Chapter 6: Knowledge Engineering

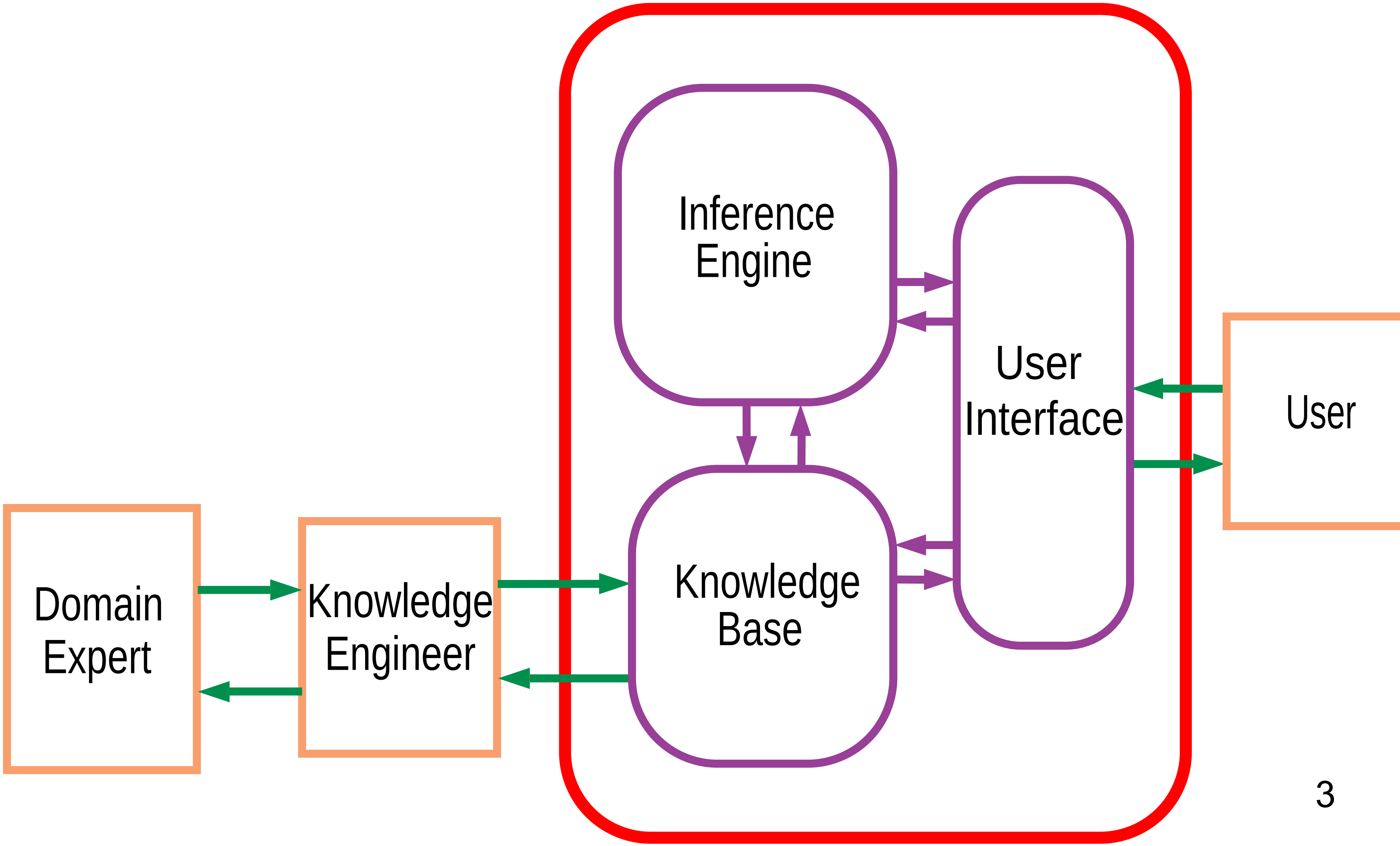
- Lecture 1 Knowledge-based systems, roles of people involved, implementing KBSs: base and metalanguages.
- Lecture 2 Vanilla meta-interpreter, depth-bounded and delaying meta-interpreters.
- Lecture 3 Users. Ask-the-user.
- Lecture 4 Explanation and knowledge-based debugging tools.

