Concepts of Programming Languages

Logic Programming and Prolog (I)

Prof. Dr. Ulrike Pado

A Proof Engine

- Prolog is an engine for logically proving your queries given a set of facts and rules previously provided

- You will receive “yes” as an answer to your query if a proof has been found (“no” otherwise)
- Alternatively, you can ask “For which elements does the query hold?” and you will receive all these elements back

Applications: Computational Linguistics

- Analysing sentence structure
  - “What structure can be assigned to the sentence *A man sees a clown with a telescope*?”
- Representing human knowledge
  - “What is the relationship between a railway station and an airport?”

Applications: Artificial Intelligence

- Expert systems:
  - “Which therapies are recommended for the following symptoms?”
  - “Which issues could have caused the following system state?”
  - Scheduling problems (HFT time table, parcel delivery, …)
  - IBM’s Watson!

LISP and Prolog

- Languages for reasoning about abstract content (as opposed to number crunching)
  - LISP: Mathematical functions
  - Prolog: First-order logic
- Programs as data
  - LISP: modify and evaluate arguments
  - Prolog: Program code is statement of knowledge

Today's Goals

- More about proofs
- First-order logic (FOL) and Horn clauses
- Creating proofs in FOL: Horn clause inference and unification
- Prolog syntax

- Next time: Proof search (finding all possible proofs)

Logic: Proofs

A proof of a hypothesis is constructed from

- a set of axioms (possibly empty)
- a set of inference mechanisms

We will call a set of axioms a knowledge base (Prolog terminology)

Axioms (Facts and Rules)

- Examples:
  - All right angles are equal. (Fact)
  - Socrates is human. (Fact)
  - Whoever is human, is mortal. (Rule)
- Axioms in the knowledge base count as proven
- In logic, it is desirable to have a minimal set of axioms (assumptions)
- In logic programming, all relevant knowledge is encoded as axioms

Inference Mechanism: Modus Ponens

- When the “if”-part of a conditional rule can be proven, we deduce the “then”-part is true also
- Example:
  - Socrates is human.
  - If Socrates is human, then Socrates is mortal.
  - Therefore, Socrates is mortal.
- In logic, there are a number of valid inference mechanisms
- Logic programming uses only modus ponens

Successful Proofs

A hypothesis is proven if

- it can be derived from the axioms given the inference mechanism (positive proof)
- proving its negation leads to a contradiction such as a & ~a (refutation, proof by contradiction)

Today’s Goals

- More about proofs
- First-order logic (FOL) and Horn clauses
- Creating proofs in FOL: Horn clause inference and unification
- Prolog syntax

First-order logic (FOL)

- Terms (taken to describe objects): `socrates`
- Predicates (taken to describe properties): `human(socrates)`
  - First-order: Predicates cannot be variables
- Connectors (to express rules):
  $\rightarrow$ (if – then), ~ (not), & (and), v (or)
- Quantifiers: $\forall$x (for all x), $\exists$x (there is an x)

FOL Example Formulas and Rules

- human(socrates) & mortal(socrates)
- teaches(socrates,plato)
- $\exists$x professor(x) & busy(x)
  "There is (at least) one busy professor"
- $\forall$x professor(x) → busy(x)
  "Anybody who is a professor is busy"

Horn Clauses

(named for logician Alfred Horn)

- Special formulation of conditional rules that relies only on disjunction and negation
- Allows modus ponens-type inference to prove a FOL hypothesis given the knowledge base
- Horn clause inference is Turing complete
  - Forms the basis of Prolog

Horn Clauses: Example

- Formulate rule as disjunction of terms
  - $\forall x$ professor(x) → busy(x) becomes
    $\forall x$ ~professor(x) v busy(x)
- Assume universal quantification of variables
  - ~professor(x) v busy(x)
- Allow at most one non-negated term

Today's Goals

- More about proofs
- First-order logic (FOL) and Horn clauses
- Creating proofs in FOL: Horn clause inference and unification
- Prolog syntax

(Much Simplified) Inference

Assume ~professor(x) v busy(x)

