Overview

• Logic is the study of correct arguments.
• Logic and computation are connected:

\[ \forall x. \exists y. P(x, y) \]

• If a proof of the claim is constructive, then for every \( x \) we can compute a \( y \)
Example

• *For every list* \(x\) *of integers, there is a list* \(y\) *with the same elements arranged in non-descending order*

• A constructive proof is an algorithm for sorting lists of integers

• There are proofs that are not constructive
  • Prove something is true, but don’t produce a “witness”, a thing exhibiting the truth of the statement
  • But that is a topic for a future lecture …
Logic Programming

• PROLOG
  • PROgramming in LOGic
  • Motivated by study of constructive reasoning
  • Most popular logic programming language

• Logic programming started in the ‘70’s
• Logic programming was big in the ‘80’s
  • 5th generation project (Japan)
• Many applications today in specialized domains
  • Databases, scheduling problems in transportation
PROLOG Basics

• PROLOG is a theorem prover
  • Consider a predicate \texttt{rev(x,y)}
  • \textit{“y is x reversed”}  
    • \texttt{rev([1,2,3],[3,2,1])} returns “true”

• More usefully \texttt{rev([1,2,3],y)} returns \texttt{true} and substitution \texttt{y=[3,2,1]}

• Intuitively, \texttt{x} is the input, \texttt{y} is the output
No Input/Output Distinction

• But logic programming is more general.
• y can be the input and x the output:
  • rev(x,[1,2,3]) returns “true” and x=[3,2,1]
• Or y and x both can be partially defined:
  • rev([1,2,a],[3,2,b]) returns true and a=3, b=1

• A computation attempts to satisfy a predicate by computing a substitution for the free variables
Syntax

• PROLOG has *terms* and *atoms*.

• A *term* is
  • a constant (e.g., 1 or nil)
  • a variable
  • $c(x,y,z)$ where
    • $c$ is a constructor (of the correct arity)
    • $x,y,z$ are terms

• An *atom* is a predicate applied to terms
  • $\text{rev}([1,2,3], y)$
Lists

• Lists have special syntax
• $\text{cons}(x, \text{cons}(y, \text{nil})) = [x, y]$
Programs

• A PROLOG program has \textit{facts} and \textit{rules}

• A rule has the form
  \[ P_1(t_{11}, \ldots) : - P_2(t_{21},\ldots),\ldots,P_n(t_{n1},\ldots) \]

• The meaning of a rule (or \textit{clause}) is
  \[ P_2(t_{21},\ldots) \land \ldots \land P_n(t_{n1},\ldots) \Rightarrow P_1(t_{11}, \ldots) \]

• A fact is a rule with no rhs. Facts are always true.
  \[ P_1(t_{11}, \ldots). \]
Reverse in PROLOG

addright(nil, X, [X]).
addright(cons(A,B), X, cons(A,Z)) :- addright(B,X,Z)

rev(nil, nil).
rev(cons(X,Y), Z) :- rev(Y,W), addright(W,X,Z)
Semantics

- Logic programming has a beautiful semantics.
- Let $\sigma$ range over all ground substitutions
  - Substitutions that map variables to terms with no variables in them
- Given a set of rules
  \[
  P_1(t_{11},\ldots) : \neg P_2(t_{21},\ldots), \ldots, P_n(t_{n1},\ldots)
  \]
- The semantics is the smallest set of atoms $F$ satisfying
  \[
  \{\sigma(P_2(t_{21},\ldots)), \ldots, \sigma(P_n(t_{n1},\ldots))\} \subseteq F \Rightarrow \sigma(P_1(t_{11},\ldots)) \in F
  \]
Semantics (Continued)

• This is the *Herbrand model*
  • after the Herbrand Universe, the set of all terms

• Note the semantics is defined bottom-up:
  • all facts are in $F$
  • any implication proven by atoms in $F$ is in $F$
Implementations

• Logic programming has
  • a very concise and well-defined semantics
  • implementations that do not follow the semantics

• Efficiency is a major problem in many logic programming languages

• Leads to compromises in implementations
PROLOG Implementation

• Start with simple things and work up.
• The following example is from Kamin’s book *Programming Languages: An Interpreter-Based Approach*

```
imokay :- youreokay, hesokay
youreokay :- theyreokay
hesokay.
theyreokay.
```
Execution

• **Rule:** Given a goal \( a \), find a rule whose left-hand side matches \( a \). Add the right-hand side atoms as subgoals