Knowledge Engineering

Overview:

- How representation and reasoning systems interact with humans.
- Roles of people involved in a RRS.
- Building RRSs using meta-interpreters.
- Knowledge-based interaction and debugging tools

Knowledge-based system architecture



Roles for people in a KBS

- **Software engineers** build the inference engine and user interface.
- **Knowledge engineers** design, build, and debug the knowledge base in consultation with domain experts.
- **Domain experts** know about the domain, but nothing about particular cases or how the system works.
- **Users** have problems for the system, know about particular cases, but not about how the system works or the domain.

Implementing Knowledge-based Systems

To build an interpreter for a language, we need to distinguish

- **Base language** the language of the RRS being implemented.
- **Metalanguage** the language used to implement the system.

They could even be the same language!

Implementing the base language

Let's use the definite clause language as the base language and the metalanguage.

- We need to represent the base-level constructs in the metalanguage.
- We represent base-level terms, atoms, and bodies as meta-level terms.
- We represent base-level clauses as meta-level facts.
- In the non-ground representation base-level variables are represented as meta-level variables.

Representing the base level constructs

- Base-level atom $p(t_1, \ldots, t_n)$ is represented as the meta-level term $p(t_1, \ldots, t_n)$.
- Meta-level term $oand(e_1, e_2)$ denotes the conjunction of base-level bodies e_1 and e_2.
- Meta-level constant *true* denotes the object-level empty body.
- The meta-level atom $clause(h, b)$ is true if "h if b" is a clause in the base-level knowledge base.

Example representation

The base-level clauses

$$connected\_to(l_1, w_0).$$

$$connected\_to(w_0, w_1) \leftarrow up(s_2).$$

$$lit(L) \leftarrow light(L) \wedge ok(L) \wedge live(L).$$

can be represented as the meta-level facts

$$clause(connected\_to(l_1, w_0), true).$$

$$clause(connected\_to(w_0, w_1), up(s_2)).$$

$$clause(lit(L), oand(light(L), oand(ok(L), live(L)))).$$

Making the representation pretty

- Use the infix function symbol "&" rather than *oand*.
  - instead of writing $oand(e_1, e_2)$, you write $e_1 \,\&\, e_2$.
- Instead of writing $clause(h, b)$ you can write $h \Leftarrow b$, where $\Leftarrow$ is an infix meta-level predicate symbol.
  - Thus the base-level clause "$h \leftarrow a_1 \wedge \cdots \wedge a_n$" is represented as the meta-level atom $h \Leftarrow a_1 \,\&\, \cdots \,\&\, a_n$.

Non-ground Representation

Representing base-level expressions in a metalanguage:

syntactic construct		meta-level representation	
variable	X	variable	X
constant	c	constant	c
function symbol	f	function symbol	f
predicate symbol	p	function symbol	p
"and" operator	^	function symbol	&
"if" operator	<-	predicate symbol	<=
clause	h <- a_1 ^ ... ^ a_n	atom	h <= a_1 & ... & a_n
clause	h.	atom	h <= true.

Example representation

The base-level clauses

$$connected\_to(l_1, w_0).$$

$$connected\_to(w_0, w_1) \leftarrow up(s_2).$$

$$lit(L) \leftarrow light(L) \land ok(L) \land live(L).$$

can be represented as the meta-level facts

$$connected\_to(l_1, w_0) \Leftarrow true.$$

$$connected\_to(w_0, w_1) \Leftarrow up(s_2).$$

$$lit(L) \Leftarrow light(L) \,\&\, ok(L) \,\&\, live(L).$$

Vanilla Meta-interpreter

$prove(G)$ is true when base-level body G is a logical consequence of the base-level KB.

$$prove(true).$$

$$prove((A \mathbin{\&} B)) \leftarrow$$
$$\quad prove(A) \land$$
$$\quad prove(B).$$

$$prove(H) \leftarrow$$
$$\quad (H \Leftarrow B) \land$$
$$\quad prove(B).$$

Example base-level KB

$$live(W) \Leftarrow$$
$$\quad connected\_to(W, W_1) \;\&$$
$$\quad live(W_1).$$
$$live(outside) \Leftarrow true.$$
$$connected\_to(w_6, w_5) \Leftarrow ok(cb_2).$$
$$connected\_to(w_5, outside) \Leftarrow true.$$
$$ok(cb_2) \Leftarrow true.$$
$$?prove(live(w_6)).$$

Expanding the base-level

Adding clauses increases what can be proved.

