Associative Database Managment

WIlensky Chapter 22

© Gunnar Gotshalks

Associative Database

- An **associative database** is a collection of facts retrievable by their contents
  - » **Is a poodle a dog? Which people does Alice manage?**
  - » **As opposed to retrieving facts by their position in the data base**
    - > **Give me the 10'th fact**
      What is the 10'th fact ?
- The facts in a database can be stored as patterns
- We can use a pattern matcher to search for facts in a database
  - » **Match a query pattern against the patterns in the database looking for one or more matches**

© Gunnar Gotshalks

Example database facts

- Simple facts have no variables

  (dog fido) — **fido is a dog**

  (loves John Mary) — **John loves Mary**

- Can have more complex facts

  (implies (dog ?x) (animal ?x))

  x is a dog implies x is an animal

  (loves ?x ?x)

  A person loves themselves

- One has to carefully consider how to represent facts. In the Lisp world it is customary to have the first item on a list be the main predicate and the remaining items be the arguments to the predicate

© Gunnar Gotshalks

Example queries

- Queries are patterns themselves – they can be without variables
  - **Is fido a dog?** **(dog fido)**
  - **Does John love Mary?** **(loves John Mary)**
- Can have more complex queries
  - **What is fido?**
    (?what fido)
  - **Who does John love?**
    (loves John ?who)
  - **Who loves whom?**
    (loves ?who ?whom)

© Gunnar Gotshalks

Implementation

- In designing a database we need to consider how the facts will be stored
- In our first implementation the facts are all stored in a list.

```
( dog  fido )
( loves  John  Mary )
( implies  ( dog  ?x )  ( animal  ?x ) )
( loves  ?x  ?x )

-->

( ( loves  ( *var*  #:var12 )  ( *var*  #:var12 ) )
  ( implies  ( dog  ( *var*  #:var11 ) )
  ( animal  ( *var*  #:var11 ) ) )
  ( loves  john  mary )
  ( dog  fido )
)
```

© Gunnar Gotshalks

Add to the database

- Store the database as the value of a symbol.
  - **Want to pass an unevaluated pattern and unevaluated symbol to our add operation**
    - **Use a macro**
    - **Change the value of the symbol to update the database**
    - **Replace the names of the pattern matching variables to be unique**

```
(defmacro  add-to-data-base  ( item  d-b-name )
 `( setq  ,d-b-name
          ( cons  ( replace-variables  ( quote  ,item ) )
                  ,d-b-name ) ) )
```

© Gunnar Gotshalks

Replace variable names

- Replace the variables names in **item**
  - > **Replacing variables names needs to be done consistently.**
  - > **Create a binding list that keeps track of renaming.**
  - > **Start off with a nil binding**
  - > **Returns the rebuilt item and the bindings of old and new variable names**

```
(defun  replace-variables  ( item )
 ( values  ( replace-variables-with-bindings  item  nil )))
```

© Gunnar Gotshalks

Replace variable names using bindings

◊ Use the current bindings to replace variables consistently

```
(defun replace-variables-with-bindings
        ( item  bindings )

  > For an atom nothing to replace

  ( cond  ( ( atom  item )  ( values  item  bindings ) )

  > For a pattern variable return a replacement, if necessary

     ((pattern-var-p item)
      (let ((var-binding (get-binding item bindings)))
        (if var-binding   ; if on binding list return the binding
          (values var-binding bindings)

                          ; else generate a new symbol

          (let ((newvar (list  '*var*  (gensym "VAR"))))
            (values newvar (add-binding item newvar
                                        bindings))))))
```

© Gunnar Gotshalks

Replace variable names using bindings – 2

> Item is neither an atom nor a pattern variable – use recursion

> Have to remember bindings from the "car" recursion for the "cdr" recursion

```
(t ( multiple-value-bind ( newlhs lhsbindings )
     (replace-variables-with-bindings
        ( car item )  bindings )
     ( multiple-value-bind  ( newrhs  finalbindings )
        ( replace-variables-with-bindings
           ( cdr item )  lhsbindings )
        ( values  ( cons  newlhs  newrhs )
                  finalbindings ))))
))
```

© Gunnar Gotshalks

Replace variable examples

```
(replace-variables '(loves john mary))
   -->  (LOVES JOHN MARY)

(replace-variables '(loves ?x ?x))
   --> (LOVES (*VAR* #:VAR20) (*VAR* #:VAR20))
```

