$GUMS_1$: A General User Modeling System

Tim Finin
Computer and Information Science
University of Pennsylvania
Philadelphia, PA

David Drager
Arity Corporation
Concord, MA

Abstract

This paper describes a general architecture of a domain independent system for building and maintaining *long term models* of *individual users*. The user modeling system is intended to provide a well defined set of services for an *application system* which is interacting with various users and has a need to build and maintain models of them. As the application system interacts with a user, it can acquire knowledge of him and pass that knowledge on to the user model maintenance system for incorporation. We describe a prototype *general user modeling system* (hereafter called $GUMS_1$) which we have implemented in Prolog. This system satisfies some of the desirable characteristics we discuss.

Introduction - The Need for User Modeling

Systems which attempt to interact with people in an intelligent and cooperative manner need to know many things about the individuals with whom they are interacting. Such knowledge can be of several different varieties and can be represented and used in a number of different ways. Taken collectively, the information that a system has of its users is typically refered to as its *user model*. This is so even when it is distributed through out many components of the system.

Examples that we have been involved with include systems which attempt to provide help and advice [4, 5, 15], tutorial systems [14], and natural language interfaces [16]. Each of these systems has a need to represent information about individual users. Most of the information is acquired incrementaly through direct observation and/or interaction. These systems also needed to infer additional facts about their users based on the directly acquired information. For example, the WIZARD help system [4, 15] had to represent which VMS operating system objects (e.g. commands, command qualifiers, concepts, etc) a user was familiar with and to infer which other objects he was likely to be familiar with.

We are evolving the e design of a general user model maintenance system which would support the modeling needs of the projects mentioned above. The set of services which we envision the model maintenance system performing includes:

- maintaining a data base of observed facts about the user.
- infering additional true facts about the user based on the observed facts.
- infering additional facts which are likely to be true based on default facts and default rules.
- informing the application system when certain facts can be infered to be true or assumed true.
- maintaining the consistency of the model by retracting default information when it is not consistent with the observed facts.
- providing a mechanism for building hierarchies of *stereotypes* which can form initial, partial user models.
- recognizing when a set of observed facts about a user is no longer consistent with a given stereotype and suggesting alternative stereotypes which are consistent.

This paper describes a general architecture for a domain independent system for building and maintaining *long term models* of *individual users*. The user modeling system is intended to provide a well defined set of services for an *application system* which is interacting with various users and has a need to build and maintain models of them. As the application system interacts with a user, it can acquire knowledge of him and pass that knowledge on to the user model maintenance system for incorporation. We describe a prototype *general user modeling system* (hereafter called $GUMS_1$) which we have implemented in Prolog. This system satisfies some of the desirable characteristics we discuss.

What is a User Model?

The concept of encorporating user models into interactive systems has become common, but what has been meant by a user model has varied and is not always clear. In trying to specify what is being refered to as a user model, one has to answer a number of questions: who is being modeled; what aspects of the user are being modeled; how is the model to be initially acquired; how will it be maintained; and how will it be used. In this section we will attempt to characterize our own approach by answering these questions.

Who is being modeled?

The primary distinctions here are whether one is modeling individual users or a class of users and whether one is attempting to construct a short or long term model. We are interested in the aquisition and use of long term models of individual users. We want to represent the knowledge and beliefs of individuals and to do so in a way that results in a persistent record which can grow and change as neccessary.

It will be neccessary, of course,to represent generic facts which are true of large classes (even all) of users. In particular, such facts may include inference rules which relate a person's belief, knowledge or understanding of one thing to his belief, knowledge and understanding of others. For example in the context of a timeshared computer system we may want to include a rule like:

> *If a user U believes that machine M is running, then U will believe that it is possible for him to log onto M.*

It is just this sort of rule which is required in order to support the kinds of cooperative interactions studied in [6] and [7], such as the following:

```
User:   Is UPENN-LINC up?

System: Yes, but you can't log on now.
        Preventative maintenance is being
        done until 11:00am.
```

What is to be modeled?

Our current work is focused on building a general purpose, domain independent model maintenance system. Exactly what information is to be modeled is up to the application. For example, a natural language system may need to know what language terms a user is likely to be familiar with [16], a CAI system for second language learning may need to model a user's knowledge of grammatical rules [14], an intelligent database query system may want to model which fields of a data base relation a user is interested in [10], and an expert system may need to model a user's domain goals [11].

