GATE user's manual

Version 2.2

by Erik T. Mueller

Introduction

GATE is an AI development tool for Common Lisp. GATE includes slot-filler objects, a type hierarchy, unification, instantiation, variabilization, theorem proving, and a context mechanism.

GATE has been used in the construction of three AI programs: Daydreamer (Mueller, 1990), a computer model of human daydreaming, RINA (Zernik & Dyer, 1987), a program for learning new English idioms in context, and OpEd (Alvarado, 1990), an editorial comprehension program.

A tutorial introduction to GATE

This section provides a tutorial introduction to GATE. It walks the new user through most of the important capabilities of GATE.

Loading

Start up Common Lisp and `cd` to the `src` subdirectory of the downloaded Daydreamer/GATE distribution. Then type:

```
(load "load.cl")
```

The system will ask you whether you want to load Daydreamer or GATE. Once GATE is loaded a message such as the following will be printed:

```
======================
Welcome to GATE 2.2, Common Lisp version of 19990506
Bugs/problems/questions to erik@panix.com
======================
```

Obs and types

The basic data objects of GATE are called *obs*. Obs are similar to records or structures in programming languages such as Pascal, C, and Common Lisp, but with additional features useful for AI programming.

Obs are used to represent the entities of interest in your program. For example, suppose you want to represent a person. First, define a *type* called `PERSON`, having *slots* for name, age, and occupation:

```
> (ty$fcreate 'PERSON nil '(name age occupation))
#{PERSON}
>
```

(The case of input is ignored by GATE. However, by convention, types are in all upper case, slots are in lower case, and obnames are in upper and lower case.) Next, create some obs which are *instances* of this type:

```
> (ob$fcreate '(PERSON name "Karen"
                       age 27
                       occupation 'DOCTOR
                       obname Karen1))
#{KAREN1: (PERSON name "Karen" age 27...)}
> (ob$fcreate '(PERSON name "Jim"
                       age 31
                       occupation 'COMPOSER
                       obname Jim1))
#{JIM1: (PERSON name "Jim" age 31 occupation ...)}
>
```

The optional `obname` slot enables you to specify a name for an ob. You can then refer to the ob by name:

```
> ^Karen1
#{KAREN1: (PERSON name "Karen" age 27...)}
> ^Jim1
#{JIM1: (PERSON name "Jim" age 31 occupation ...)}
>
```

You can print an ob in its entirety:

```
> (po ^Karen1)
(PERSON name "Karen"
        age 27
        occupation 'DOCTOR)
> (po ^Jim1)
(PERSON name "Jim"
        age 31
        occupation 'COMPOSER)
>
```

You can get the *values* of particular slots:

```
> (ob$get ^Karen1 'name)
"Karen"
> (ob$get ^Jim1 'age)
31
> (ob$get ^Jim1 'occupation)
COMPOSER
>
```

You can also set the value of a slot:

```
> (ob$set ^Jim1 'age 32)
32
> (po ^Jim1)
(PERSON name "Jim"
        age 32
        occupation 'COMPOSER)
>
```

Now suppose you want to represent simple world actions, as in Conceptual Dependency (CD) notation

(Schank & Abelson, 1977). First, define a type called ACTION:

```
> (ty$fcreate 'ACTION nil '(actor from to obj))
#{ACTION}
>
```

Then, define *subtypes* of ACTION called ATRANS, MTRANS, and PTRANS:

```
> (ty$fcreate 'ATRANS '(ACTION) nil)
#{ATRANS}
> (ty$fcreate 'MTRANS '(ACTION) nil)
#{MTRANS}
> (ty$fcreate 'PTRANS '(ACTION) nil)
#{PTRANS}
>
```

In CD, an ATRANS represents transfer of possession of a physical object from one person to another. For example, create an ob to represent the action "Jim gives Karen a copy of Ear Magazine":

```
> (ty$fcreate 'MAGAZINE nil '(name))
#{MAGAZINE}
> (ob$fcreate '(ATRANS actor Jim1
                       from Jim1
                       to Karen1
                       obj (MAGAZINE name "Ear Magazine")
                       obname Atrans1))
#{ATRANS1: (ATRANS actor Jim1 from Jim1...)}
>
```