- Non-negated term busy(x) is proven if the positive version of all negated terms can be proven
- busy(x) is sometimes called the **goal clause** in logic programming (though this has a different meaning for Horn clauses originally)
- The task of proving busy(x) becomes proving professor(x) and any other **fact clauses**
- Proving these fact clauses involves either lookup of facts in the knowledge base or modus ponens inference (if the fact is the goal clause of a rule)

Horn Clause Inference: Example

Hypothesis: busy(pado)

Assume KB: ~professor(x) v busy(x)

professor(pado)

- Find goal clause that corresponds to hypothesis, replacing variables if necessary
  - Replace x by pado
- Prove negation of fact clauses
  - professor(pado) is in KB

Replacing Variables

- Find goal clause that corresponds to hypothesis, replacing variables if necessary
- Unification: Substitute a term (or a predicate) for a variable – if this can be done consistently!

Unification Rules

- Two terms unify if they are the same string or number
- A variable unifies with a term, a predicate or another variable, taking the other term's value
- Two predicates unify if
  - they share a “name”
  - they have the same number of arguments
  - their arguments unify

Unification in General

- Unification is a general mechanism
- Used, e.g., in type inference
- Powerful and efficient way to share information
- After two terms are unified, they are *the same term*

Danger! Infinite Loops!

- When unifying a predicate with a variable, beware of loops:
  - X = f(X)
  - f(X) = f(f(X))
  - and so on, into infinity

What to do with X=f(X)?

a) The terms unify, because ultimately, there will be infinitely many f(f(f(… on both sides

b) The terms don't unify, because we cannot compute their unification with finite equipment

- Standard unification algorithms take the second stance and perform an **occurs check** for the variable in the predicate
- Prolog takes the first!

X=f(X)? Yes

- Prolog does not run occurs checks before every unification operation for efficiency reasons
- Old Prolog implementations performed unification until they ran out of memory…
- Modern implementations deal gracefully, but still accept recursive unification

Today's Goals

- More about proofs
- First-order logic (FOL) and Horn clauses
- Creating proofs in FOL: Horn clause inference and unification
- Prolog syntax

Logic and Programming

We want to know (given a knowledge base)

- whether a query is true given what we know
  - Does the patient have measles?
- which variable bindings make the query true
  - Which illnesses could cause a red rash and a fever?

Stating Knowledge

- Facts
  - Pizza is tasty: `tasty(pizza).`
  - If a fact is in the knowledge base, it counts as logically proven
- Rules
  - If pizza is tasty, then students will eat it:
    `eat(student,pizza) :- tasty(pizza).`
  - We know the “if”-part is true, therefore we can deduce the “then”-part is true also

More Complex Rules

- AND: comma
  - `eats(student,pizza) :- tasty(pizza), cheap(pizza).`
- OR: semicolon
  - `eats(student,pizza) :- tasty(pizza); cheap(pizza).`
- NOT: not
  - `eats(student,pizza) :- tasty(pizza), not(expensive(pizza)).`

Success and Failure

- Prolog answers queries according to its success in proving them
  - `tasty(pizza).` yes
  - `eats(student,pizza).` yes
  - `gone(pizza).` yes
  - `eats(student,pasta).` no

```
tasty(pizza).
eats(student,pizza) :- tasty(pizza).
gone(pizza) :- eats(student,pizza).
```

Queries with Variables: Unification

- Substitute a variable for a term to find all terms that satisfy the query
  - `tasty(X)` X=pizza
  - `eats(X,Y)` X=student
    Y=pizza
- In a query with variables, hit ";" after receiving the first variable binding to see the next option

```
tasty(pizza).
eats(student,pizza) :- tasty(pizza).
gone(pizza) :- eats(student,pizza).
```

Recursion and Cases

- Recursion instead of loops
- Different rules for different cases instead of if/else

if the list is empty

recursive base case

else

```
printlist([]) :- write(' Done!').

printlist([H|T]) :- write(H),
                    printlist(T).
```

recursive call

Prolog Literature

- Blackburn, Bos & Striegnitz, “Learn Prolog Now!” (e-Book)
- Clocksin & Mellish, “Programming in Prolog”