How is the model to be aquired and maintained?

We are exploring a system in which an initial model of the user will be selected from a set of stereotypical user models [13]. Selecting the most appropriate stereotype from the set can be accomplished by a number of techniques, from letting the user select one to surveying the user and having an expert system select one. Once an initial model has been selected, it will be updated and maintained as direct knowledge about the user is aquired from the interaction. Since the use of stereotypical user models is a kind of *default reasoning* [12], we will use *truth maintenance* techniques [9] for maintaining a consistent model.

In particular, if we learn something which contradicts a fact in the our current model of the user than we need to update the model. Updating the model may lead to an inconsistency which must be squared away. If the model can be made consistent by changing any of the *default* facts in the model, then this should be done. If there is a choice of which defaults to alter, then a mechanism must be provided to do this (e.g. through further dialogue with the user). If there are no defaults which can be altered to make the model consistent then the stereotype must be abandoned and a new one sought.

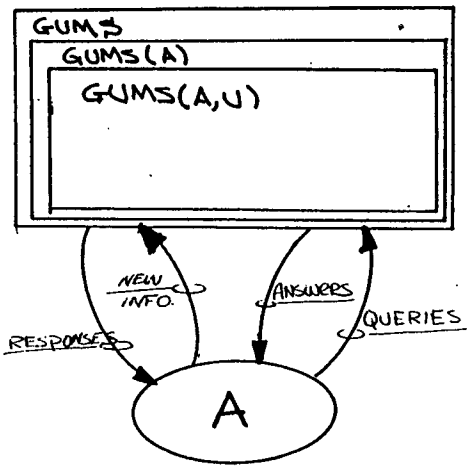
How is the model to be used?

The model can be accessed in two primary ways: facts can be added, deleted or updated from the model and facts can be looked up or infered. A forward chaining component together with a truth maintenance system can be used to update the default assumptions and keep the model consistent.

Architectures for User Modeling Systems

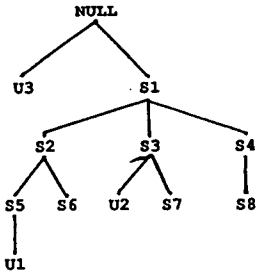
Our goal is to provide a general user modeling utility organized along the lines shown in figures 1 and 2. The user modeling system provides a service to an application program which interacts directly with a user. This application program gathers information about the user through this interaction and choses to store some of this information in the user model. Thus, one service the user model provides is accepting (and storing!) new information about the user. This information may trigger an inferential process which could have a number of outcomes:

- The user modeling system may detect an inconsistency and so inform the application.
- The user model may infer a new fact about the user which triggers a demon causing some action (e.g. informing the application).



A: an Application
GUMS: General User Modeling System
GUMS(A): Modeling System for Application A
GUMS(A,U): Model for User U in Application A

Figure 1: A General Architecture for a User Modeling Utility



NULL: the Empty Stereotype
Si: Stereotype i
Ui: User i

Figure 2: A User Modeling System for an Application

- The user model may need to update some previously infered default information about the user

Another kind of service the user model must provide is answering queries posed by the application. The application may need to look up or deduce certain information about its current user.

We are currently experimenting with some of these ideas in a system called $GUMS_1$. This system is implemented in prolog and used a simple default logic together with a backward chaining interpreter rather than a truth maintenance system and a forward chaining engine. The next section describes $GUMS_1$ and its use of default logic.

Default Logic and User Modeling

A user model is most useful in a situation where the application does not have complete information about the knowledge and beliefs of its users. This leaves us with the problem of how to model a user given we have only a limited amount of knowledge about him. Our approach involves using several forms of default reasoning techniques: stereotypes, explicit default rules, and failure as negation.

We assume that the $GUMS_1$ system will be used in an application which incrementaly gains new knowledge about its users throughout the interaction. But the mere ability to gain new knowledge about the user is not enough. We can not wait until we have full knowledge about a user to reason about him. Fortunately we can very often make generalization about users or classes of users. We call a such a generalization a stereotype. A stereotype consists of a set of facts and rules that are believed to applied to a class of users. Thus a stereotype gives us a form of default reasoning.