An MTRANS represents transfer of mental information from one person to another. For example, create an ob to represent "Jim tells Peter that he gave Karen a copy of Ear Magazine":

```
> (ob$fcreate '(MTRANS actor Jim1
                       from Jim1
                       to (PERSON name "Peter" age 26 occupation 'MUSICIAN)
                       obj Atrans1
                       obname Mtrans1))
#{MTRANS1: (MTRANS actor Jim1 from Jim1...)}
>
```

You can interrogate whether an ob is of a given type:

```
> (ty$instance? ^Atrans1 'ACTION)
#T
> (ty$instance? ^Atrans1 'ATRANS)
#T
> (ty$instance? ^Atrans1 'MTRANS)
()
> (ty$instance? ^Atrans1 'PERSON)
()
> (ty$instance? ^Mtrans1 'MTRANS)
#T
>
```

As you can see, an ob of type ATRANS is also considered to be of type ACTION, since ATRANS is a subtype of ACTION.

Unification and instantiation

AI programs are built out of two basic operations: *unification* (pattern matching) and *instantiation*. For example, create a pattern ob as follows:

```
> (lset pattern (ob$fcreate '(MTRANS actor ?Person1
                                    from ?Person1
                                    to ?Person2
                                    obj ?Anything)))
#{OB.60: (MTRANS actor ?Person1 from ......)}
>
```

`?Person1`, `?Person2`, and `?Anything` are *variables*. Two obs unify if values for variables can be found such that substituting the values for those variables in the obs would produce equivalent structures. For example, unify the above pattern with the previously created ob `Mtrans1` as follows:

```
> (lset bd (ob$unify pattern ^Mtrans1 *empty-bd*))
(T (ANYTHING #{ATRANS1: (ATRANS actor Jim1 from Jim1...)})
   (PERSON2 #{OB.50: (PERSON name "Peter" age 26...)})
   (PERSON1 #{JIM1: (PERSON name "Jim" age 32 occupation ...)}))
>
```

The result is a list of the found variable values-called a *binding list*.

Instantiation creates a copy of an ob (and embedded obs) in which all variables have been replaced by their values specified by a given binding list. For example:

```
> (lset instan-ob (ob$instantiate pattern bd))
#{OB.61: (MTRANS actor Jim1 from Jim1...)}
> (po instan-ob)
(MTRANS actor Jim1
        from Jim1
        to (PERSON name "Peter"
                   age 26
                   occupation 'MUSICIAN)
        obj Atrans1)
>
```

Here, you instantiate the pattern with the bindings resulting from the unification of the pattern with the ob `Mtrans1`. This results in an ob similar to `Mtrans1`:

```
> (po ^Mtrans1)
(MTRANS actor Jim1
        from Jim1
        to (PERSON name "Peter"
                   age 26
                   occupation 'MUSICIAN)
        obj Atrans1)
>
```

A simple application: Inferencing

Suppose you wish to build a simple program to generate inferences from CD actions. For example, after an `ATRANS` of an object from one person to another, you would like to infer possession of that object by the other person.

First, define a type for states and a subtype for possession as follows:

```
> (ty$fcreate 'STATE nil nil)
#{STATE}
> (ty$fcreate 'POSS '(STATE) '(actor obj))
#{POSS}
>
```

Next, define a type for inference rules:

```
> (ty$fcreate 'INFERENCE nil '(if then))
#{INFERENCE}
>
```

An inference rule consists of an `if` action and a `then` state.

Next, create a list of inference rules:

```
> (lset *infs*
        (list (ob$fcreate '(INFERENCE if (ATRANS actor ?Person1
                                                   from ?Person1
                                                   to ?Person2
                                                   obj ?Object)
                                         then (POSS actor ?Person2
                                                   obj ?Object)))))
(#{OB.73: (INFERENCE if (ATRANS actor ......)...)})
>
```

So far, the list consists of one rule for inferring possession after an `ATRANS`.

Now, define a function for generating inferences:

```
> (defun forward-inferences (cd)
    (yloop (initial (bd nil) (result nil))
           (yfor inf in *infs*)
           (ydo (if (setq bd (ob$unify (ob$get inf 'if) cd *empty-bd*))
                    (setq result (cons (ob$instantiate (ob$get inf 'then) bd)
                                       result))))
           (yresult result)))
>
```

In order to generate inferences from a given action CD, the function loops through all inference rules. Whenever an inference rule is found whose `if` unifies with the action CD, the `then` of that rule is instantiated with the bindings resulting from the unification and added to a list of inferred states. This list is returned at the end.

