CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

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:

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

Logic Programming

age(person1, 21).

age(person2, 23).

age(person3, 25).

age(person4, 27).

$$\frac{\mathrm{age}(X, X_{\mathrm{age}}) \quad \mathrm{age}(Y, Y_{\mathrm{age}}) \quad X_{\mathrm{age}} > Y_{\mathrm{age}}}{\mathrm{older}(X, Y)}$$

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

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

Exercise: What values of X and Y make this query true?

Logic Programming

age(person1, 21).

age(person2, 23).

age(person3, 25).

age(person4, 27).

$$\frac{\mathrm{age}(X, X_{\mathrm{age}}) \quad \mathrm{age}(Y, Y_{\mathrm{age}}) \quad X_{\mathrm{age}} > Y_{\mathrm{age}}}{\mathrm{older}(X, Y)}$$

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

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

X = person2, Y = person3; X = person2, Y = person4;
X = person3, Y=person4.

Logic Programming

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad (v = v_1 + v_2)}{e_1 + e_2 \Downarrow v}$$

```
eval(add(E1, E2), V) :- eval(E1, V1), eval(E2, V2), V = V1 + V2.
```

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Logic Programming: Syntax

T ::= true | <ident> | <#> | <Ident> | <ident>(*T*, ..., *T*)

"atom"

- Examples: socrates, person1, pizza, ...

Logic Programming: Syntax

T ::= true | <ident> | <#> | <Ident> | <ident>(*T*, ..., *T*)

variable

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

Logic Programming: Syntax

T ::= true | <ident> | <#> | <Ident> | <ident>(*T*, ..., *T*)

predicate

- Examples: mortal, age, has_value, ...
- Can take any number of arguments

Logic Programming: Syntax

T ::= true | <ident> | <#> | <Ident> | <ident>(*T*, …, *T*)

R ::= *T* :- *T*, …, *T*.

Q ***::=*** ?- *T*, …, *T*.

P ::= *R* … *R* *Q*

Syntactic sugar: t. => t :- true.

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

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’)) =

$\{X' \mapsto X, Y' \mapsto person1\}$

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')) =

$\{X' \mapsto X, Y' \mapsto person1\}$

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 ↦ person1, Xage ↦ 21}

Logic Programming: Execution

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

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

Unprovable!

Logic Programming: Execution

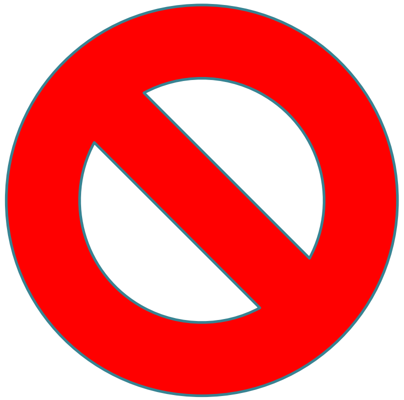
Rules: age(person1, 21), ..., older(X, Y) :- ...

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

age(person1, 21).

unify(age(X, Xage), age(person1, 21)) =

{X $\mapsto$ 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)) =

$\{X \mapsto \text{person2}, \text{Xage} \mapsto 23\}$

Logic Programming: Execution

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

Goals:

{X ↦ person2, Y ↦ person3}

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

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, σ, k) where:
  - — g is the list of goals
  - — R is the set of rules left to consider at this step
  - — σ is the solution (substitution) computed so far
  - — k is the stack for backtracking
- The small-step relation is

$$R_0 \vdash (g, R, \sigma, k) \to (g', R', \sigma', k')$$

since we need to keep track of the full rule list as well

Logic Programming: Semantics

$$\frac{r \in R}{R_0 \vdash (g :: gs, R, \sigma, k)}$$

- 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

$$\frac{r \in R}{R_0 \vdash (g :: gs, R, \sigma, k)}$$

- Pick a goal to prove next
- Find a rule whose conclusion matches the goal
  - Choose a rule

Logic Programming: Semantics

$$\frac{r \in R \qquad \text{make\_fresh}(r) = t :- t_1, \dots, t_n}{R_0 \vdash (g :: gs, R, \sigma, k)}$$

- 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

$$\frac{r \in R \quad \text{make\_fresh}(r) = t :- t_1, \ldots, t_n \quad \text{unify}(g, t) = \sigma_1}{R_0 \vdash (g :: gs, R, \sigma, k)}$$

- 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

$$\frac{r \in R \quad \text{make\_fresh}(r) = t :- t_1, \dots, t_n \quad \text{unify}(g, t) = \sigma_1}{R_0 \vdash (g :: gs, R, \sigma, k) \to ([\sigma_1]([t_1; \dots; t_n] \mathbin{@} gs), R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k)}$$

- 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

$$\frac{r \in R \quad \text{make\_fresh}(r) = t :- t_1, \ldots, t_n \quad \text{unify}(g, t) = \sigma_1}{\begin{array}{c} R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow \\ ([\sigma_1]([t_1; \ldots; t_n] \mathbin{@} gs), R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k) \end{array}}$$

$$\frac{r \in R \quad \text{make\_fresh}(r) = t :- t_1, \ldots, t_n \quad \text{unify}(g, t) = \text{fail}}{R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow (g :: gs, R - \{r\}, \sigma, k)}$$

- If the rule doesn't match, try another rule

Logic Programming: Semantics

$$\frac{}{R_0 \vdash ([], R, \sigma, k) \rightarrow \sigma}$$

- If we solve all the goals, return the current substitution σ

Logic Programming: Semantics

$$\overline{R_0 \vdash ([], R, \sigma, k) \rightarrow \sigma}$$

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

- 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

$$\overline{R_0 \vdash ([], R, \sigma, k) \rightarrow \sigma}$$

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

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

- 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

$$\frac{\text{circular}(X)}{\text{circular}(X)}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Logic Programming: Negation

- We can define other connectives in Prolog:

```
and(P, Q) :- P, Q.
```

$$\frac{P \quad Q}{P \land Q}$$

```
or(P, Q) :- P.
or(P, Q) :- Q.
```

$$\frac{P}{P \lor Q} \qquad \frac{Q}{P \lor Q}$$

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 ::= … | fail | !

R ::= *T* :- *T*, …, *T*.

Q ::= ?- *T*, …, *T*.

P ::= *R* … *R* *Q*

Logic Programming: Semantics

$$\frac{}{R_0 \vdash (\text{fail} :: gs, R, \sigma, (gs', R', \sigma') :: k) \to (gs', R', \sigma', k)}$$

$$\frac{}{R_0 \vdash (! :: gs, R, \sigma, k) \to (gs, R, \sigma, [])}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

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!