Stereotypes can be organized in hierarchies in which one stereotype subsumes another if it can be thought to be more general. A stereotype S_1 is said to be more general than a stereotype S_2 if everything which is true about S_1 is neccessarily true about S_2. Looking at this from another vantage point, a stereotype inherits all the facts and rules from every stereotype that it is subsumed by. For example, in the context of a *programmer's apprentice* application, we might have stereotypes corresponding to different classes of programmer, as is suggested by the the hierarchy in figure 2.

In general, we will want a stereotype to have any number of immediate ancestors, allowing us to compose a new stereotype out of several existing ones. In the context of a *programmers apprentice*, gor example, we may wish to describe a particular user as a *SymbolicsWizard* and a *UnixNovice* and a *ScribeUser*. Thus, the stereotype system should form a general lattice. Our current system constrains the system to a tree.

Within a stereotype we can have default information as well. For instance, we can be sure that a programmer will know what a *file* is, but we can only guess that a programmer will know what a *file directory* is. If we have categorized a given user under the *programmer stereotype* and discover[1] that he is not familiar with the concept of a *file* then we can concludethat we had improperly chosen a stereotype and must choose a new one. But if we got the information that he did not know what a *file directory* was, this would not rule out the possibility of him being a programmer. Thus $GUMS_1$

[1]perhaps through direct interaction with her

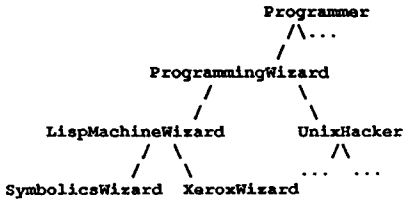


Figure 3: A Hierachy of Stereotypes

allows rules and facts within a stereotype to be either definitely true or true by default (i.e. in the absence of information to the contrary.)

In $GUMS_1$ we use the **certain/1** predicate to introduce a definite fact or rule and the **default/1** predicate to indicate a default fact or rule, as in:

certain(P).	a definite fact: P is true.
certain(P if Q).	a definite rule: P is true if Q is definitely true and P is assumed to be true if Q is only assumed to be true.
default(P).	a default fact: P is assumed to be true unless it is known to be false.
default(P if Q).	a default rule: P is assumed to be true if Q is true or assumed to be true and there is no definite evidence to the contrary.

As an example, consider a situation in which we need to model a persons familiarity with certain terms. This is a common situation in systems which need to produce text as explanations or in response to queries and in which there is a wide variation in the users' familiarity with the domain. We might use the following rules

(a) default(understandsTerm(ram)).

(b) default(understandsTerm(rom)
if understandsTerm(ram)).

(c) certain(understandsTerm(pc)
if understandsTerm(ibmpc)).

(d) certain(~understandsTerm(cpu)).

to represent these assertions, all of which are considered as pertaining to a particular user with respect to the stereotype containing the rules:

(a) Assume the user understands the term *ram* unless we know otherwise.

(b) Assume the user understands the term *rom* if we know or believe he understands the term *ram* unless we know otherwise.

(c) This user understands the term *pc* if he understands the term *ibmpc*.

(d) This user does understand the term *cpu*.

$GUMS_1$ also treats negation as failure in some cases as a default rule. In general, logic is interpreted using an open world assumption. That is, the failure to be able to prove a proposition is not taken as evidence that it is not true. Many logic programming languages, such a prolog, encourage the interpretation of unprovability as logical negation. Two approaches have been forwarded to justify the

negation as failure rule. One approach is the closed world assumption [2]. In this case we assume that anything not inferable from the database is by necessity false. One problem with this assumption is that this is a metalevel assumption and we do not know what the equivalent object level assumptions are. A second approach originated by Clark is based upon the concept of a completed database [1]. A completed database is the database constructed by rewriting the set of clauses defining each predicate to an if and only if definition that is called the completion of the predicate. The purpose of the completed definition is to indicate that the clauses that define a predicate define every possible instance of that predicate.

Any approach to negation as failure requires that a negated goal be ground before execution, (actually a slightly less restrictive rule could allow a partially instantiated negated goal to run but would produce the wrong answer if any variable was bound.) Thus we must have some way of insuring that every negated literal will be bound. In $GUMS_1$ we have used a simple variable typing scheme to achieve this, as will be discussed later.

We have used a variant of the completed database approach to show that a predicate within the scope of a negation is closed. A predicate is closed if and only if it is defined by an iff statement and every other predicate in the definition of this predicate is closed. We allow a metalevel statement *completed(P)* that is used to signify that by predicate P we really intend the iff definition associated with P. This same technique was used by Kowalski [8] to indicate completion. By default we believe *competed(P)* where not indicated. So if *P* is not explicitly closed *not P* is decided by default.

