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:

man(socrates).
mortal(X) :- man(X).
?- mortal(socrates).
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:

\[
\text{man(}\text{socrates}\text{).}
\]
\[
\text{mortal}(X) :\neg \text{man}(X).
\]
\[
?\text{- mortal}(X).
\]
\[
X = \text{socrates}.
\]
age(person1, 21).
age(person2, 23).
age(person3, 25).
age(person4, 27).

older(X, Y) :- age(X, X_{age}), age(Y, Y_{age}), X_{age} > Y_{age}.

?- older(X, person1), older(Y, X).
X = person2, Y = person3; X = person2, Y = person4; X = person3, Y = person4.
Logic Programming: Syntax

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

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

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

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

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

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

\text{older(X', Y') :- age(X', Xage), age(Y', Yage), Xage > Yage.}

\text{unify(older(X, person1), older(X', Y')) =}

\{X' \mapsto X, Y' \mapsto \text{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 \mapsto \text{person1}, \ Xage \mapsto 21\}
Logic Programming: Execution

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

Goals: age(person1, Yage), 21 > Yage, older(Y, person1)

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: 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
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

\[ r \in R \quad \text{make\_fresh}(r) = t : -t_1, \ldots, t_n \quad \text{unify}(g, t) = \sigma_1 \]

\[ 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) \]

\[ r \in R \quad \text{make\_fresh}(r) = t : -t_1, \ldots, t_n \quad \text{unify}(g, t) = \text{fail} \]

\[ R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow (g :: gs, R - \{r\}, \sigma, k) \]
Logic Programming: Semantics

\[
R_0 \vdash ([], R, \sigma, k) \to \sigma
\]

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

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

• Values: substitutions $\sigma$, and false
• 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: Cut

• Consider:

max(X, Y, X) :- X >= Y. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: max(78, 51, Z), gs
Stack: [(gs1, ...); ...]
Logic Programming: Cut

• Consider:

max(X, Y, X) :- X > Y. (* max rule 1 *)
max(X, Y, Y) :- X < Y. (* max rule 2 *)

Goals: 78 >= 51, [Z ↦ 78](gs)
Stack: [(max(78, 51, Z), ...); (gs1, ...); ...]
Logic Programming: Cut

• Consider:

max(X, Y, X) :- X >= Y. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: [Z ↦ 78](gs)
Stack: [(max(78, 51, Z), ...); (gs1, ...); ...]
Logic Programming: Cut

• Consider:

max(X, Y, X) :- X >= Y. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: max(78, 51, Z), gs
Stack: [(gs1, ...); ...]
Logic Programming: Cut

• We can fix this with a cut:

max(X, Y, X) :- X >= Y, !. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: max(78, 51, Z), gs
Stack: [(gs1, ...); ...]
Logic Programming: Cut

• We can fix this with a cut:

\[
\text{max}(X, Y, X) :- X \geq Y, \text{!} \text{.} \quad (* \text{max rule 1 } *) \\
\text{max}(X, Y, Y) :- X < Y. \quad (* \text{max rule 2 } *)
\]

Goals: !, [Z ↦ 78](gs)
Stack: [(max(78, 51, Z), ...); (gs1, ...); ...]
Logic Programming: Cut

• We can fix this with a cut:

max(X, Y, X) :- X >= Y, !. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: [Z ↦ 78](gs)
Stack: [(gs1, ...); ...]
Logic Programming: Cut

• We can fix this with a cut:

max(X, Y, X) :- X >= Y, !. (* max rule 1 *)
max(X, Y, Y) :- X < Y.   (* max rule 2 *)

Goals: gs1
Stack: [...]


Logic Programming: Cut

• We can cut after a complicated computation:

\[ \text{prop}(X, Y) :- \ p1(X), \ p2(X), \ p3(Y), \ !. \]
Logic Programming: Syntax

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

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

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

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

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

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

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

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

\[
\begin{align*}
& r \in R \quad \text{make\_fresh}(r) = t : - t_1, \ldots, t_n \\
& \text{unify}(g, t) = \sigma_1 
\end{align*}
\]

\[
R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow \\
([\sigma_1]( [t_1; \ldots; t_n] \@ gs ) , R_0, \sigma_1 \circ \sigma , (g :: gs, R - \{r\}, \sigma) :: k )
\]
Logic Programming: Semantics

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

\[ r \in R \quad \text{make\_fresh}(r) = t : - t_1, ..., t_n \quad \text{unify}(g, t) = \sigma_1 \]

\[ \text{mark\_cuts}(k, t_1, ..., t_n) = u_1, ..., u_n \]

\[ R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow ([\sigma_1](u_1; ...; u_n) \ @ gs), R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k) \]
Logic Programming

• A form of declarative programming where we give a set of rules and then 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