Thus, for example, this function produces the following result when applied to `Atrans1`:

```
> (lset states (forward-inferences ^Atrans1))
(#{OB.79: (POSS actor Karen1 obj (MAGAZINE...))})
> (po (car states))
(POSS actor Karen1
      obj (MAGAZINE name "Ear Magazine"))
>
```

Summary

The eight most common functions of GATE are:

1. Create type:

   ```
   (ty$fcreate type-name supertypes slots) -> type
   ```

2. Create ob:

   ```
   (ob$fcreate oblist) -> ob
   ```

3. Get slot value:

   ```
   (ob$get ob slot) -> value
   ```

4. Set slot value:

   ```
   (ob$set ob slot value) -> value
   ```

5. Print ob:

   ```
   (po ob)
   ```

6. Unify:

   ```
   (ob$unify ob1 ob2 bindings) -> bindings
   ```

7. Instantiate:

   ```
   (ob$instantiate ob bindings) -> ob
   ```

8. Determine if type:

   ```
   (ty$instance? ob type-name) -> boolean
   ```

Reference manual

This section forms a reference manual on GATE.

Obs

The basic data structure of GATE is called the *ob*. Obs are similar to the slot-filler objects of Schank and Riesbeck (1981), frames (Minsky, 1975), Lisp a-lists (McCarthy et al., 1965), and the structures or records of traditional programming languages such as Pascal (Wirth, 1971).

An ob consists of:

- one or more *obnames*,
- an optional *type*,
- zero or more pairs, where each pair consists of a *slot name* and a *slot value*.

Obnames and slot names are Lisp atoms. A slot value is either an ob or some other Lisp object (such as a character string or a function). Several pairs with the same slot name are permitted. Only one ob may have a given obname.

The following functions are used to create an ob:

- `(ob$create-empty) -> ob`

  Create and return a new empty ob.

- `(ob$create-named-empty obname) -> ob`

  Create and return a new empty ob with the specified obname.

- `(ob$fcreate list) -> ob`

  Create and return a new ob according to a given list which specifies the type, slot names and values of the ob. (See the section below on printing and reading obs for the form of this list. See also the tutorial above for examples.) This function also creates any enclosed obs.

The following functions deal with obnames:

- `(ob$add-name ob obname) -> obname`

  Add an obname to a given ob.

- `(ob$names ob) -> obnames`

  Return a list of the obnames of a given ob.

- `(ob$name->ob obname) -> ob or NIL`

  Return the ob having the given obname, or `NIL` if no ob has that name.

The following functions deal with types:

- `(ob$ty ob) -> ty`

  Return the type of a given ob.

The following functions manipulate pairs of an ob:

- `(ob$add ob slot-name slot-value) -> slot-value`

  Add a pair consisting of the given slot-name and slot-value to a given ob.

- `(ob$remove ob slot-name slot-value)`

Remove a pair consisting of the given slot-name and slot-value from a given ob (or signal an error if there is no such pair).

- `(ob$gets ob slot-name) -> slot-values`

  Return a (possibly `NIL`) list of the slot values of those pairs of a given ob whose slot name is the given slot-name.

- `(ob$get ob slot-name) -> slot-value`

  Return the slot value of an arbitrary pair of a given ob whose slot name is the given slot name, or `NIL` if there are no such pairs.

- `(ob$pairs ob) -> list`

  Return a (possibly `NIL`) list of all the pairs of a given ob.

The following functions are similar to the above functions, except that a *path* may be specified in order to traverse a path starting from the top-level ob to reach a pair in an embedded ob:

- `(ob$padd ob slot-path slot-value) -> slot-value`

  Add a slot-value according to the given slot-name path to a given ob.

- `(ob$premove ob slot-path slot-value)`

  Remove the pair according to the given slot-name path and slot-value from a given ob (or signal an error if there is no such pair).

- `(ob$pget ob slot-path) -> slot-value`

  Return the slot value of an arbitrary pair of a given ob according to the given slot-name path, or `NIL` if there are no such pairs.

Other functions dealing with obs are as follows:

- `(ob? obj) -> boolean`

  Given an arbitrary Lisp object, return `T` if it is an ob, otherwise return `NIL`.

- `(ob$copy ob) -> ob`

  Return a new ob having the same type and pairs (but not obname) as a given ob.

Printing and reading obs

The following functions deal with ob printing and reading:

- `(ob$print ob stream)`

  Print a given ob on a given stream.