Thus in $GUMS_1$ we have the ability to express that a default should be taken from the lack of certain information (i.e. negation as failure) as well as from the presence of certain information (i.e. default rules). For example, we can have a default rule for the programmer stereotype that can conclude knowledge about linkers from knowledge about compilers, as in:

```
default(knows(linkers) if knows(compilers))
```

We can also have a rule that will take the lack of knowledge about compilers as an indication that the user probably knows about interpreters, as in:

```
certain(knows(interpreters)
        if ~ knows(compilers))
```

This system also allows explicit negative facts and default facts. When negation is proved in reference to a negative fact then negation is not considered a default case. Similarly negation as failure is not considered a default when the predicate being negated is closed. Such distinctions are possible because the $GUMS_1$ interpreter is based on a four value logic.

The distinction between truth or falsity by default (i.e. assumption) and truth or falsity by logical implication is an important one to this system. The central predicate of the system is the two argument predicate *show* which relates a goal *G* expressed as a literal to a truth value. Thus **show(Goal,Val)** returns in the variable *Val* the current belief in the literal *Goal*. The variable *Val* can be instantiated to **true**, **false**, **assume(true)**, or **assume(false)**. The meanings of these values are as follows:

true	definitely true according to the current database.
assume(true)	true by assumption (i.e. true by default)
assume(false)	false by assumption
false	definitely not true.

These values represent truth values for a given user with respect to a given stereotype. If the stereotype is not appropriate, then even definte values may have to change.

Having a four value logic allows us to distinguish conclusions made from purely logical information from those dependent on default information. Four value logic also allows a simple type of introspective reasoning that may be useful for modeling the beliefs of the user. We currently use a default rule to represent an uncertain belief about what the user knows or believes, but we could imagine a situation where we would like to model uncertainties that the user has in his beliefs or knowledge. One such predicate is an embeded *show* predicate. For example we might have a rule that a user will use a operating system command that he believe might erase a file only if he is certain that he knows how to use that command. This might encode as:

```
certain(okay_to_use(Command) if
          can_erase_files(Command),
           show(know(Command),true)).
```

Another predicate *assumed(Pred)* will evaluate the truth of *Pred* and "strengthen" the result. That is

```
demo(assumed(P),V) :-
  demo(P,V2),
  strengthen(V2,V).
```

where the strengthen relation maps assumed values into definite values (e.g. **assume(true)** becomes **true**, **assume(false)** becomes **false** and **true** and **false** remain unchanged). The *assumed* predicate is used to express a certain belief from an uncertain knowledge or belief. For example we might want to express a rule that a user will always want to use a screen editor if he believes one may be available.

```
certain(willUse(screenEditor) if
     assumed(available(screenEditor))).
```

The interpreter that $GUMS_1$ is base on is a metalevel interpreter written in Prolog. The interpreter must generate and compare many possible answers to each subquery, because of the multiple value logic and the presence of explicit negative information. Strong answers to a query (i.e. *true* and *false*) are sought first, followed by weak answers (i.e. *assume(true)* and *assume(false)*). Because strong answers have precedence over weak ones, it is not necessary to remove weak information that contradicts strong information.

Another feature of this system is that we can specify the types of arguments to predicates. This type information can be used to allow the system to handle non-ground goals. In our system, a type provides a way to enumerate a complete set of possible values subsumed by that type. When the top-level *show* predicate is given a partially instantiated goal to solve, it uses the type information to generate a stream of consistent fully instantiated goals. These ground goals are tried sequentially.

That goals must be fully intantiated follows from the fact that negation as failure is built into the evaluation algorithm. Complex terms will be instantiated to every pattern allowed by the datatype given the full power of unification. To specify the type information, one should specify argument types for a predicate, subtype information and type instance information. For example, the following says that the **canProgram** predicate ranges over instances of the type **person** and **programmingLanguage**, that the type **functionalLanguage** is a sub-type of **programmingLanguage** and

that the value **scheme** is an instance of the type **functionalLanguage**:

```
declare(canProgram(person,
                   programmingLanguage)).

subtype(programmingLanguage,
        functionalLanguage).

inst(functionalLanguage,scheme).
```