- **Disjunction** Let $a; b$ be the base-level representation for the disjunction of a and b. Body $a; b$ is true when a is true, or b is true, or both a and b are true.

- **Built-in predicates** You can add built-in predicates such as N is E that is true if expression E evaluates to number N.

Expanded meta-interpreter

$$prove(true).$$

$$prove((A \,\&\, B)) \leftarrow$$

$$\quad prove(A) \wedge prove(B).$$

$$prove((A; B)) \leftarrow prove(A).$$

$$prove((A; B)) \leftarrow prove(B).$$

$$prove((N \text{ is } E)) \leftarrow$$

$$\quad N \text{ is } E.$$

$$prove(H) \leftarrow$$

$$\quad (H \Leftarrow B) \wedge prove(B).$$

Depth-Bounded Search

- Adding conditions reduces what can be proved.

% $bprove(G, D)$ is true if G can be proved with a proof tree

% of depth less than or equal to number D.

$$bprove(true, D).$$

$$bprove((A \,\&\, B), D) \leftarrow$$

$$bprove(A, D) \wedge bprove(B, D).$$

$$bprove(H, D) \leftarrow$$

$$D \geq 0 \wedge D_1 \text{ is } D - 1 \wedge$$

$$(H \Leftarrow B) \wedge bprove(B, D_1).$$

Delaying Goals

Some goals, rather than being proved, can be collected in a list.

- To delay subgoals with variables, in the hope that subsequent calls will ground the variables.
- To delay assumptions, so that you can collect assumptions that are needed to prove a goal.
- To create new rules that leave out intermediate steps.
- To reduce a set of goals to primitive predicates.

Delaying Meta-interpreter

% $dprove(G, D_0, D_1)$ is true if D_0 is an ending of list of
% delayable atoms D_1 and $KB \wedge (D_1 - D_0) \models G$.

$$dprove(true, D, D).$$

$$dprove((A \,\&\, B), D_1, D_3) \leftarrow$$
$$\quad dprove(A, D_1, D_2) \wedge dprove(B, D_2, D_3).$$

$$dprove(G, D, [G|D]) \leftarrow delay(G).$$

$$dprove(H, D_1, D_2) \leftarrow$$
$$\quad (H \Leftarrow B) \wedge dprove(B, D_1, D_2).$$

Example base-level KB

$$live(W) \Leftarrow$$

$$connected\_to(W, W_1) \ \&$$

$$live(W_1).$$

$$live(outside) \Leftarrow true.$$

$$connected\_to(w_6, w_5) \Leftarrow ok(cb_2).$$

$$connected\_to(w_5, outside) \Leftarrow ok(outside\_connection).$$

$$delay(ok(X)).$$

$$?dprove(live(w_6), [\,], D).$$

Trace of dprove example

Each forward step is indicated as a box:

?<goal of proof step>	<parent box#> <box#>
<matching clause of knowledge base before unification>	
<matching clause of knowledge base after unification>	

The box colors indicate:

- fail
- success
- subgoal calls

Subgoal successes are fed back to the parent box:

<proved goal>	<box#> -> <parent box#>

Proof steps using dprove (1)

?dprove(live(w6), [ ], D) 0 1
dprove(G, D, [G| D]) <- delay(G)
dprove(live(w6), [ ], [live(w(6)]) <- delay(live(w6))

?dprove(live(w6), [ ], D) 1 2
dprove(H, D1, D2) <- (H <= B) ∧dprove(B, D1, D2)
dprove(live(w6), [ ], D) <- (live(w6) <= B) ∧dprove(B, [ ], D)

?(live(w6) <= B) 2 3
live(W) <= connected_to(W, W1) & live(W1)
live(w6) <= connected_to(w6, W1) & live(W1)

?dprove(connected_to(w6, W1) & live(W1), [ ], D) 2 4
dprove((A & B), D4, D6) <- dprove(A, D4, D5) ∧dprove(B, D5, D6)
dprove((connected_to(w6, W1) & live(W1)), [ ], D) <-
dprove(connected_to(w6, W1), [ ], D5) ∧dprove(live(W1), D5, D)

Proof steps using dprove (2)