- `(po ob)`

  Print a given ob on the standard output.

- `(ob$fread stream) -> ob`

  Read and create an ob from a given stream.

- `(ob$fcreate list) -> ob`

  Read and create an ob according to the given list.

Obs have *textual representations* for printing and reading. The textual representation for printing obs is of the following form:

```
(TYPE slot-name1 slot-value1a slot-value1b ...
      slot-name2 slot-value2a slot-value2b ...
      ...)
```

Rather than displaying each pair separately, all of the slot values of those pairs having the same slot name are displayed together in a single line (space permitting). Slot values may be obs themselves. In printing, if the slot value ob has a (non-automatically generated) name, only the name is printed; otherwise, the full textual representation is recursively printed. Slot values which are Lisp objects other than numbers and strings (such as atoms, lists, and functions) appear quoted in the ob textual representation.

Type names, slot names, and obnames are all represented as Lisp atoms. Although Lisp atoms are normally printed in uppercase, these ob entities are printed in special cases to distinguish them from other atoms: types are displayed in uppercase; slot names are displayed in lowercase; obnames are displayed in a capitalized lowercase.

For example, consider the following ob:

```
(PTRANS actor John1
        from (RESIDENCE obj John1)
        to Store1
        obj John1 Mary1)
```

The ob is of type `PTRANS` and consists of 5 pairs. One pair of the ob consists of the slot name `actor` and slot value `John1`. This slot value refers to another ob whose name is `John1`. The slot value of the pair whose slot name is `from` is another ob which does not have a name. The textual representation of this ob thus appears recursively in the textual representation of the enclosing ob. There are two pairs of the main ob whose slot name is `obj`. The slot value of one pair is `John1` while the slot value of the other is `Mary1`. Both pairs are displayed in a single line.

The textual representation for reading obs is similar to the representation for printing, with the following

differences:

- Only one slot value is permitted per slot. If you wish to specify several pairs, you must specify each slot name and value separately.
- The value of the `obname` slot may be used to indicate the name of the ob.
- An atom slot value refers to an existing ob of that name, while a full textual representation results in the creation of a new ob.

Types

Types are organized into a hierarchy. The purpose of types is to enable classification of obs and to specify their textual representation.

The following functions manipulate types:

- `(ty$fcreate type-name supertype-names slots) -> type`

  Create a new type having the given supertypes and slots.

- `(ty$instance? ob type-name) -> boolean`

  Given an ob, return `T` if it is an instance of the specified type (directly or indirectly through inheritance), otherwise return `NIL`.

- `(ty$supertypes type) -> types`

  Return the parent types of the given type.

- `(ty$subtypes type) -> types`

  Return the children types of the given type.

- `(ty$supertypes* type) -> types`

  Return the improper ancestor types of the given type.

- `(ty$subtypes* type) -> types`

  Return the improper descendant types of the given type.

- `(ty$least-common-supertype type1 type2) -> type`

  Return the least common supertype of two types.

Unification

Unification is a pattern-matching operation performed on two obs. The obs may contain a special kind of ob called a *variable*. Two obs unify if values for variables can be found such that substituting the

values for those variables in the obs would produce equivalent structures. For example, if

```
(PTRANS actor ?Person
        to ?Location)
```

is unified with

```
(PTRANS actor John1
        to Store1)
```

the resulting variable *binding list* is:

```
(T (PERSON #{JOHN1: (PERSON)})
   (LOCATION #{STORE1: (STORE)}))
```

The GATE unification algorithm is based on previous unifiers (Schank & Riesbeck, 1981; Charniak, Riesbeck, & McDermott, 1980) with appropriate extensions for typed variables and obs, multiple slot values per slot name, special obs, and cyclic data structures. The following functions deal with unification:

- `(ob$unify ob1 ob2 bd) -> bd or NIL`

  Given two obs and a binding list, attempt to unify the two obs. If unification is successful, the original binding list augmented with new variable values is returned. Otherwise, `NIL` is returned.

- `(ob$unify1 ob1 ob2 bd ignore-slots) -> bd or NIL`

  Same as above except the specified slots are ignored in the unification process.

