CS 476 – Programming Language Design

William Mansky
Questions?
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:

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

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

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.
```
Questions?

Top
Logic Programming: Syntax

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

“atom”

• Examples: socrates, person1, pizza, ...
Logic Programming: Syntax

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

- Examples: X, Y, Z, ...

variable
Logic Programming: Syntax

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

• Examples: mortal, age, has_value, ...
• Can take any number of arguments
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: \( t. \Rightarrow t :- \text{true}. \)
Questions?
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: 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)

Unprovable!
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: \texttt{age(person1, 21), ..., older(X, Y) :- ...}

Goals:

\{X \mapsto \texttt{person2, Y} \mapsto \texttt{person3}\}
Questions?

Top
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

\[ r \in R \]

\[ R_0 \vdash (g :: gs, R, \sigma, k) \]

• 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
Logic Programming: Semantics

\[ r \in R \]
\[ R_0 \vdash (g :: gs, R, \sigma, k) \]

• Pick a goal to prove next
• Find a rule whose conclusion matches the goal
  — Choose a rule
Logic Programming: Semantics

\[ r \in R \quad \text{make\_fresh}(r) = t : - t_1, \ldots, t_n \]
\[ R_0 \vdash (g :: gs, R, \sigma, k) \]

• Pick a goal to prove next
• Find a rule whose conclusion matches the goal
  — Choose a rule
  — Make a fresh copy of the rule, so variables don’t overlap
Logic Programming: Semantics

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

\[ R_0 \vdash (g :: gs, R, \sigma, k) \]

• Pick a goal to prove next
• Find a rule whose conclusion matches the goal
  — Choose a rule
  — Make a fresh copy of the rule, so variables don’t overlap
  — Check whether the rule’s conclusion matches the goal
Logic Programming: Semantics

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

• Find a rule whose conclusion matches the goal
  — Choose a rule
  — Make a fresh copy of the rule, so variables don’t overlap
  — Check whether the rule’s conclusion matches the goal

• Replace the goal with instantiated premises of the rule
Logic Programming: Semantics

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

\[
\begin{align*}
  r \in R \quad & \text{make\_fresh}(r) = t :: t_1, \ldots, t_n \quad \text{unify}(g, t) = \text{fail} \\
  R_0 \vdash (g :: gs, R, \sigma, k) \to (g :: gs, R - \{r\}, \sigma, k)
\end{align*}
\]

• If the rule doesn’t match, try another rule
Logic Programming: Semantics

\[ R_0 \leftarrow ([] , R, \sigma , k) \rightarrow \sigma \]

- If we solve all the goals, return the current substitution \( \sigma \)
Logic Programming: Semantics

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

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

• If no rules apply (i.e., we run out of rules to try), backtrack to the last decision point in the stack and make a different choice
Logic Programming: Semantics

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

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

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

• If there’s nowhere to backtrack to, the goal is unprovable
Logic Programming: Execution

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

\[
\text{circular}(X) \\
\text{circular}(X)
\]
Questions?
Logic Programming: Negation

• We can define other connectives in Prolog:

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

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

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

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

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

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

What about “not”?
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:

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

• No backtracking past ! ("cut")
Logic Programming: Syntax

\[ T ::= \ldots \mid \text{fail} \mid ! \]

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

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

\[ P ::= R \ldots R \, Q \]
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!
End-of-Semester Feedback

• Exercise: Fill out the teaching evaluation, and submit exercise 11/22 with a message saying “finished”

• As before, please don’t submit your actual feedback on Gradescope!

• Thank you for your feedback!