Limitations of the Present System

Our current system has several limitations. One problem is that it does not extract all of the available information from a new fact learned of the user. If we assert that a predicate is *closed*, we are saying that the set of (certain) rules for the predicate form a definition, i.e. a neccessary and sufficient description. In our current system, however, the information still only flows direction! For example, suppose that we would like to encode the rule that a user knows about *I/O redirection* if and only of they know about *files* and about *pipes*. Further, let's suppose that the default is that a person in this stereotype does not know about *files* or *pipes*. This can be expresses as:

```
certain(knows(io_redirection) if
          knows(pipes),
          knows(files)).

default(~knows(pipes)).

default(~knows(files))

closed(knows(io_redirection)).
```

If we learn that a particular user does know about *I/O redirection* then it should follow that she neccessarily knows about both *files* and *pipes*. Adding the assertion

```
certain(knows(io_redirection))
```

however, will make no additional changes in the data base. The values of **knows(pipes)** and **knows(files)** will not change! A sample run after this change might be :

```
?- show(knows(io_redirection),Val).
        Val = true

?- show(knows(pipes),Val).
        Val = assume(false)

?- show(knows(files),Val).
        Val = assume(false).
```

The reason for this problem is that the current interpreter was designed to be able to incorporate new information without actually using a full truth maintenance system. Before a fact *F* with truth value *V* is to be added to the data base, *GUMS*$_1$ checks to see if an inconsistent truth value *V'* can be derived for *F*. If one can be, then a new stereotype is sought in which the contradiction goes away. New knowledge that does not force an obvious inconsistency within the database is added as is. Neither redundant information or existing default information effect the correctness of the interpreter. Subtler inconsistencies are possible, of course.

Another limitation of the current system its inefficiency. The use of default rules requires us to continue to search for solutions for a goal until a strong one is found or all solutions have been checked. These two limitations may be addressable by redesigning the system to be based on a forward chaining truth maintenance system. The question is whether the relative efficiency of forward chaining will offset the relative inefficiency of truth maintenance. The use of an assumption based truth maintenance system [3] is another alternative that we will investigate.

The *GUMS*$_1$ Command Language

Our current experimental implementation provides the following commands to the application.

show(Query,Val) succeeds with *Val* as the strongest truth value for the gaol *Query*. A Query is a partially or fully instantiated positive or negative literal. Val is return and is the value the current belief state. If *Query* is partially instantiated then it will return more answers upon backtracking if possible. In general one answer will be provided for every legal ground substitution that agrees with current type declarations.

add(Fact,Status) sets belief in *Fact* to true. If *Fact* or any legal instance of it contradicts the current belief state then the user model adopts successively higher stereotypes in the hierarchy until one is found in which all of the added facts are consistent. If no stereotype is successful then no stereotype is used, all answers will be based entirely on added facts. *Fact* must be partially or fully instantiated and can be either a positive or negative literal. *Status* must be uninstantiated and will be bound to a message describing the result of the addition (e.g. one of several error messages, *ok*, the name of a new stereotype, etc.).

create_user(UserName,Stereotype,File,Status) stores the current user if necessary and creates a new user who then is the current user. *UserName* is instantiated to the desired name. *Stereotype* is the logical name of the stereotype that the system should assume to hold. *File* is the name of the file that information pertaining to the user will be stored. *Status* is instantiated by the system and returns error messages. A user must be created in order for the system to be able to answer queries.

store_current(Status) stores the current users information and clears the workspace for a new user. *Status* is instantiated by the system on an error.

restore_user(User,Status) restores a previous user after saving the current user if necessary. *User* is the name of the user. *Status* is instantiated by the system to pass error messages.

done stores the system state of the user modeling system, saving the current user if necessary. This command should be the last command issued and needs to be issued at the end of every session.

Conclusions

Many interactive systems have a strong need to maintain models of individual users. We have presented a simple architecture for a general user modeling utility which is based on the ideas of a default logic. This approach provides a simple system which can maintain a database of known information about users as well as use rules and facts which are associated with a stereotype which is believed to be appropriate for this user. The stereotype can contain definite facts and define rules of inference as well as default information and rules. The rules can be used to derive new information, both definite and assumed, from the currently believed information about the user.