The structures resulting from substitution do not actually have to be equivalent. Rather, one structure must be a substructure of the other: one ob unifies with another if each pair in the first ob unifies with a unique pair in the second ob; however, each pair in the second ob need not have been accounted for. For example,

```
(PTRANS actor ?Person)
```

will unify with

```
(PTRANS actor John1
        to Store1)
```

but

```
(PTRANS actor ?Person
        from ?Location1
        to ?Location2)
```

will not unify with

```
(PTRANS actor John1
        to Store1)
```

Thus unification in GATE is asymmetrical.

Binding lists and variables

The following operations on binding lists are provided:

- `bd-create -> bd`

  Create an empty binding list.

- `(bd-bind variable-name value bd) -> bd`

  Bind a given variable to a new value returning a new binding list. This is sometimes called *augmenting* the binding list with a new binding.

- `(bd-lookup variable-name bd) -> value`

  Look up the value of the given variable in the binding list. Return `NIL` if no value is found.

Variables are obs of the following form:

```
(UVAR name name
      unification-type type)
```

An unbound variable unifies with an ob if the ob is an instance of that variable's unification type. When an unbound variable successfully unifies with an ob, that variable is bound to the ob. A bound variable unifies with an ob if the value of that variable unifies with the ob. Two variables unify if one unification type is an improper supertype of the other.

Note that the `bd-bind` and `bd-lookup` functions do not take variable obs, but rather their names. To get the name of a given variable ob, use the function `variable-name`.

Variables are not normally represented textually as above, but rather in the following form:

```
?name:TYPE
```

In addition, a short form is available in which the type of the variable is implicitly specified by its name. The following variables are of type `PERSON`:

```
?Self ?Other ?Person ?Person1 ?Person2 ?Person3 ...
```

The full representation of `?Self`, for example, is:

```
(UVAR name Self
      type PERSON)
```

The following variables are of type `PHYS-OBJ`:

```
?Phys-Obj ?Phys-Obj1 ?Phys-Obj2 ...
```

The following variables are of type `LOCATION`:

`?Location ?Location1 ?Location2 ...`

Other variables follow a similar convention; this works for any defined type.

Variables which unify with anything (including obs and Lisp objects) are specified by:

`?name:NOTYPE`

Unnamed variables (which do not have values) are specified by the following forms:

`?:TYPE ??`

Special obs

In addition to variables, a set of *special* obs provide an extended syntax and semantics for unification. These features were inspired by the pattern matcher of the DIRECTOR language (Kahn, 1978). We have extended the constructs to full unification-that is, the constructs may now be used in both arguments to the matcher, with a well-defined semantics.

The special obs are as follows:

`(UAND obj` *`obj1 obj2 ...`*`)`

A `UAND` ob unifies with another object if all of *obj1 obj2* ... unify with that object. Bindings are augmented in a cumulative manner from each unification.

`(UOR obj` *`obj1 obj2 ...`*`)`

A `UOR` ob unifies with another object if any of *obj1 obj2* ... unifies with that object. Bindings are augmented by the first successful unification.

`(UNOT obj` *`obj`*`)`

A `UNOT` ob unifies with another object if *obj* does not unify with that object. Bindings are not augmented.

`(UPROC proc` *`proc`*`)`

A `UPROC` ob unifies with another object if *proc* (a Lisp lambda expression) applied to that object returns a non-`NIL` value. Bindings are not augmented.

What if both arguments to unification are special obs? Although this case may appear to be accounted for in the above definitions through appropriate recursion, there are certain subtleties which merit examination. We consider each possibility in turn.

If both obs are of type `UAND`, then every element of one ob must unify successfully with every element of the other-that is, unification succeeds if every pair in the *cross product* of the elements of the two obs unifies successfully. Each unification is performed in the context of bindings which have been accumulated in previous unifications. For example, if the special ob

`(UAND obj` *`obj1 obj2`*`)`

is unified with

`(UAND obj` *`obj3 obj4`*`)`

then *obj1* will first be unified with *obj3*. If this unification is successful, the resulting bindings are employed in the unification of *obj1* with *obj4*. If this unification is successful, the resulting bindings are then employed in the unification of *obj2* with *obj3*. The resulting bindings are similarly employed in the final unification of *obj2* with *obj4*. The final bindings are the result of the unification of the two `UAND`s. If at any point a unification fails, then the result of the unification of the two `UAND`s is `NIL`.

If both obs are of type `UOR`, then some element of one ob must unify successfully with some element of the other-that is, unification succeeds if any pair in the cross product of the elements of the two obs unifies successfully. The bindings which result from such a unification are simply the bindings passed to the algorithm augmented with the bindings resulting from a single unification (of one of the pairs in the cross product).

