3.1. Types of Logic

Types of Inference

- There are several types of (formal) logic inferences:

  1. **Deduction** (from general to particular):
     - e.g., from $\forall x \ p(x) \rightarrow q(x)$ and $p(A)$ we can infer $q(A)$
     - All dogs have tails, Wags is a dog, thus Wags has a tail

  2. **Abduction** (from effects/observations to causes/hypotheses)
     - e.g., from $\forall x \ p(x) \rightarrow q(x)$ and $q(A)$ we could infer $p(A)$
     - All dogs have tails, Wags has a tail, thus Wags is a dog

  3. **Induction** (from particular to general)
     - e.g., from $p(A), p(B), p(C)$, we could infer $\forall x \ p(x)$
     - Wags is a dog and has a tail, Spotty is a dog and has a tail, Rover is a dog and has a tail.
     - Thus, All dogs have a tail.
### 3.1. Types of Logic

**Limitations of 1st Order Predicate Logic**

1st order predicate calculus is:

- **Binary**: only 2 level of truth: true or false (no degrees of uncertainty).
  - Other logic systems allow uncertainty, e.g., If body temperature suddenly raises to 39 celsius degrees, there is 0.8 evidence of a viral disease.
- **Monotonic**: what is known cannot be retracted or changed.
  - A nonmonotonic system allows truth values to change.
  - E.g., ‘default’ logic: if we know that Cleo is a bird, and thus conclude it flies. Later, we learn Cleo is an ostrich, so we must retract that conclusion.
  - E.g., ‘temporal’ logic: input to medical expert system includes patient’s temperature. Later, temperature rises, and we thus need to change the fact.

### 3.2 Reasoning with Rules

**Kinds of rules**

- Rules express various different kinds of relations:
  - **situation** → **action**
    e.g.: high temperature → give aspirin
  - **cause** → **effect**
    e.g.: flu → fever (symptom)
  - **conditions** → **conclusions**
    e.g.: fever (symptom) → flu
  - **subclass** → **class**
    e.g.: ostrich → bird
3.2 Reasoning with Rules

Requirements of a Rule-based system

- **Unification**: We need to match (unify) terms against the components of rules (premises, conclusions)

- **Inference**: a mechanism to apply rules to the facts we have to reach new conclusions.

Types of reasoning with rules:

- **Forward chaining**: match the premises of each rule against our known facts to derive new facts
  - Also known as “forward reasoning” or “data driven” reasoning

- **Backward chaining**: test whether a given hypothesis is true by:
  - find a rule which has the hypothesis as the conclusion, and testing its premises.
  - Also known as “backward reasoning”, or “goal-driven” reasoning
3.2 Reasoning with Rules

Choose Mechanism to Suit the Task

• Different tasks/problems typically treated by different reasoning methods:
  – Forward chaining:
    • Synthesis tasks (design, configuration, planning, ...)
      – In some cases, not very efficient as we might generate many thousands of conclusions before the one we want is generated
      – Useful where we want to know everything which can be derived from the known facts
      – Useful where there are many valid solutions, all potentially valid.
      – Can be used where all input data (facts) are assumed to be available from the beginning of each problem.

• Backward reasoning:
  • Analysis tasks (classification, diagnosis, ...)
    – Most useful where we have a known conclusion to check.
    – Also useful if there are a small number of possible conclusions to check (e.g., what virus does the patient have?)
    – This approach is also useful where fact base is not complete. If data is missing, the user is asked to provide missing details.
3.2 Reasoning with Rules

Example: rules for classifying fruit

R1: IF Shape = long and Color = green or yellow THEN Fruit = banana
R2: IF Shape = round or oblong and Diameter > 4 inches THEN FruitClass = vine
R3: IF Shape = round or oblong and Diameter < 4 inches THEN FruitClass = tree
R4: IF Seedcount = 1 THEN SeedClass = stonefruit
R5: IF Seedcount > 1 THEN SeedClass = multiple
R6: IF FruitClass = vine and Color = green THEN Fruit = watermelon
R7: IF FruitClass = vine and Surface = rough and Color = tan THEN Fruit = watermelon
R8: IF FruitClass = vine and Surface = smooth and Color = yellow THEN Fruit = cantaloupe
R9: IF FruitClass = tree and Color = orange and SeedClass = stonefruit THEN Fruit = apricot
R10: IF FruitClass = tree and Color = orange and SeedClass = multiple THEN Fruit = orange
R11: IF FruitClass = tree and Color = red or yellow or green and SeedClass = multiple THEN Fruit = apple