We believe that this kind of system will prove useful to a wide range of applications. We have implemented an initial version in Prolog and are planning to use it to support the modeling needs of several projects. We are also exploring a more powerful approach to user modeling based on the notion of a *truth maintenance system.*

Bibliography

1. Clark, Keith L. Negation as Failure. In *Logic and Databases*, J. Minker and H. Gallaire, Ed., Plenum Press, New York, 1978.

2. Reiter, R. Closed World Databases. In *Logic and Databases*, H. Gallaire & J. Minker, Ed., Plenum Press, 1978, pp. 149-177.

3. DeKleer, J. An Assumption Based Truth Maintenance System. Proceedings of IJCAI-85, IJCAI, August, 1985.

4. Finin, T.W. Help and Advice in Task Oriented Systems. Proc. 7th Int'l. Joint Conf. on Art. Intelligence, IJCAI, August, 1982.

5. Howe, A. and T. Finin. Using Spreading Activation to Identify Relevant Help. Proceeding of the 1984 Canadian Society for Computational Studies of Intelligence, CSCSI, 1984. also available as Technical Report MS-CIS-84-01, Computer and Information Science, U. of Pennsylvania.

6. Joshi, A., Webber, B. & Weischedel, R. Preventing False Inferences. Proceedings of COLING-84, Stanford CA, July, 1984.

7. Joshi, A., Webber, B. & Weischedel, R. Living Up to Expectations: Computing Expert Responses. Proceedings of AAAI-84, Austin TX, August, 1984.

8. Kowalski, Robert. *Logic for Problem Solving.* North-Holland, New York, 1979.

9. McDermott, D and J. Doyle. "Non-Monotonic Logic I". *Artificial Intelligence 13*, 1-2 (1980), 41 - 72.

10. Motro, A. Query Generalization: A Method for Interpreting Null Answers. In Larry Kerschberg, Ed., *Expert Database Systems*, Benjamin/Cummings, Menlo Park CA, 1985.

11. Pollack, M. Information Sought and Information Provided. Proceedings of CHI'85, Assoc. for Computing Machinery (ACM), San Francisco CA, April, 1985, pp. 155-160.

12. Reiter, Ray. "A Logic for Default Reasoning". *Artificial Intelligence 13*, 1 (1980), 81-132.

13. Rich, Elaine. "User Modeling via Stereotypes". *Cognitive Science 3* (1979), 329-354.

14. Schuster, E. and T. Finin. VP2: The Role of User Modelling in Correcting Errors in Second Language Learning. Proc. Conference on Artificial Intelligence and the Simulation of Behavior, AISB, 1985.

15. Shrager, J. and T. Finin. An Expert System that Volunteers Advice. Proc. Second Annual National Conference in Artificial Intelligence, AAAI, August, 1982.

16. Webber, B. and T. Finin. In Response: Next Steps in Natural Language Interaction. In *Artificial Intelligence Applications for Business*, W. Reitman, Ed., Ablex Publ. Co., Norwood NJ, 1984.

Appendix - The Demo Predicate

This appendix defines the *demo* predicate which implements the heart of the *GUMS*$_1$ interpreter. The relation

```
show(Goal,Value)
```

holds if the truth value of proposition *Goal* can be shown to be *Value* for a particular ground instance of *Goal*. The *show* predicate first makes sure that *Goal* is a ground instance via a call to the *bindVars* predicate and then invokes the meta-evaluator *demo*. The relation

```
demo(Goal,Value,Level)
```