If both obs are of type `UNOT`, then unification succeeds if the (one and only) element of one ob unifies with the element of the other. However, no new bindings result in the process. Thus in this case a simple yes-no result is given, without specification of the bindings necessary to unify the elements of the two obs.

If both obs are of type `UPROC`, one function is simply invoked on the other (and unification is successful if the function returns a non-`NIL` value)-it is assumed that the functions are able to handle such a case. It is sometimes safe for such functions simply to return `T` if invoked on a function. If it is desired for such a case to be undefined, functions may be written to produce error messages if invoked on another function.

What, now, if the two special obs are of different type? How do different special obs interact with one another? If the first ob is of type `UOR` and the second is of type `UAND`, then unification is successful if there is an element of the first ob which unifies with every element of the second-that is, unification succeeds if any row of the cross product of the elements of the two obs consists of successful unifications. The bindings which result from such a unification are accumulated from that row.

If the first ob is of type `UAND` and the second is of type `UOR` (i.e., the situation is the other way around), unification is as above with the first and second obs switched (in recursive unification of components, however, it is necessary to switch the order of arguments once again so that the appropriate asymmetric behavior with respect to the first and second arguments is retained).

If the first ob is of type `UOR` and the second is of type `UNOT`, then unification is successful if there is an element of the first ob which does not unify with the (one and only) element of the second-that is, unification succeeds if any pair of the cross product of the elements of the two obs (which in this case is the cross product of a many-element vector with a one-element vector) consists of an unsuccessful unification. No new bindings result from such a unification.

If the first ob is of type `UNOT` and the second is of type `UOR`, the obs are switched and the above algorithm is employed (with appropriate switching for recursive unifications).

If the first ob is of type `UAND` and the second is of type `UNOT`, then unification is successful if no element of the first ob unifies with the (one and only) element of the second-that is, unification succeeds if every

pair of the cross product of the elements of the two obs (again a cross product of a many-element vector with a one-element vector) consists of an unsuccessful unification. No new bindings result from such a unification.

If the first ob is of type `UNOT` and the second is of type `UAND`, the obs are switched and the above algorithm is employed with appropriate switching.

If the first ob is of type `UAND` and the second is of type `UPROC`, then unification is successful if the function of the second ob returns a non-`NIL` value for every element of the first ob.

If the first ob is of type `UOR` and the second is of type `UPROC`, then unification is successful if the function of the second ob returns a non-`NIL` value for any element of the first ob.

If the first ob is of type `UNOT` and the second is of type `UPROC`, then unification is successful if the function of the second ob returns `NIL` when applied to the (single) element of the first ob. In each case, no new bindings result from the unification and the other orders are handled in the usual way (except that no recursive switching is required since `PROC` does not result in recursive unifications).

Note that `UOR` and multiple slot values per slot name introduce the potential for more than one solution (set of bindings) for a given unification. However, only the first solution that is found is returned in the current GATE unifier.

Instantiation

Instantiation takes an ob and a binding list, and returns a copy of the ob in which any variables have been replaced by their values. Unbound variables remain as variables in the copy. The ob may contain other obs-the complete structure with ob as root is copied, with any cycles preserved in the copy. The instantiation function is as follows:

- `(ob$instantiate ob bd) -> ob`

  Instantiate a given ob with the given binding list.

For example, if the ob:

```
(PTRANS actor ?Person
        to ?Location)
```

is instantiated using the binding list:

```
(T (PERSON #{JOHN1: (PERSON)})
   (LOCATION #{STORE1: (STORE)}))
```

the returned ob is:

```
(PTRANS actor John1
        to Store1)
```

Unification composed with instantiation is almost the identity transformation-if one ob unifies successfully with another, and then the first ob is instantiated with the bindings from the unification, the

result will be a copy of the second ob (however with any pairs not referred to in the first ob omitted).

Variabilization

Another pseudo-inverse for instantiation is *variabilization*. Variabilization takes an ob and a predicate and returns a complete copy of the ob (with cycles preserved) in which any enclosed obs answering non-`NIL` to the predicate have been replaced by unique variables. Multiple occurrences of the same ob will become the same variable. Variabilization can be used to perform simple inductive generalization from single examples. The variabilization function is as follows:

- `(ob$varize ob predicate) -> ob`

  Variabilize a given ob according to the given predicate.

