CS 476 – Programming Language Design

William Mansky
Logic Programming

• Declarative programming: say what you want, not how to do it
• A logic program consists of a series of logical assertions, and a query:

\[
\text{man(socrates).}
\]
\[
\text{mortal(X) :- man(X).}
\]
\[
?\text{- mortal(socrates).}
\]
\[
\text{true.}
\]
Logic Programming

• Declarative programming: say what you want, not how to do it
• A logic program consists of a series of logical assertions, and a query:

\[
\text{man(socrates).}
\]
\[
\text{mortal(X) :- man(X).}
\]
\[
?- \text{mortal(X).}
\]
\[
X = \text{socrates.}
\]
Logic Programming

age(person1, 21).
age(person2, 23).
age(person3, 25).
age(person4, 27).

\[
\begin{align*}
\text{older}(X, Y) & : \text{age}(X, X_{\text{age}}), \text{age}(Y, Y_{\text{age}}), X_{\text{age}} > Y_{\text{age}} \\
\text{older}(X, Y) & \neg\text{age}(X, \text{person1}), \text{older}(Y, X).
\end{align*}
\]

? - older (X, person1), older(Y, X).

X = person2, Y = person3; X = person2, Y = person4;
X = person3, Y = person4.
Logic Programming: Syntax

\[ T ::= \text{true} \mid \text{<ident>} \mid \text{<#>} \mid \text{<Ident>} \mid \text{<ident>}(T, \ldots, T) \]

\[ R ::= T :: T, \ldots, T. \]

\[ Q ::= ?- T, \ldots, T. \]

\[ P ::= R \ldots R Q \]

Syntactic sugar: \( t. \Rightarrow t :: \text{true} \).
Logic Programming: Execution

• Maintain a list of goals that still need to be proved
• Pick a goal to prove next
• Find a rule whose conclusion matches the goal, and apply it:
  — Instantiate it to match the goal, by unification
  — Replace the goal with the instantiated premises of the rule
• If no rules apply, backtrack to the last decision point and make a different choice
• If all goals are solved, output the solution
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...

Goals: older(X, person1), older(Y, X)

older(X, Y) :- age(X, Xage), age(Y, Yage), Xage > Yage.
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...
Goals: older(X, person1), older(Y, X)

older(X’, Y’) :- age(X’, Xage), age(Y’, Yage), Xage > Yage.
unify(older(X, person1), older(X’, Y’)) =
   {X’ → X, Y’ → person1}
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...

Goals: age(X, Xage), age(person1, Yage), Xage > Yage, older(Y, X)

older(X’, Y’) :- age(X’, Xage), age(Y’, Yage), Xage > Yage.

unify(older(X, person1), older(X’, Y’)) =

{X’ ↦ X, Y’ ↦ person1}
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...

Goals: age(X, Xage), age(person1, Yage), Xage > Yage, older(Y, X)

age(person1, 21).

unify(age(X, Xage), age(person1, 21)) =
  {X ↦ person1, Xage ↦ 21}
Logic Programming: Execution

Rules: \text{age(person1, 21), ..., older(X, Y) :- ...}

Goals: \text{age(person1, Yage), 21 > Yage, older(Y, person1)}

\text{age(person1, 21).}

\text{unify(age(X, Xage), age(person1, 21)) =}
\quad \{X \mapsto \text{person1, } Xage \mapsto 21\}
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...
Goals: 21 > 21, older(Y, person1)

Unproveable!
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...  
Goals: age(X, Xage), age(person1, Yage), Xage > Yage, older(Y, X)

age(person1, 21).
unify(age(X, Xage), age(person1, 21)) =  
\{X \mapsto \text{person1}, Xage \mapsto 21\}
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...
Goals: age(X, Xage), age(person1, Yage), Xage > Yage, older(Y, X)

age(person2, 23).
unify(age(X, Xage), age(person2, 23)) =
   {X ↦ person2, Xage ↦ 23}
Logic Programming: Execution

Rules: age(person1, 21), ..., older(X, Y) :- ...

Goals:

\{X \mapsto \text{person2}, Y \mapsto \text{person3}\}
Logic Programming: Execution

• Maintain a list of goals that still need to be proved
• Pick a goal to prove next
• Find a rule whose conclusion matches the goal, and apply it:
  — Instantiate it to match the goal, by unification
  — Replace the goal with the instantiated premises of the rule
• If no rules apply, backtrack to the last decision point and make a different choice
• If all goals are solved, output the solution
Logic Programming: Semantics

• A configuration is a tuple \((g, R, \sigma, k)\) where:
  — \(g\) is the list of goals
  — \(R\) is the set of rules left to consider at this step
  — \(\sigma\) is the solution (substitution) computed so far
  — \(k\) is the stack for backtracking