• Goal **imokay** yields true:
  • **imokay** matches **imokay** :- **youreokay**, **hesokay**
  • **youreokay**, **hesokay** are subgoals
  • Rule is applied recursively to subgoals
    • **youreokay** matches **youreokay** :- **theyreokay**
  • **hesokay** and **theyreokay** are both facts
imokay :- youreokay, hesokay

youreokay :- theyreokay

hesokay.

theyreokay.

\[ \Gamma \vdash \text{imokay} \]

\[ \Gamma \vdash \text{theyreokay} \]

\[ \Gamma \vdash \text{youreokay} \]

\[ \Gamma \vdash \text{hesokay} \]

\[ \Gamma \vdash \text{imokay} \]
Multiple Matches

• PROLOG works from goals towards facts.
  • Goals are replaced by subgoals according to the rules.

• What if more than one rule matches a goal?

• Add three rules to our program

imokay :- youreokay, hesokay
youreokay :- theyreokay
hesokay.
theyreokay.
hesnotokay :- imnotokay
shesokay :- hesnotokay
shesokay :- theyreokay
Rule Order and Backtracking

• *Refine Rule*: Select the first matching rule.
  • “first” means first textually
  • if a subgoal fails, select the next matching rule
  • if no matching rule is found, fail.

• This is backtracking
  • The first matching rule not already tried is always chosen
Example

• To prove shesokay:

  • Goal matches shesokay :- hesnotokay
    • Subgoal hesnotokay matches hesnotokay :- imnotokay
      • imnotokay fails (no matching rule), backtrack.
    • hesnotokay fails, backtrack

  • Goal matches shesokay :- theyreokay
    • theyreokay is a fact.

  imokay :- youreokay, hesokay
  youreokay :- theyreokay
  hesokay.
  theyreokay.
  hesnotokay :- imnotokay
  shesokay :- hesnotokay
  shesokay :- theyreokay
Example

?- imnotokay
   |- imnotokay

?- hesnotokay
   |- hesnotokay
   |- shesokay

?- shesokay
   |- shesokay

imokay :- youreokay, hesokay
youreokay :- theyreokay
hesokay.
theyreokay.
hesnotokay :- imnotokay
shesokay :- hesnotokay
shesokay :- theyreokay
Example

\[ \neg \text{theyreokay} \]

\[ \rightarrow \text{shesokay} \]

\[ \neg \text{imokay} \rightarrow \text{youreokay}, \text{hesokay} \]

\[ \text{youreokay} \rightarrow \text{theyreokay} \]

\[ \text{hesokay} \]

\[ \text{theyreokay} \]

\[ \neg \text{hesnotokay} \rightarrow \neg \text{imnotokay} \]

\[ \text{shesokay} \rightarrow \neg \text{hesnotokay} \]

\[ \text{shesokay} \rightarrow \text{theyreokay} \]
Proof Trees

• PROLOG attempts to build a proof tree starting from the goal
  • The clauses are the inference rules
  • The atoms are the axioms

• PROLOG execution is proof search
  • Try all possible proofs until success or exhaustion
Incomplete Proof Search

• PROLOG semantics implies breadth-first search of the tree
  • Finds a proof if one exists

• Breadth-first is very slow

• Implementations use depth-first
  • May lead to non-termination
  • Consider adding the rule hesnotokay :- shesokay
  • Now the goal shesokay loops, even though it remains provable
Example

\[
\begin{align*}
\vdash \ldots \\
\vdash \text{shesokay} \\
\vdash \text{hesnotokay} \\
\vdash \text{shesokay}
\end{align*}
\]

imokay :- youreokay, hesokay
youreokay :- theyreokay
hesokay.
theyreokay.
hesnotokay :- shesokay
hesnotokay :- imnotokay
shesokay :- hesnotokay
shesokay :- theyreokay
Substitutions

• In general, execution must also compute a substitution for the variables of a goal

• *Revised rule*: To satisfy a goal $g$, find the first untried rule $G ::= H_1,...,H_n$ such that $s_1 = \text{unify}(g,G)$
  
  - $\text{unify}$ computes a substitution $s_1$ such that $s_1(g) = s_1(G)$
  - Add $s_1(H_1)$ as a subgoal.
  - If $s_1(H_1)$ succeeds, it returns a substitution $s_2$
  - Add $s_2(s_1(H_2))$ as a subgoal, repeat.
  - If all subgoals succeed, result is the substitution $s_n \circ ... \circ s_2 \circ s_1$
Backtracking Revisited