?dprove(connected_to(w6, W1), [ ], D5) 4 5
dprove(G, D, [G| D]) <- delay(G)
dprove(connected_to(w6, W1), [ ], [connected_to(w6, W1)]) <-
delay(connected_to(w6, W1))

?dprove(connected_to(w6, W1), [ ], D5) 4 6
dprove(H, D7, D8) <- (H <= B) ∧dprove(B, D7, D8)
dprove(connected_to(w6, W1), [ ], D5) <- (connected_to(w6, W1) <= B)
∧dprove(B, [ ], D5)

?(connected_to(w6, W1) <= B) 6 7
connected_to(w6, w5) <= ok(cb2)
connected_to(w6, w5) <= ok(cb2)

?dprove(ok(cb2), [ ], D5) 6 8
dprove(G, D9, [G| D9]) <- delay(G)
dprove(ok(cb2), [ ], [ok(cb2)]) <- delay(ok(cb2))

Proof steps using dprove (3)

?delay(ok(cb2)) 8 9
delay(ok(X))
delay(ok(cb2))
dprove(ok(cb2), [ ], [ok(cb2)]) <- true
dprove(connected_to(w6, w5), [ ], [ok(cb2)]) <- true

dprove(ok(cb2), [ ], [ok(cb2)]) <- true 9 -> 8

dprove(connected_to(w6, w5), [ ], [ok(cb2)]) <- true 8 -> 6

?dprove(live(w5), [ok(cb2)], D) 4 10
dprove(G, D, [G| D]) <- delay(G)
dprove(live(w5), [ok(cb2)], [live(w5)| [ok(cb2)]]) <- delay(live(w5))

?dprove(live(w5), [ok(cb2)], D) 4 11
dprove(H, D11, D12) <- (H <= B) ∧dprove(B, D11, D12)
dprove(live(w5), [ok(cb2)], D) <- (live(w5) <= B) ∧dprove(B, [ok(cb2)], D)

Proof steps using dprove (4)

? (live(w5) <= B) 11 12
live(W2) <= connected_to(W2, W3) & live(W3)
live(w5) <= connected_to(w5, W3) & live(W3)

?dprove((connected_to(w5, W3) & live(W3)), [ok(cb2)],) 4 13
dprove((A & B), D13, D15) <- dprove(A, D13, D14) ∧dprove(B, D14, D15)
dprove((connected_to(w5, W3) & live(W3)), [ok(cb2)], D) <-
dprove(connected_to(w5, W3), [ok(cb2)], D14) ∧dprove(live(W3), D14, D)

?dprove(connected_to(w5, W3), [ok(cb2)], D14) 13 14
dprove(G, D16, [G| D16]) <- delay(G)
dprove(connected_to(w5, W3), [ok(cb2)], [connected_to(w5, W3)| [ok(cb2)]]) <-
delay(connected_to(w5, W3))

?dprove(connected_to(w5, W3), [ok(cb2)], D14) 13 15
dprove(H, D17, D18) <- (H <= B) ∧dprove(B, D17, D18)
dprove(connected_to(w5, W3), [ok(cb2)], D14) <- (connected_to(w5, W3) <= B)
∧dprove(B, [ok(cb2)], D14)

Proof steps using dprove (5)

? (connected_to(w5, W3) <= B) 15 16
connected_to(w5, outside) <= ok(outside_connection)
connected_to(w5, outside) <= ok(outside_connection)

?dprove(ok(outside_connection), [ok(cb2)], D14) 15 17
dprove(G, D19, [G| D19]) <- delay(G)
dprove(ok(outside_connection), [ok(cb2)], [ok(outside_connection)| [ok(cb2)]])
<- delay(ok(outside_connection))

?delay(ok(outside_connection)) 17 18
delay(ok(X))
delay(ok(outside_connection))

dprove(ok(outside_connection), [ok(cb2)], [ok(outside_connection)| [ok(cb2)]])
<- true 18 -> 17

Proof steps using dprove (6)

dprove(connected_to(w5, outside), [ok(cb2)], [ok(outside_connection), ok(cb2)]) <-
true 17 -> 15

?dprove(live(outside), [ok(outside_connection), ok(cb2)], D) 13 19
dprove(G, D20, [G| D20]) <- delay(G)
dprove(live(outside), [ok(outside_connection), ok(cb2)], [live(outside)| D20]) <-
delay(live(outside))