3.2 Reasoning with Rules

Example: rules for classifying fruit (ii)

Given a fact list:
- Diameter = 3 inch,
- Shape = round,
- SeedCount > 1,
- Color = yellow,
- Surface = smooth

1) Is this an apple? (backward chaining)
   - Fruit = apple?
     R11: IF FruitClass = tree and Color = red or yellow or green and SeedClass = multiple THEN Fruit = apple
     • FruitClass = tree?
       • R2: IF Shape = round or oblong and Diameter < 4 inches THEN FruitClass = tree
         - Shape = round or oblong? YES (from facts)
         - Diameter > 4 inches? YES
         - YES
         • Color = red or yellow or green? YES
         • SeedClass = multiple YES
         • YES
3.2 Reasoning with Rules

Example: rules for classifying fruit (ii)

- Given a fact list:
  - Diameter = 5 inch,
  - Shape = round,
  - SeedCount > 1,
  - Color = yellow,
  - Surface = smooth

2) What is this fruit? (forward chaining)
- Apply rules one at a time, testing premises against fact list.
- If all premises match, add conclusion to fact list and start again.

R1: IF \( \text{Shape} = \text{long} \) and Color = green or yellow
    THEN Fruit = banana \text{ FAIL}

R2: IF \( \text{Shape} = \text{round} \) or oblong and Diameter > 4 inches
    THEN Fruitclass=tree \text{ SUCCEED}
        \( \rightarrow \) ADD Fruitclass=tree to factlist

- etc.

Figure 4.3 (Dankel y González). The Fruit Domain.
Graphical representation in terms of an and-or tree
3.2 Forward Chaining

- A simple algorithm for Forward Chaining

1. newFact = False
2. For rule in rule-list:
   - If all premises match fact-base:
     - For each fact in consequences:
       - if fact not in fact-base:
         - add fact to fact-base
         - newFact = True
3. If newFact: goto 1
3.2 Forward Chaining

A simple algorithm for Forward Chaining

1. newFacts = False
2. For rule in rule-list:
   - If all premises match fact-base:
     - For each fact in consequences:
       - if fact not in fact-base:
         - add fact to fact-base
         - newFacts = True
3. If newFacts: goto 1

Keep track whether a cycle has changed the fact-base
Cycle through each rule and apply those which match
When a cycle does not add any new facts, stop

3.2 Forward Chaining

Schema for Forward Chaining from Gonzalez and Dankel (modified)

Knowledge (Rules) -> Step 1: Match applicable rules -> Step 2: Conflict Resolution selected rule -> Step 3: Execution

Facts

facts -> Step 1: Match

new facts -> Step 3: Execution
3.2 Forward Chaining

Schema for Forward chaining from Gonzalez and Dankel (modified)

The KB is a set of rules

Knowledge (Rules) → applicable rules → selected rule → Execution

Step 1: Match

Step 2: Conflict Resolution

Step 3: Execution

Hold the known facts

Facts → new facts

First step: match premises of each rule against known facts to produce set of applicable rules
3.2 Forward Chaining

Schema for Forward chaining from Gonzalez and Dankel (modified)

Step 1: Match

Step 2: Choose one of applicable rules to apply (e.g., first rule which produces new facts)

Step 3: Apply the rule, adding new facts to the fact-base. Then repeat from (1)

Knowledge (Rules) → applicable rules → Steps 1-3 → Facts

New facts → Facts
3.2 Forward Chaining

The original diagram from Gonzalez and Dankel allowed the THEN of a rule to be a rule itself:

If Fruit=apple THEN if Color=yellow THEN Name=Golden

The execution of a rule could then add new rules rather than new facts.

3.2 Forward Chaining

The Forward Reasoning Algorithm:

1. **Match.** Compare the premises of each rule against the fact-base to discover the set of applicable rules.

2. **Conflict Resolution:** Where multiple rules are applicable, choose one rule to apply on some basis. E.g. Ignore rules where the conclusion is already known and choose first of the remaining rules.

3. **Execution** (apply the selected rule). Add the conclusion of the rule to the fact-base. In a more advanced system, derived facts could have a level of uncertainty associated, facts might be eliminated (negative certainty), we might add rules rather than facts, etc.