For example, given:

```
(PTRANS actor John1
        to Store1
        obj John1 Mary1)
```

and an appropriate predicate, variabilization returns:

```
(PTRANS actor ?Person1
        to Location1
        obj ?Person1 ?Person2)
```

Contexts

A context consists of a collection of obs called *facts* which specify the state of a possible world. Each fact which is in a given context is said to be *true* in that context. Any fact which is not in a given context is said to be *not true* in that context. GATE contexts are similar to other context mechanisms such as OMEGA viewpoints (Barber, 1983) and AP3 contexts (Goldman, 1982), all of which derive from the original contexts of QA4 (Rulifson, Derksen, & Waldinger, 1972).

The following functions manipulate contexts:

- `(cx$create) -> cx`

  Create a new context. Initially, no facts are true in the new context.

- `(cx$assert cx fact)`

  Assert a given fact into a given context. The fact becomes true in the context (whether or not it was already true).

- `(cx$retract cx fact)`

  Retract a given fact from a given context. After this operation, the fact is not true in the context (whether or not it was already not true).

- `(cx$true? cx fact) -> boolean`

  Return `T` if the given fact is true in the given context, otherwise return `NIL`.

- `(cx$retrieve cx ob) -> bds`

  Retrieve all the true facts in the given context which unify with the given pattern ob. A list of binding lists is returned, in which the first element of each binding list is the retrieved fact. (Hashing on the types of facts is employed to improve the efficiency of this function.)

- `(cx$retrieve-bd cx ob bd) -> bds`

  Same as above, except unification proceeds starting from the given bindings list.

- `(cx$sprout cx) -> cx`

  Sprout a new context which is a child of the given parent context. Initially, each fact which is true in the parent context is true in the new child context. However, subsequent asserts in the child context may add new facts to this context just as subsequent retract operations may remove facts. (In the current GATE asserts and retracts do *not* affect the truth status of facts in descendant contexts. That is, there is no dynamic inheritance of the truth status of facts.) The new context is called a *child* of the old context; the old context is called the *parent* of the new context; we also speak of *ancestor* and *descendant* contexts.

- `(cx$children cx) -> cxs`

  Return the children of a given context.

- `(cx$parent cx) -> cx`

  Return the parent of a given context.

- `(cx$ancestors cx) -> cxs`

  Return the proper ancestors of a given context.

- `(cx$descendants cx) -> cxs`

  Return the proper descendants of a given context.

For example, suppose that the following two objects are asserted into a newly created context:

```
(PTRANS actor John1
        to Store1)

(PTRANS actor Mary1
        to Store1)
```

A retrieval from this context using the pattern:

```
(PTRANS actor ?Person
        to Store1)
```

will return the following binding lists:

```
(#{OB.245: (PTRANS actor John1 ...)} (PERSON #{JOHN1: (PERSON)}))
```

and

```
(#{OB.249: (PTRANS actor Mary1 ...)} (PERSON #{MARY1: (PERSON)}))
```

Theorem prover

A Prolog-like theorem prover (Clocksin & Mellish, 1981) is available with GATE. The following functions are provided:

- `(ob$prove1 ob bd max-number prules pfacts-cx ignore-slots) -> bds`

  Prove the given ob, possibly containing variables, with respect to the given binding list, using the given list of proof rules, using the facts contained in a context, and ignoring the given slot names. Generate at most the given number of solutions. Return a list of augmented binding lists, or `NIL` if the proof fails. If the proof fails, a list of unproved facts is returned in the global variable `*PROOF-FAILURES*`.

- `(ob$prove ob bd max-number) -> cxs`

  Same as above, except use the global variables `*PRULES*` and `*PFACTS*` for the proof rules and facts. Do not ignore any slots.

Proof rules are of type `PRULE`. Each proof rule contains a `goal` slot and a `subgoal` slot. The `goal` slot contains a goal ob pattern to be proved. The `subgoal` slot specifies how that goal may be proved:

- if the `subgoal` slot contains a `ROR` ob, *any* of the `obj` values of that ob may (recursively) be proved;
- if the `subgoal` slot contains a `RAND` ob, *all* of the `obj` values of that ob must (recursively) be proved;
- if the `subgoal` slot contains a `RNOT` ob, the `obj` value of that ob must (recursively) not be proved;
- otherwise the `subgoal` slot must (recursively) be proved.