• A new form of backtracking arises with substitutions

• Consider a rule $G : - H_1, H_2, \ldots, H_n$
  • If $s_1(H_2)$ fails, maybe $H_1$ could succeed with a different substitution $s_1'$
    • Maybe $H_1$ could be proven using a different rule with a different substitution $s_1'$
  • We must try all possible ways to prove $H_1$ using different rules to try to prove $H_2$
  • In general, backtracking must be done within a single right-hand side to ensure all possible ways of satisfying subgoals are tried
Example, Part 1

Goal: \( \text{rev}(\text{cons}(1,\text{cons}(2,\text{nil})), A) \)

Rule: \( \text{rev}(\text{cons}(X,Y),Z) : - \text{rev}(Y,W), \text{addright}(W,X,Z) \)

\[
\text{unify(} \text{rev}(\text{cons}(1,\text{cons}(2,\text{nil})),A),\text{rev}(\text{cons}(X,Y),Z)\text{)} = \{X=1, Y=\text{cons}(2,\text{nil}), A=Z\}
\]

Goal: \( \text{rev}(\text{cons}(2,\text{nil}), W) \)

Rule: \( \text{rev}(\text{cons}(X1,Y1), Z1) : - \text{rev}(Y1,W1), \text{addright}(W1,X1,Z1) \)

\[
\text{unify(} \text{rev}(\text{cons}(2,\text{nil}),W),\text{rev}(\text{cons}(X1,Y1),Z1)\text{)} = \{X1=2, Y1=\text{nil}, Z1 = W\}
\]
Example, Part 2

Goal: \texttt{rev(nil,W1)}

Rule: \texttt{rev(nil,nil)}.

\[
\text{unify}(\texttt{rev(nil,W1)}, \texttt{rev(nil,nil)}) = \{ \text{W1= nil} \}
\]

Goal: \texttt{addright(nil,2,W)}

Rule: \texttt{addright(nil,X2,[X2]).}

\[
\text{unify}(\texttt{addright(nil,2,W)}, \texttt{addright(nil,X2,[X2])}) = \{ \text{X2=2, W=[2]} \}
\]
Example, Part 3

Goal: \( \text{addright}(\text{cons}(2,\text{nil}),1,A) \)

Rule: \( \text{addright}(\text{cons}(A3,B3),X3,\text{cons}(A3,Z3)) :\) \( \text{:- addright}(B3,X3,Z3) \)

unify: ... \( \{ A3=2, B3=\text{nil}, X3=1, A=\text{cons}(2,Z3) \} \)

Goal: \( \text{addright}(\text{nil},1,Z3) \)

Rule: \( \text{addright}(\text{nil},X4,[X4]). \)

Unify: ... \( \{ X4=1, Z3=[1] \} \)

The answer is \( A \) in the final substitution: \( A = [2,1] \)
The Occurs Check

• PROLOG deviates from the semantics in ways besides using depth-first search

• The semantics only allows finite terms in substitutions.
  • Requires an occur check on $a = T$ to ensure $a$ does not occur in $T$
  • The occurs check is expensive and claimed to be rarely needed
  • Most implementations omit the occurs check
Cut

• Backtracking can be expensive, so PROLOG includes a feature ! (pronounced “cut”) to control it

• Consider A :- B, C, !, D
  • PROLOG will not backtrack past a !
  • If D fails, the implementation will not attempt to resatisfy B and C
  • The entire rhs fails immediately

• Controlling backtracking is critical to writing respectably efficient PROLOG programs.
Discussion

• The building blocks of PROLOG implementations are:
  • matching to select clauses that could satisfy a goal
  • unification
  • backtracking

• Implementations are sensitive to the order of rules and the order of subgoals on rule right-hand sides

• Cut provides even more control
Opinions

• Logic programming is interesting.

• At best:
  • very declarative
  • very easy to write certain programs (e.g., search)

• At worst:
  • ideas of “algorithm” and “complexity” are obscured
    • really just one algorithm, exponential proof search
  • performance relies on tricky rule/goal orderings:
    • not very scalable
    • obscure
More Opinions

• Logic programming languages are usually untyped or only weakly typed

• Difficult to design reasonably strong type systems
Logic Programming Today

• Popularity in the ‘80’s to bust in the ‘90’s
  • General purpose logic programming is out of fashion

• But special-purpose logic programming is commercially important
  • Domain-specific logic languages for scheduling
    • airline crews, trucking, manufacturing, chip design
    • Use search techniques and constraint languages to solve NP-hard problems
  • Databases
  • Programming languages
    • Type inference!