© Gunnar Gotshalks

Start a database

```
( setq  DB  nil )
( add-to-data-base  ( loves  john  mary )  DB )
--> (  ( loves  john  mary )  )

( add-to-data-base  ( loves  ?x  ?x )  DB )
--> (  ( loves  ( *var*  #:var22 )  ( *var*  #:var22 ) )
       ( loves  john  mary )  )

(add-to-data-base (dog fido) DB)
--> (  ( dog  fido )
       (  ( loves  ( *var*  #:var22 )  ( *var*  #:var22 ) )
       ( loves  john  mary )  )
```

© Gunnar Gotshalks

Query the data base

◊ Use the matcher program to query the database

> Returns a list of bindings that match

```
(defun  query  ( request  data-base )
 ( mapcan
     #'( lambda  ( item )
          ( multiple-value-bind  ( flag  bindings )
                                 ( match  item  request )
          ( if  flag  ( list  bindings ))))
  data-base )
)
```

> mapcan is like mapcar except it uses nconc in place of append

> nconc is a destructive replacement of the cdr part of a cell for speed

© Gunnar Gotshalks

Example queries

```
(query  '( fido  dog )  DB )     ; not in database
-->   nil

( query  '( dog  fido )  DB )     ; in DB - no variables
-->   ( nil )

(query '(loves john john) DB)   ; in DB - hidden variables
-->  ( (   ( ( *var*  #:var22 )  john )  ) )

( query  '( dog  ?name )  DB )  ; Variable in query
-->  ( (   ( ( *var*  name )  fido )  ) )

( query  '( loves  ?x  ?y )  DB )  ; Multiple matches
-->  ( (  ( (*var*  x ) (*var*  y) )  ( (*var*  #:var61 ) (*var* x ) ) )
       (  ( (*var* y ) mary )  ( (*var* x)  john ) )   )
```

© Gunnar Gotshalks

Implementation

- Previous implementation becomes slow as the database increases in size.
  - » **Why?**

© Gunnar Gotshalks

Implementation – 2

- Previous implementation becomes slow as the database increases in size.
  - » **Search is O(n) – where n is the number of facts**
- What can we do to increase the speed?

© Gunnar Gotshalks

Implementation – 3

- Reduce search time by indexing the facts
  - **Put facts with different predicates on different lists**
  - **Put facts with the same predicate on the same list**
  - **Search significantly shorter lists by only searching lists that match the predicate in the query**
- What lists can we use?

© Gunnar Gotshalks

Implementation – 4

- The fact lists are put on the property list of the predicate with the key being the database symbol
  - **Facts could be in some databases and not in others**

© Gunnar Gotshalks

Indexing example

- Enter the following into the indexed database

```
(index  '( loves  john  mary )  'DB )
(index  '( loves  ?x  ?x )  'DB )
(index  '( person  john )  'DB )
(index  '( poodle  fido )  'DB )
```

- Then look at the property lists for the predicates

```
( symbol-plist  'person )  -->  ( db  ( ( person  john )))
( symbol-plist  'poodle )  -->  ( db  ( ( poodle  fido )))
( symbol-plist  'loves )   -->
    ( db  ( ( loves  (*var*  #:var13 ) (*var*  #:var13 ) )
            ( loves  john  mary ) ) )
```

© Gunnar Gotshalks

Other index lists

- The previous examples assumed facts would begin with an atom
- With what else could a fact begin?
- What do we do about it?

© Gunnar Gotshalks

Other index lists – 2

- What if a fact begins with a list?
  - > **For example, could represent "if x is a man then x is mortal" as the following (--> is a valid symbol in Lisp)**

((?x man) --> (?x mortal))

  - > **Have the special atom *list* to hold such facts**

© Gunnar Gotshalks

Other index lists – 3

- What if a fact begins with a variable?
  - > **"everyone loves Barney" could be encoded as**

(?x loves Barney)

  - > **Have the special atom *var* to hold such facts**

© Gunnar Gotshalks

What about searching the entire DB?

- If we have a query that begins with a variable, then the variable could match a variable, a list or any atom. Hence the entire data base would need to be searched.
- How can we do this if the database is scattered across the property lists of many symbols?