?dprove(live(outside), [ok(outside_connection), ok(cb2)], D) 13 20
dprove(H, D21, D22) <- (H <= B) ∧dprove(B, D21, D22)
dprove(live(outside), [ok(outside_connection), ok(cb2)], D2) <- (live(outside) <= B)
∧dprove(B, [ok(outside_connection), ok(cb2)], D)

? (live(outside) <= B) 20 21
live(outside) <= true
live(outside) <= true

Proof steps using dprove (7)

?dprove(true, [ok(outside_connection), ok(cb2)], D) 20 22
dprove(true, D23, D23)
dprove(true, [ok(outside_connection), ok(cb2)], [ok(outside_connection), ok(cb2)])

dprove((connected_to(w5, outside) & live(outside)), [ok(cb2)],
[ok(outside_connection), ok(cb2)]) <- true 22 -> 20

dprove(live(outside), [ok(outside_connection), ok(cb2)], [ok(outside_connection),
ok(cb2)]) <- true 20 -> 13

dprove(live(w5), [ok(cb2)], [ok(outside_connection), ok(cb2)]) <- true 13 -> 11

dprove((connected_to(w6, w5) & live(w5)), [ ], [ok(outside_connection), ok(cb2)])
<- true 11 -> 4

dprove(live(w6), [ ], [ok(outside_connection), ok(cb2)]) <- true 4 -> 2

dprove(live(w6), [ ], [ok(outside_connection), ok(cb2)]) <- true 2 -> 0

Users

How can users provide knowledge when

- they don’t know the internals of the system
- they aren’t experts in the domain
- they don’t know what information is relevant
- they don’t know the syntax of the system
- but they have essential information about the particular case of interest?

Querying the User

- The system can determine what information is relevant and ask the user for the particular information.
- A top-down derivation can determine what information is relevant. There are three types of goals:
  - Goals for which the user isn't expected to know the answer, so the system never asks.
  - Goals for which the user should know the answer, and for which they have not already provided an answer.
  - Goals for which the user has already provided an answer.

Yes/No questions

- The simplest form of a question is a ground query.
- Ground queries require an answer of “yes” or “no”.
- The user is only asked a question if
  - the question is askable, and
  - the user hasn’t previously answered the question.
- When the user has answered a question, the answer needs to be recorded.

Ask-the-user meta-interpreter

% $aprove(G)$ is true if G is a logical consequence of the
% base-level KB and yes/no answers provided by the user.

$$aprove(true).$$

$$aprove((A \,\&\, B)) \leftarrow aprove(A) \wedge aprove(B).$$

$$aprove(H) \leftarrow askable(H) \wedge answered(H, yes).$$

$$aprove(H) \leftarrow$$

$$askable(H) \wedge unanswered(H) \wedge ask(H, Ans) \wedge$$

$$record(answered(H, Ans)) \wedge Ans = yes.$$

$$aprove(H) \leftarrow (H \Leftarrow B) \wedge aprove(B).$$

Functional Relations

- You probably don't want to ask ?$age(fred, 0)$, ?$age(fred, 1)$, ?$age(fred, 2)$, . . .
- You probably want to ask for Fred's age once, and succeed for queries for that age and fail for other queries.
- This exploits the fact that *age* is a functional relation.
- Relation $r(X, Y)$ is functional if, for every X there exists a unique Y such that $r(X, Y)$ is true.

Getting information from a user

- The user may not know the vocabulary that is expected by the knowledge engineer.
- Either:
  - The system designer provides a menu of items from which the user has to select the best fit.
  - The user can provide free-form answers. The system needs a large dictionary to map the responses into the internal forms expected by the system.

More General Questions

Example: For the subgoal $p(a, X, f(Z))$ the user can be asked:

for which X, Z is $p(a, X, f(Z))$ true?

- Should users be expected to give all instances which are true, or should they give the instances one at a time, with the system prompting for new instances?

Example: For which S, C is $enrolled(S, C)$ true?

- Psychological issues are important.

Reasking Questions

When should the system repeat or not ask a question?

Example:

Query	Ask?	Response
$?p(X)$	yes	$p(f(Z))$
$?p(f(c))$	no	
$?p(a)$	yes	yes
$?p(X)$	yes	no
$?p(c)$	no	

Don't ask a question that is more specific than a query to which either a positive answer has already been given or the user has replied *no*.