• The small-step relation is:
  \[ R_0 \vdash (g, R, \sigma, k) \rightarrow (g', R', \sigma', k') \]
  since we need to keep track of the full rule list as well
Logic Programming: Semantics

\[
\begin{align*}
\text{if } r \in R \text{ then } \text{make_fresh}(r) &= t : -t_1, \ldots, t_n \quad \text{unify}(g, t) = \sigma_1 \\
R_0 \vdash (g :: gs, R, \sigma, k) &\rightarrow \\
([\sigma_1]([t_1; \ldots; t_n] @ gs), R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k) \\
\end{align*}
\]

\[
\begin{align*}
\text{if } r \in R \text{ then } \text{make_fresh}(r) &= t : -t_1, \ldots, t_n \quad \text{unify}(g, t) = \text{fail} \\
R_0 \vdash (g :: gs, R, \sigma, k) &\rightarrow (g :: gs, R - \{r\}, \sigma, k) \\
\end{align*}
\]
Logic Programming: Semantics

\[ R_0 \vdash (\varnothing, R, \sigma, k) \rightarrow \sigma \]

\[ R_0 \vdash (g :: gs, \{\}, \sigma, (gs', R', \sigma') :: k) \rightarrow (gs', R', \sigma', k) \]

\[ R_0 \vdash (g :: gs, \{\}, \sigma, []) \rightarrow \text{false} \]

• Values: substitutions \( \sigma \), and \( \text{false} \)
Logic Programming: Execution

• Note: this language is Turing-complete!
• So there are non-terminating logic programs

\[
\text{prop}(X) \\
\text{prop}(X)
\]

• Syntax-directed rule systems avoid backtracking, but might still fail to terminate
Logic Programming: Negation

- We can define other connectives in Prolog:

\[
\text{and}(P, Q) :\neg P, Q.
\]

\[
\text{or}(P, Q) :\neg P.
\]

\[
\text{or}(P, Q) :\neg Q.
\]

What about “not”? 

\[
\frac{P}{P \land Q}
\]

\[
\frac{Q}{P \lor Q}
\]

\[
\frac{P}{P \lor Q}
\]
Logic Programming: Negation

• We can define other connectives in Prolog:

\[
\text{not}(P) :\ - P, \ \text{fail}.
\]

• Problem: \text{not}(P) can never be proved true!
Logic Programming: Negation

• We can define other connectives in Prolog:

```
not(P) :- P, fail.
not(P).
```

• Problem: not(P) can always be proved true!
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

\[
\text{not}(P) :- P, !, \text{fail}.
\]
\[
\text{not}(P).
\]

• No backtracking past ! ("cut")
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

not(P) :- P, !, fail.
not(P).

?- not(older(person2, person1)).
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

not(P) :- P, !, fail.
not(P).

older(person2, person1), !, fail.
Backtracking stack: not(older(person2, person1))
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

```prolog
not(P) :- P, !, fail.
not(P).

!, fail.
Backtracking stack: older(person2, person1), not(older(person2, person1))
```
Logic Programming: Negation by Cut

- We can define other connectives in Prolog:

\[
\text{not}(P) :- P, !, \text{fail.}
\]

\[
\text{not}(P).
\]

\[
\text{fail.}
\]

Backtracking stack:

Result: \textbf{false}
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

```
not(P) :- P, !, fail.
not(P).

?- not(older(person1, person2)).
```
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

\[
\text{not}(P) :- P, !, \text{fail.}
\]

\[
\text{not}(P).
\]

\[
\text{older(person1, person2), !, fail.}
\]

Backtracking stack: \text{not(older(person2, person1))}
Logic Programming: Negation by Cut

• We can define other connectives in Prolog:

\[
\text{not(P) :- P, !, fail.}
\]
\[
\text{not(P).}
\]

\[
\text{not(older(person2, person1))}
\]

Backtracking stack:
Result: \text{true}
Logic Programming: Syntax

\[ T ::= \ldots \mid \text{fail} \mid ! \]
\[ R ::= T :- T, \ldots, T. \]
\[ Q ::= \text{?-} T, \ldots, T. \]
\[ P ::= R \ldots R \ Q \]
Logic Programming: Semantics

\[ R_0 \vdash (\text{fail} :: gs, R, \sigma, (gs', R', \sigma') :: k) \rightarrow (gs', R', \sigma', k) \]

\[ R_0 \vdash (! :: gs, R, \sigma, k) \rightarrow (gs, R, \sigma, []) \]
Logic Programming

• Give a set of rules, ask questions about what can be proved
• Searches for a proof tree for the query, filling in variables as it goes, and backtracking when it hits a dead end
• Uses unification to figure out how to apply a rule to a goal
• Useful for databases and knowledge retrieval systems
• Can be used for PL too, but not as efficient as syntax-directed algorithms
• See also λProlog: Prolog + lambda calculus!