requires that *Goal* be a fully instantiated term and *Level* be an integer that represents the level of recursion within the demo predicate. The relation holds if the "strongest" truth value for *Goal* is *Value*.

```
:- op(950,fy,'~').
:- op(1150,xfy,'if').

show(P,V) :- bindVars(P), demo(P,V,0).

% truth values
demo(P,P,_) :- truthValue(P),!.

% reflection...
demo(demo(P,V1),V,D) :-
  !,
  nonvar(V1),
  demo(P,V1,D) -> V=true;V=false.

% disjunction ...
demo((P;Q),V,D) :- !,
  demo(P,V1,D),
  demo(Q,V2,D),
  upperbound(V1,V2,V).

% conjunction ...
demo((P,Q),V,D) :- !,
  demo(P,V1,D),
  demo(Q,V2,D),
  lowerbound(V1,V2,V).

% negation ...
demo(~P,V,D) :- !,
  demo(P,V1,D),
  negate(V1,V,P).

% assumption ...
demo(assumed(P),V,D) :- !,
  demo(P,V1,D),
  strengthen(V1,V).

% call demo1 with deeper depth and then cut.
demo(P,V,Depth) :-
  Deeper is Depth+1,
  demo1(P,V,Deeper),
  retractall(temp(_,Deeper)),
  !.

% definite facts...
demo1(P,true,_) :- certain(P).
demo1(P,false,_) :- certain(~P).

% find a definite rule that yields TRUE or FALSE.
demo1(P,V,D) :-
  forsome(certain(P if Q),(demo(Q,V,D),demoNote(V,D))).

demo1(P,V,D) :-
  forsome(certain(~P if Q),
          (demo(Q,V1,D),
           negate(V1,V,P),
           demoNote(V,D))).

% stop if the best so far was ASSUME(TRUE).
demo1(P,assume(true),D) :-
  retract(temp(assume(true),D)).

% default positive facts.
demo(P,assume(true),_) :- default(P).

% try default rules 'til one gives a positive value.
demo1(P,assume(true),D) :-
  forsome(default(P if Q),(demo(Q,V,D),positive(V))).

% default negative facts.
demo(P,assume(false),_) :- default(~P).

% default negative rules.
```

```
demo1(P,assume(false),D) :-
  forsome(default(~P if Q),(demo(Q,V,D),positive(V))).

% if P is closed, then its false.
demo1(P,false,_) :- closed(P),!.

% the default answer.
demo1(P,assume(false),_).

% demoNote(X,D) succeeds if X is TRUE or FALSE,
% otherwise it fails after updating temp(A,_)
% to be the strongest value known so far.

demoNote(V,_) :- known(V).
demoNote(V,D) :-
  not(temp(_,D)),
  !,
  assert(temp(V,D)),
  fail.
demoNote(assume(true),D) :-
  retract(temp(_,D)),
  !,
  assert(temp(assume(true),D)),
  fail.
```

% Relations on Truth Values

```
positive(X) :- X == true ; X == assume(true).

known(X) :- X == true ; X == false.

higher(true,_).
higher(assume(true),assume(false)).
higher(_,false).

upperbound(X,Y,Z) :- higher(X,Y) -> Z=X ; Z=Y.

lowerbound(X,Y,Z) :- higher(X,Y) -> Z=Y ; Z=X.

strengthen(assume(X),X).
strengthen(true,true).
strengthen(false,false).

% negation is relative to a predicate.
negate(true,false,_).
negate(assume(true),assume(false),_).
negate(assume(false),assume(true),_).
negate(false,true,P) :- closed(P).
negate(false,assume(true),P) :- not(closed(P)).

truthValue(true).
truthValue(false).
truthValue(assume(X)) :- truthValue(X).
```

% The Type System

```
% isSubtype(T1,T2) iff type T1 has an
% ancestor type T2.
isSubtype(T1,T2) :- subtype(T1,T2).
isSubtype(T1,T2) :-
  subtype(T1,T),
  isSubtype(T,T2).

% true if instance I is descendant from type T.
isInstance(I,T) :- inst(I,T).
isInstance(I,T) :-
  isSubtype(T1,T),
  isInstance(I,T1).

% true if T is a type.
isType(T) :- inst(_,T).
isType(T) :- subtype(T,_).
isType(T) :- subtype(_,T).
```

% Grounding Terms

```
% bindVars(P) ensures that all variables
%  in P are bound or it fails.
bindVars(P) :- var(P),!,fail.
bindVars(P) :- atomic(P),!.
bindVars(P) :-
  schema(P,PS),
  P =.. [_|Args],
  PS =.. [_|Types],
  bindArgs(Args,Types).

bindArgs([],[]).
bindArgs([Arg|Args],[Type|Types]) :-
  bindArg(Arg,Type),
  bindArgs(Args,Types).

bindArg(Arg,Type) :-
  var(Arg),
  isInstance(Arg,Type).
bindArg(Arg,_) :- bindVars(Arg).

% scheme(P,S) is true if S is the schema for P, eg
% schema(give(john,X,Y),give(person,person,thing)).

% find a declared schema.
schema(P,S) :-
  functor(P,F,N),
  functor(S,F,N),
  declare(S),
  !.

% use the default schema F(thing,thing,...).
schema(P,S) :-
  functor(P,F,N),
  functor(S,F,N),
  for(I,1,N,arg(I,S,thing)),
  !.
```