Logic programming: use resolution refutation with unification to provide values of variables in goal.

31

32

Practical aspects

Resolution theorem proving is too slow.

1. What clauses to pick?
2. What literals to try to make complementary?
3. What instantiation of variables?

Key is to limit form of clauses:

Clause $L_1 \vee \dots \vee L_n \vee \neg M_1 \vee \dots \vee \neg M_m$

rewrites by deMorgan's laws as

$(M_1 \wedge \dots \wedge M_m) \rightarrow (L_1 \vee \dots \vee L_n)$

Multiple positive literals: only some need to be true for whole clause to be true.

A **Horn clause** is a clause with at most 1 positive literal.

A **Goal clause** has 0 positive literals

A **Definite clause** has 1 positive literal

A **Unit clause** is special case: 1 positive, 0 negative

Horn clauses are sufficient to write expressive programs.

33

Practical aspects

Theorem: Given a set of Horn clauses and a goal clause, we can limit the proof strategy by resolving the goal clause with a definite clause, creating a new goal clause. Once a particular goal clause has been used once it can be discarded. If the definite clause is a unit clause, then the proof is complete.

This method is called **SLD resolution**. **Selected, Linear, Definite resolution**

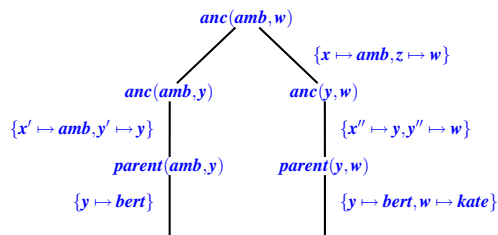
34

SLD resolution proof trees

Just show (sub)goals: At root write goal literal.

For each subtree, rewrite goal literal into new goal literal(s) that replace current goal.

If resolving with a definite clause with multiple negative literals, then there are multiple subtrees, one for each negative literal.



At root resolves with $\neg anc(x, y) \wedge \neg anc(y, z) \wedge anc(x, z)$

Left branch resolves subgoal with $\neg parent(x, y) \wedge anc(x, y)$
then with unit clause $parent(amb, bert)$

Right branch resolves subgoal with $\neg parent(x, y) \wedge anc(x, y)$
then with unit clause $parent(bert, kate)$

(See J.R. Fisher's "prolog :- tutorial" http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/pt_framer.html, "Prolog derivation trees and choices")

35

Prolog's choices

1. **What clauses to pick?** SLD: choose one clause to resolve against goal (as opposed to two clauses to resolve). Prolog: Appear to try clauses sequentially in the order in which they exist in the program, to find one with a complementary literal (Actually, use indexing rather than linear search.)
2. **What literals to try to make complementary?** SLD: if more than one matches, pick any matching literal to grow fringe of tree. Prolog: Appear to do depth-first left-to-right search in SLD proof tree for matching literals.
3. **What instantiation of variables?** SLD: Most general unifier. Prolog: Most general unifier – but not quite. Prolog tries to get a bit more speed out of omitting occurs check.

36

More notation!

Clause notation	Implication notation	Prolog
Definite clause: $L \vee \neg M_1 \vee \dots \vee \neg M_m$	$(M_1 \wedge \dots \wedge M_m) \rightarrow L$	$L :- M_1, \dots, M_m.$
Unit Clause: L	$\rightarrow L$	$L.$
Goal clause: $\neg M_1 \vee \dots \vee \neg M_m$	$(M_1 \wedge \dots \wedge M_m) \rightarrow$	$?- M_1, \dots, M_m.$

?- is actually a prompt, not part of goal clause.

Prolog variables are upper case, all other symbols are lower case.

Prolog example

```
%% anc(Old, Young)
%% -----
%%
anc(Old, Young) :- anc(Old, Mid), anc(Mid, Young).
anc(Old, Young) :- parent(Old,Young).
%%
%% parent(Parent,Child)
%% -----
%%
parent(amberly,bertrand).
parent(bertrand,kate).
```

etc.

Sample run:

```
| ?- anc(amberly,kate).
^C
Prolog interruption (h for help)? a
[ Execution aborted ]
```

We forgot clause selection order in Prolog!
Always looping back to get first clause of anc.

37

38

Prolog example fixed I

Switch order do checks parent before checks anc again.

```
%% anc(Old, Young)
%% -----
%%
anc(Old, Young) :- parent(Old,Young).
anc(Old, Young) :- anc(Old, Mid), anc(Mid, Young).
%%
%% parent(Parent,Child)
%% -----
%%
parent(amberly,bertrand).
parent(bertrand,kate).
```

```
| ?- anc(amberly,kate).
Yes
| ?- anc(amberly,X).
X = bertrand ;
X = kate ;
ERROR: out of local stack.
```

; asks for another answer (another proof).

Problem: literal selection order. When attempts a third proof, is looking for an anc(Old, Mid) who is not a parent(Old, Mid). Gets stuck looping on second anc clause.

39

Prolog example fixed II

Force anc to try parent first in all cases.

```
%% anc(Old, Young)
%% -----
%%
anc(Old, Young) :- parent(Old,Young).
anc(Old, Young) :- parent(Old, Mid), anc(Mid, Young).
%%
%% parent(Parent,Child)
%% -----
%%
parent(amberly,bertrand).
parent(bertrand,kate).
```

etc.

Sample run:

```
| ?- anc(amberly,kate).
Yes
| ?- anc(amberly,X).
X = bertrand ;
X = kate ;
No
```

When out of parents there is no more possible proof.

40

Summary

Quick development of **logic programming** from **denotational semantics** of first-order logic.

Can try to calculate by: exhaustion, deduction, or resolution refutation.

Of these **resolution refutation works best**. Requires logical statements in **clausal form** (or as a clause set).

Resolution refutation still needs restrictions for efficiency: **Horn clause logic**.

Prolog implements a strategy for resolution refutation in Horn clause logic. Prolog has an **operational semantics** that is more specific than the resolution refutation: Can go into infinite loops in cases where there is a proof, and in cases where there is no proof.

Understanding logic programming is useful for understanding Prolog, (and other logic-programming languages) but must understand operational semantics as well (just as needed to know if a functional language implemented normal order or applicative order).