Here is a sample proof rule:

```
(PRULE subgoal (ROR obj (PTRANS actor ?Person
                                to ?Location)
                        (LIVES-IN actor ?Person
                                  loc ?Location))
       goal (PROX actor ?Person
                  loc ?Location))
```

Facts are arbitrary obs. A goal is proved whenever it unifies with a fact. Here is a sample fact:

```
(PTRANS actor John1
```

```
        to Store1)
```

The following goal can be proved using the above proof rule and fact:

```
(PROX actor ?Person
      loc Store1)
```

The following list of binding lists would be returned:

```
((T (PERSON #{JOHN1: (PERSON)})))
```

Other functions

There are many useful functions not documented in this manual. To find out about them, you must look at the source code. Also, understanding exactly how certain functions work may require examination of the source code-especially for unification, instantiation, and variabilization.

History and acknowledgements

GATE was developed at the UCLA artificial intelligence laboratory. The initial inspiration came from a node and link graphics package that Brigham Bell developed at USC/ISI on Symbolics 3600s. In the summer of 1984, I ported this package to the T dialect of Scheme on Apollo workstations, but it ran so slowly it had to be scrapped. Then Perry Busalacchi rewrote the node and link graphics in C, making the first version of GATE possible in October 1984. Seth Goldman wrote routines to make use of Apollo graphics and, along with Eric Preyss, created the T flavors package in which the first version of GATE (Mueller & Zernik, 1984) was implemented. The first version contained a demon programming language by Uri Zernik, but this has since been abandoned. In January 1985, the full unifier was written, replacing the earlier pattern matcher. The context mechanism was added in June 1985, partially in response to Inference Corporation's ART package. The current ob printer and reader were added in November 1985. In January 1986, special unification forms were converted to obs from Lisp objects. In September 1986, GATE was rewritten to be independent of flavors, and its speed increased dramatically. In February 1987, a theorem prover was added. In May 1999, GATE was ported to Common Lisp from T.

GATE was influenced by previous programs created by members of the AIRHEADS group at UCLA, such as Charlie Dolan's T-CD**2 package. It was strongly influenced by feedback from its original users, Uri Zernik, Sergio Alvarado, and Ric Feifer.

Comments, questions, and improvements should be directed to erik@panix.com.

References

Alvarado, S. J. (1990). *Understanding editorial text*. New York: Kluwer.

Barber, G. (1983). Supporting organizational problem solving with a work station. *ACM Transactions on Office Information Systems, 1*(1), 45-67.

Charniak, E., Riesbeck, C. K., & McDermott, D. V. (1980). *Artificial intelligence programming.*

Hillsdale, NJ: Lawrence Erlbaum.

Clocksin, W. F., & Mellish, C. S. (1981). *Programming in Prolog*. Berlin: Springer-Verlag.

Goldman, N. (1982). *AP3 reference manual*. Unpublished report. Marina del Rey, CA: USC/Information Sciences Institute.

Kahn, K. M. (1978). *Director guide* (AI Memo 482). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. (1965). *LISP 1.5 programmer's manual*. Cambridge, MA: MIT Press.

Minsky, M. (1975). A framework for representing knowledge. In P. H. Winston (Ed.), *The psychology of computer vision*. New York: McGraw-Hill.

Mueller, E. T. (1990). *Daydreaming in humans and machines*. Norwood, NJ: Ablex. [abstract]

Mueller, E. T., & Zernik, U. (1984). *GATE reference manual* (Technical Report UCLA-AI-84-5). Los Angeles: University of California, Artificial Intelligence Laboratory.

Rulifson, J., Derksen, J., & Waldinger, R. (1972). *QA4: A procedural calculus for intuitive reasoning* (Technical Note 73). Stanford, CA: Stanford Research Institute, Artificial Intelligence Center.

Schank, R. C., & Abelson, R. P. (1977). *Scripts, plans, goals, and understanding*. Hillsdale, NJ: Lawrence Erlbaum.

Schank, R. C., & Riesbeck, C. K. (1981). *Inside computer understanding: Five programs plus miniatures*. Hillsdale, NJ: Lawrence Erlbaum.

Wirth, N. (1971). The programming language PASCAL. *Acta Informatica, 1*, 35-63.

Zernik, U., & Dyer, M. G. (1987). The self-extending phrasal lexicon. *Computational Linguistics, 13*, 308-327.

Daydreamer home

(erik@panix.com, www.panix.com/~erik)