4. **End Condition.** Where no rule is applicable, or no rule meets the selection criteria.
3.2 Forward Chaining

**Example:** Rules for the classification of fruit:

- Look at the graph representing the rules
- It is very simple:
  - There are no variables (yet!)
  - There are no uncertainty factors (yet!)
  - No control rules (strategic rules, meta-rules)
3.2 Forward Chaining

3.2.1 implementation: representing facts

- All facts are unary (one argument)
- In Lisp, we will represent them in the form:
  `'\(\text{is} \ \text{<pred>} \ \text{<value>}\)`
  e.g.
  `'\(\text{is} \ \text{color} \ \text{red}\)`

- The code provides two functions to access fact parts:
  ```lisp
  (defun fact-prop (fact) (first fact))
  (defun fact-arg1 (fact) (second fact))
  (defun fact-arg2 (fact) (third fact))
  ```

3.2.1 implementation: representing rules

- Rules have the form:
  ```lisp
  (**id** IF **premiseList** THEN **conclusionList**)
  ```
- E.g.,
  ```lisp
  (R1a IF ((\text{is shape long})
           (\text{is color green}))
         THEN ((\text{is fruit banana}))
  ```

- The code provides two functions to access rule parts:
  ```lisp
  (defun rule-ifs (rule) (third rule))
  (defun rule-thens (rule) (fifth rule))
  ```
3.2 Forward Chaining

3.2.1 implementation: global variables

- The code depends on two global variables, representing the rule-base (knowledge base) and the fact-list (problem-base):

```
(defvar *rule-list*)
(defvar *fact-list*)
```

Defining the rule list:
```
(setq *rule-list*
  '( (R1a IF ((is shape long)
                (is color green))
      THEN ((is fruit banana))
    )
    (R1b IF ((is shape long)
             (is color yellow))
      THEN ((is fruit banana))
    )
    (R2a IF ((is shape round)
             (> diameter 10))
      THEN ((is fruitclass vine))
    )
    (R2b IF ((is shape oblong)
             (> diameter 10))
      THEN ((is fruitclass vine))
    )
  ))
```
3.2 Forward Chaining

3.2.1 implementation: global variables

- Defining the fact list:

```lisp
(setq *fact-list*
  '( (is diameter 5)
    (is shape round)
    (> seedcount 1)
    (is color yellow)
    (is surface smooth) )
```

3.2 Forward Chaining

3.2.1 implementation: **main function**

```lisp
(defun forward ()
  (if (execute-rule rule-list)
      (forward)))
```

Basic logic:

- While there is another rule to execute:
  - execute the rule

- Otherwise return nil.
3.2 Forward Chaining

3.2.1 implementation: **main function**

- A modification such that the topmost call to `forward` returns `true` if the fact-base was modified at all, `nil` otherwise:

```lisp
(defun forward (&optional (atLeastOne? nil))
  (if (execute-rule)
      (forward T)
      atLeastOne?))
```

3.2 Forward Chaining

3.2.1 implementation: **execute-rule**

- Searches through rule-list for a rule which is applicable, and produces new facts, and then executes it:
- Returns true if one applied, nil otherwise

```lisp
(defun execute-rule (rules)
  (cond ((null rules) nil)
        ((eval-rule-f (car rules)) t)
        (t (execute-rule (cdr rules)))))
```

OR more simply: **CHECK IF ANY WORKS HEE**

```lisp
(defun execute-rule ()
  (find-if #'eval-rule-f *rule-list*))
```
3.2 Forward Chaining

3.2.1 Implementation: **eval-rule-f**

- If the rule is applicable (all premises match) and one or more conclusions is new, then return true:

  ```lisp
  (defun eval-rule-f (rule)
    (if (satisfiable-ifs (rule-ifs rule))
      (derive-new-fact? (rule-thens rule)) )
  )
  
  (defun satisfiable-ifs (ifs)
    "Determine if all premises are satisfied"
    (every #'(lambda(x)
      (member x *fact-list* :test #'equal)) ifs))
  ```

3.2 Forward Chaining

3.2.1 Implementation: **derive-new-facts?**

- Returns true if any conclusion is new, else nil.

  ```lisp
  (defun derive-new-facts? (facts)
    (find T (mapcar #'(lambda(fact)
      (add-new-fact? fact))
      facts))
  )
  
  • **add-new-fact?** If conclusion is new, print it, add it to fact-list, and return t

    ```lisp
    (defun add-new-fact? (fact)
      (cond
        ((member fact *fact-list* :test #'equal) nil)
        (T (push fact *fact-list*)
          (display-results fact)
          T)))
    ```
Function calls in Forward.lisp

1. FORWARD
   ↓
2. EXECUTE-RULE
   ↓
3. EVAL-RULE-F
   → SATISFIABLE-IFS
   ↓
   → DERIVE-NEW-FACT?
   ↓
   → ADD-NEW-FACT?