Delaying Asking the User

- Should the system ask the question as soon as it's encountered, or should it delay the goal until more variables are bound?

- Example consider query $?p(X)$ & $q(X)$, where $p(X)$ is askable.
  - If $p(X)$ succeeds for many instances of X and $q(X)$ succeeds for few (or no) instances of X it's better to delay asking $p(X)$.
  - If $p(X)$ succeeds for few instances of X and $q(X)$ succeeds for many instances of X, don't delay.

Explanation

- The system must be able to justify that its answer is correct, particularly when it is giving advice to a human.
- The same features can be used for explanation and for debugging the knowledge base.
- There are three main mechanisms:
  - Ask HOW a goal was derived.
  - Ask WHYNOT a goal wasn't derived.
  - Ask WHY a subgoal is being proved.

How did the system prove a goal?

- If g is derived, there must be a rule instance

  $$g \Leftarrow a_1 \ \& \ \ldots \ \& \ a_k.$$

  where each a_i is derived.

- If the user asks HOW g was derived, the system can display this rule. The user can then ask

  HOW i.

  to give the rule that was used to prove a_i.

- The HOW command moves down the proof tree.

Meta-interpreter that builds a proof tree

% $hprove(G, T)$ is true if G can be proved from the base-level

% KB, with proof tree T.

$$hprove(true, true).$$

$$hprove((A \mathbin{\&} B), (L \mathbin{\&} R)) \leftarrow$$

$$hprove(A, L) \wedge$$

$$hprove(B, R).$$

$$hprove(H, if(H, T)) \leftarrow$$

$$(H \Leftarrow B) \wedge$$

$$hprove(B, T).$$

Why Did the System Ask a Question?

It is useful to find out why a question was asked.

- Knowing why a question was asked will increase the user's confidence that the system is working sensibly.
- It helps the knowledge engineer optimize questions asked of the user.
- An irrelevant question can be a symptom of a deeper problem.
- The user may learn something from the system by knowing why the system is doing something.

WHY question

- When the system asks the user a question g, the user can reply with

  WHY

- This gives the instance of the rule

  $$h \Leftarrow \cdots \,\&\, g \,\&\, \cdots$$

  that is being tried to prove h.

- When the user asks WHY again, it explains why h was proved.

Meta-interpreter to collect rules for WHY

% $wprove(G, A)$ is true if G follows from base-level KB, and

% A is a list of ancestor rules for G.

$$
\begin{array}{l}
wprove(true, Anc). \\
wprove((A \,\&\, B), Anc) \leftarrow \\
\qquad wprove(A, Anc) \wedge \\
\qquad wprove(B, Anc). \\
wprove(H, Anc) \leftarrow \\
\qquad (H \Leftarrow B) \wedge \\
\qquad wprove(B, [(H \Leftarrow B)|Anc]).
\end{array}
$$

Debugging Knowledge Bases

There are four types of nonsyntactic errors that can arise in rule-based systems:

- An incorrect answer is produced; that is, some atom that is false in the intended interpretation was derived.
- Some answer wasn't produced; that is, the proof failed when it should have succeeded, or some particular true atom wasn't derived.
- The program gets into an infinite loop.
- The system asks irrelevant questions.

Debugging Incorrect Answers

- An incorrect answer is a derived answer which is false in the intended interpretation.
- An incorrect answer means a clause in the KB is false in the intended interpretation.
- If g is false in the intended interpretation, there is a proof for g using $g \Leftarrow a_1 \,\&\, \ldots \,\&\, a_k$. Either:
  - Some a_i is false: debug it.
  - All a_i are true. This rule is buggy.

Debugging Missing Answers

- WHYNOT *g*. *g* fails when it should have succeeded.

  Either:

  - There is an atom in a rule that succeeded with the wrong answer, use HOW to debug it.
  - There is an atom in a body that failed when it should have succeeded, debug it using WHYNOT.
  - There is a rule missing for *g*.

Debugging Infinite Loops

- There is no automatic way to debug all such errors: halting problem.
- There are many errors that can be detected:
  - If a subgoal is identical to an ancestor in the proof tree, the program is looping.
  - Define a well-founded ordering that is reduced each time through a loop.