Overview

MIDTERM: Friday 3/1 (next meeting)

- Project 1 tips: lists, queues, stacks, and trees
- Unification algorithm
- Unification in LISP
- Factors
- Resolvents

---

Project 1 Tips

Queues vs. Stacks:

- Queue: append to the end
  \( \text{(append 'a b c) 'd) -\rightarrow (A B C D) } \)

- Stack: append to the beginning
  \( \text{(cons 'd '(a b c)) or (append '(d) '(a b c)) -\rightarrow (D A B C) } \)

  of push
  \( \text{(setq my-stack '(b c d))} \)
  \( \text{(push 'a my-stack) -\rightarrow (A B C D)} \)

Dequeue (Pop)

- \( \text{(car my-stack) -\rightarrow A} \); non-destructive, my-stack = (A B C D)
- \( \text{(pop my-stack) -\rightarrow A} \); destructive, my-stack = (B C D)

---

Project 1 Tips (cont’d)

- Think in terms of queues and stacks, not in terms of trees, i.e. do not try to implement an elaborate tree structure with pointers etc.
- Try to understand how the General-Search algorithm makes use of the above point.
- For example, when evaluating arithmetic expressions in prefix, postfix, and infix, you don’t need to explicitly represent the parse tree.

---

Unification: Review

- A substitution \( \theta \) is called a unifier for a set \( \{E_1, \ldots, E_k\} \) iff \( E_1 \theta = E_2 \theta = \ldots = E_k \theta \).
- The set \( \{E_1, \ldots, E_k\} \) is said to be unifiable if there is a unifier for it.

---

Notes:

Composition of Substitution

Composition of substitution is simply a serial application of the substitutions in question:

- $E(\theta \circ \lambda) = (E\theta)\lambda$

Example: $\theta = \{x/f(y), y/z\}, \lambda = \{x/a, y/b, z/y\}$

- Thus, $\theta \circ \lambda = \{x/f(b), z/y\}$

- Given $E = P(x, y, z)$,
  - $E\theta = P(f(y), z, z)$
  - $(E\theta)\lambda = P(f(b), y, y) = E(\theta \circ \lambda)$

Disagreement Set

Let $W$ be a nonempty set of expressions $\{E_1, \ldots, E_n\}$. The disagreement set $D$ of $W$ is obtained by locating the first symbol (counting from the left) at which not all the expressions in $W$ have exactly the same symbol, and then extracting from each expression $E_i$ in $W$ the subexpression that begins with the symbol occupying that position.

Example:

$W = \{P(x, y, a, f(x)), P(x, y, a, g(x)), P(x, y, a, z)\}$

Symbols to the right of the vertical bar differ.

$$D = \{f(x), g(x), z\}$$

Unification Algorithm

Let $W = \{E_1, \ldots, E_n\}$ be the set of expressions to be unified.

1. If necessary, rename variables so that no pair $(E_i, E_j)$ from different clauses has any variables in common.

2. Set $k = 0$, $W_k = W$, $\sigma_k = \epsilon$ (empty substitution).

3. If $W_k$ is a singleton (contains only one expr), stop; $\sigma_k$ is a most general unifier for $W$. Otherwise, let $D_k$ be the disagreement set for $W_k$.

4. If there exist elements $v_k$ and $t_k$ in $D_k$ such that $v_k$ is a variable that does not occur in term $t_k$, go to step 5. Otherwise, stop; $W$ is not unifiable.

5. Let $\sigma_{k+1} = \sigma_k \circ \{v_k/t_k\}$ and $W_{k+1} = W_k \{v_k/t_k\}$. (Note that $W_{k+1} = W_k \sigma_{k+1}$)

6. Set $k = k + 1$ and go to step 3.
Unification Theorem

If \( W \) is a finite nonempty unifiable set of expressions, then the unification algorithm will always terminate at step 3, and the last \( \sigma_k \) is a most general unifier for \( W \) (i.e. not unnecessary substitutions).

The algorithm must terminate because each pass through the loop reduces the number of variables by 1, and there are only finitely many of them.

Unification Example

\[ P(x, f(x), z) \text{ vs.} \]
\[ P(g(y), f(g(a)), y) \]

1. \( \{x/g(y)\} \):
   \[ P(g(y), f(g(y)), z) \]
   \[ P(g(y), f(g(a)), y) \]

2. \( \{y/a\} \):
   \[ P(g(a), f(g(a)), z) \]
   \[ P(g(a), f(g(a)), a) \]

3. \( \{z/a\} \):
   \[ P(g(a), f(g(a)), a) \]

Unifier: \( \{x/g(a), y/a, z/a\} \)

Representation of Predicates and Terms in LISP

- Constants: \( a = (A) \text{, Socrates} = \text{(SOCRATES)} \)
- Variables: \( x = X, y = Y \)
- Functions: \( f(x) = (F\ X), f(a,y,z) = (F\ (A)\ Y\ Z) \)
- Predicates: \( P(x) = (P\ X), P(x,b,f(z)) = (P\ X\ (B)\ (F\ Z)) \)

Note how the representation of the constants can come in handy.

SUBLIS : substitution in LISP

(sublis <list-of-alist> <expr>): simultaneous substitution

- **alist**, or association list: \( (A\ .\ B) \), which is the same as \( \text{(cons } A\ \text{' } B) \) (note that B is not a list but an atom in this case).
- <list-of-alist>: a list of \( (<\text{pattern}>\ <\text{replace}>\) pairs.
- <expr>: the expression to be worked on.
- Replace every occurrence of <pattern> in <expr> with <replace>.