© Gunnar Gotshalks

What about searching the entire DB? – 2

- If we have a query that begins with a variable, then the variable could match a variable, a list or any atom. Hence the entire data base would need to be searched.
- How can we do this if the database is scattered across the property lists of many symbols?
- Have to keep track of the index symbols with the symbol for the database
  - > **Add to the property list for the database symbol the list of *keys* that have been used as indices.**
  - > **In the example, several slides back, you could look at the symbol list for DB**

(symbol-plist 'DB) --> (*keys* (poodle person loves))

© Gunnar Gotshalks

Index function for a database

```
(defun  index  ( item  data-base )
  > place is where we want to store the item – use the key
    for the pattern
  (let ( ( place ( cond ( ( atom  ( car  item ) )  ( car  item ) )
                       ( ( pattern-var-p  ( car  item ) )  '*var* )
                       ( t  '*list* ) ) ) )
  > Store the item itself
     (setf  ( get  place  data-base )
        ( cons  ( replace-variables  item )  ; rename variables
                ( get  place  data-base ) ) )
  > Store the key for the item – adjoin adds only if not there
     (setf  ( get  data-base  '*keys* )
            ( adjoin  place  ( get data-base  '*keys* ))  )))
```

© Gunnar Gotshalks

Fast query

```
(defun fast-query (request data-base)
 (if (pattern-var-p (car request))
    (mapcan  #'(lambda (key)    ; Search entire DB
                  (query request (get key data-base)))
             (get data-base '*keys*))
    (nconc
 > else search under "atom" or *list*
     (query request  (get   (if (atom (car request))
                            (car request)    '*list*)
                       data-base)
     )
 > Add in search under *var* if "atom" or *list*
   search
     (query  request  (get '*var* data-base)))))
```

© Gunnar Gotshalks

Deductive retrieval

- We use **backward chaining**
- Store implications in the database in the following form

  (<- consequent antecedent)
- In addition to querying the database in the normal way we add the following query

  (<- request antecedent)
- If the second query suceeds we recursively query using the returned antecedent as a new request

  (<- previous-antecedent antecedent)
- And so on – we proceed backwards from the query to the **base facts**

© Gunnar Gotshalks

Deductive retrieval example

- Let's add the following to the database

```
(index  '( <-  ( mammal  ?x )  ( dog  ?x ) )  'DB )
(index  '( <-  ( dog  ?x )  ( poodle  ?x ) )  'DB )
(index  '( poodle  fido )  'DB )
```

- And make the following query

```
( mammal  fido )
```

  > matches fact 1 using the implication search with antecedent --> (dog fido)

- Make the recursive query – matches fact 2

```
antecedent  -->  ( poodle fido )
```

- Make the recursive query - matches fact 3

  » return success ; no further recursion

© Gunnar Gotshalks

Deductive retrieval function – 1

```
(defun  retrieve  ( request  data-base )
  > Combine a regular seach
  ( nconc  ( fast-query  request  data-base )
  > with a recursive search over the implications
       (mapcan
                ... the function to apply to the
                    implication search ...
  > Get the next level of implication search – note
    the use of a macro to construct the pattern to
    use for the search
          (fast-query  `( <-   ,request   ?antecedent )
                       data-base )
)))
```

© Gunnar Gotshalks

Deductive retrieval function – 2

... the function to apply to the implication search ...

```
#'( lambda  ( bindings )
```

> Search for each of the bindings of antecedent and add to the list of bindings

```
    ( mapcar  #'( lambda  ( rbindings )
                  ( append  rbindings   bindings ) )
```

> Recursive search on an antecedent. Need to replace the variables in antecedent with their values, if any

```
        (retrieve  ( substitute-vars
                       ( get-binding   '?antecedent
                                       bindings )
                       bindings)
                     data-base)
    ))
```

© Gunnar Gotshalks

Substituting variables

- Suppose we have the following binding list

  ((?antecedent (loves john ?y)) (?y ?z) (?z mary))

- We do not want to search for the more general

  (loves john ?y)

- Because we have bindings that restrict the value of **?y**
- A first level substitution for **?z --> ?y** yields a search pattern of

  (loves john ?z)

- But this is still too general as we have a binding for **?z**
- Need to do a second level, **?mary --> ?z**, **recursive substitution** to get the pattern we want to search on

  (loves john mary)

© Gunnar Gotshalks

Substitute variables for deductive retrieval

```
(defun substitute-vars (item bindings)
  > Nothing to do if item is an atom
  (cond ((atom item) item)
  > Potential substitution if a variable
        ((pattern-var-p item)
          (let ((binding (get-binding item bindings)))
  > Substitute only if we have a binding for the item
              (if binding
                  (substitute-vars binding bindings)
                  item)))
  > Have a list, so recursively substitute on first and rest
        (t (cons (substitute-vars (car item) bindings)
                 (substitute-vars (cdr item) bindings)))))
```

© Gunnar Gotshalks