Another useful function: (subst <repl> <pattern> <expr>)
SUBLIS Examples

Basically, replace (car alist) with (cdr alist) of each element in the <list-of-alist>:

> (sublis '((x . (20))) '(* x 1))
(* (20) 1)

>(sublis '((x 20)) '(* x 1))
(* (20) 1)

>(sublis '((x 20) (y . 10)) '(* x (/ 5 y)))
(* 20 (/ 5 10))

Unification in LISP

(defun unify (u v)
  (let (($u* (copy-tree u))
        ($v* (copy-tree v)) *subs*)
    (declare (special $u* $v* *subs*)
      (if (unifyb $u* $v*) (or *subs* (list (cons t t)))))
    )
)

(defun unifyb (u v)
  (cond ((eq u v))
    ((symbolp u) (varunify v u))
    ((symbolp v) (varunify u v))
    ((and (consp u) (consp v)
      (eq (car u) (car v))
      (eql (length (cdr u))
           (length (cdr v))))
      (every #'unifyb (cdr u) (cdr v)))))

Unification in LISP (cont’d)

(defun varunify (term var)
  (declare (special $u* $v* *subs*)
    (unless (occurs var term)
      (dolist (pair *subs*)
        (setf (cdr pair)
          (subst term var (cdr pair))))
      (nsubst term var $u*)
      (nsubst term var $v*)
      (push (cons var term) *subs*))))


UNIFY : examples

(unify '(p x) '(p (a)))
(unify '(p (a)) '(p x))
(unify '(p x (g x) (g (b))) '(p (f y) z y))
(unify '(p (g x) (h w) w) '(p y (h y) (g (a))))
(unify '(p (f x) (g (f (a))) x) '(p y (g y) (b)))
(unify '(p x) '(p (a) (b)))
(unify '(p x (f x)) '(p (f y) y))

13

14

15

16
Resolution in Predicate Calculus

- Factors
- Binary resolvent
- Properties of resolution

Factor of a Clause

**Definition:** If two or more literals of a clause $C$ (with the same sign) have a most general unifier $\sigma$, then $C\sigma$ is called a **Factor** of $C$. If $C\sigma$ is a unit clause, it is called a **Unit Factor** of $C$.

**Example:** $C = P(x) \lor P(f(y)) \lor \neg Q(x)$.

- The first two literals have a unifier $\sigma = \{x/f(y)\}$, so $C$ has a factor $C\sigma = P(f(y)) \lor \neg Q(f(y))$.

**Note:** Factors of a clause are much succinct and when two clauses $C_1$ and $C_2$ cannot be resolved directly, their factors (let’s call them $C'_1$ and $C'_2$) can be resolved.

Resolving Two Clauses

**Definition:** Let $C_1$ and $C_2$ be two clauses (called parent clauses) with no variables in common, and with complementary literals $L_1$ and $L_2$ such that $L_1$ and $\neg L_2$ have a most general unifier $\sigma$. Then the clause

$$(C_1 \sigma - L_1 \sigma) \cup (C_2 \sigma - L_2 \sigma)$$

is called a **binary resolvent** of $C_1$ and $C_2$. The literals $L_1$ and $L_2$ are called the **literals resolved upon**.

**Note:** A clause can be treated as a set of literals.

$$\{P(x)\} \cup \{Q(x)\} = \{P(x), Q(x)\} = P(x) \lor Q(x)$$

**Example:** Resolve the following (hint: $\sigma = \{x/a\}$)

$C_1 = P(x) \lor Q(x)$ and $C_2 = \neg P(a) \lor R(y)$.

Resolvent

**Definition:** A **resolvent** of parent clauses $C_1$ and $C_2$ is one of the following binary resolvents:

1. a binary resolvent of $C_1$ and $C_2$
2. a binary resolvent of $C_1$ and a factor of $C_2$
3. a binary resolvent of a factor of $C_1$ and $C_2$
4. a binary resolvent of a factor of $C_1$ and a factor of $C_2$

**Example:** resolve the two clauses
1. $C_1 = P(x) \lor P(f(y)) \lor R(g(y))$ and
2. $C_2 = \neg P(f(g(a))) \lor Q(b)$.

(hint: resolve the factor of $C_1$ and clause $C_2$)
Property of Resolution for First-Order Logic

- **Complete**: If a set of clauses \( S \) is unsatisfiable, resolution will eventually derive \( \mathbf{F} \).
  - *Everything that is true can be proved (eventually).*

- **Sound**: If \( \mathbf{F} \) is derived by resolution, then the original set of clauses \( S \) is unsatisfiable.
  - *Everything that is proved is true.*

Weakness of Resolution

Basically, resolution tries to derive

\[
\text{Axioms} \land \neg \text{Theorem} = \mathbf{F}
\]

- Is there a \( \mathbf{F} \) in the axioms? If there is, the whole formula will always be unsatisfiable no matter what.
- Can we tell whether axioms alone can derive \( \mathbf{F} \) ? (generally, this is not the case)

Key Points

- unification algorithm
- factors: definition, and how to derive, why factors are important
- resolvent: definition, and how to derive

Next Time

- Resolution: a full example
- Automating resolution: various strategies
- Uncertainty: chapter 14