

AILog User Manual

Version 2.3

David Poole
Department of Computer Science,
University of British Columbia,
Vancouver, B.C. Canada V6T 1Z4
Email: poole@cs.ubc.ca
URL: <http://www.cs.ubc.ca/spider/poole>

December 1, 2008

Abstract

This manual describes AILog, (formerly CILog), a simple representation and reasoning system based on the books *Artificial Intelligence: foundations of computational agents* [Poole and Mackworth, 2009] and *Computational Intelligence: A Logical Approach* Poole, Mackworth, and Goebel [1998], and the Independent Choice Logic [Poole, 2008] for the probabilistic reasoning. AILog provides:

- a definite clause representation and reasoning system
- a simple tell-ask user interface, where the user can tell the system facts and ask questions of the system
- explanation facilities to explain how a goal was proved, why an answer couldn't be found, why a question was asked, why an error-producing goal was called, and why the depth-bound was reached
- knowledge-level debugging tools, that let the user debug incorrect answers, missing answers, why the system asks a question, system errors, and possible infinite loops
- depth-bounded search, that can be used to investigate potential infinite loops and used to build an iterative-deepening search procedure
- sound negation-as-failure, that interacts appropriately with the depth-bound
- ask-the-user facilities
- assumables for finding conflicts (e.g., in consistency based diagnosis) and for abduction
- probabilistic reasoning, integrated with negation as failure.

Although it has similarities with Prolog, it should not be seen as “programming in logic”, but as “representing in logic”. The tools are designed for the case where you are trying to model a domain in logic and can ask the system questions about what follows. It can explain its reasoning in terms of the domain.

AILog is a purely declarative representation and reasoning system. It is intended as a pedagogical tool to present a simple logic that can be used for AI problems. It is meant for programming in the small, where

you can axiomatize a domain, ask questions and debug the knowledge base, without knowing how answers are produced. There may be sophisticated problem-solving techniques used in finding answers¹.

We have many examples of probabilistic reasoning in AILog2, as well as of the examples from the new book available. The examples from *cilog* should also work.

1 Getting and Starting AILog

To start AILog just load the file *ailog2.pl* into Prolog.

Currently we have:

- *ailog2.pl* is currently at version 2.3. This runs on SWI Prolog version 5.3 or later.
- Ailog version 1 is an older version that does not include probabilities, but has versions that run in older versions of SWI Prolog and in Sicstus Prolog. Try that if you have problems with this version.
- *ailog21.pl* is Ailog version 2.1, which finds all explanations and so does not work for large examples.

2 Help and Exiting

The basic system prompt is:

```
ailog:
```

This means that AILog is waiting for input from the user. All user inputs end with a period (“.”) and a return (enter).

At any stage you can ask for help by issuing a help command:

```
ailog: help.
```

This gives a brief description of the available commands.

To quit AILog, you can issue the command:

```
ailog: quit.
```

AILog is written in Prolog. To exit AILog to the underlying Prolog, you can use the command: .

```
ailog: prolog.
```

To restart AILog, you should use the command:

```
| ?- start.
```

at the prolog prompt. Note that this does not clear the knowledge base.

To turn on reporting the run time of queries, you can do

```
ailog: stats runtime.
```

On some systems, this is the elapsed time since the query was asked or the user asked for more answers, which may be a very inaccurate of run time if there was user interaction. To turn off reporting of run time, you can do:

```
ailog: stats none.
```

¹There weren't any when this manual was written, but one of the design goals is that we should be able to use, for example, constraint satisfaction techniques to solve goals, without needing to change the user-level view of the system. It may just run faster (or not, as the case may be).

3 Syntax

Atoms start with lower-case letters and can include letters, numbers and underscore (“_”)

A **body** is either an atom, of the form $\alpha\&\beta$, where α and β are bodies, or of the form $\sim\alpha$, where α is a body.

A **clause** is either an atom or is a rule of the form $h<-b$ where h is an atom (the head of the clause) and b is a body.

3.1 Individuals and Relations

Ailog can be used for propositional reasoning, but can also be used for relational reasoning.

The syntax of variables and predicates is based on Prolog’s syntax in its convention for variables, but uses a different syntax for operators (because Prolog’s are so confusing and to emphasize Ailog is not Prolog).

A **variable** is a sequence of alphanumeric characters (possibly including “_”) that starts with an upper case letter or “_”. For example, X, Letter, Any_cat, A_big_dog are all variables. The variable “_” in an anonymous variable which means that all occurrences are assumed to be different variables. If a variable occurs only once in a clause, it should probably be written as “_”.

A **constant** is either:

- a sequence of alphanumeric characters (possibly including “_”) starting with a lower case letter, such as: david, comp_intell, ailog, and a45_23
- an integer or float, such as 123, -5, 1.0, -3.14159, 4.5E7, -0.12e+8, and 12.0e-9. There must be a decimal point in floats written with an exponent and at least one digit before and after a decimal point.
- any sequence of characters delimited by single quotes, such as 'X', '2b~2b', './ch2/foo.pl', 'A Tall Person'

A **term** is either a variable, a constant, or of the form $f(t_1, \dots, t_n)$, where f , a function symbol, is a sequence of alphanumeric characters (possibly including “_”) starting with a lower case letter and the t_i are terms.

An **atom** is either of the form p or $p(t_1, \dots, t_n)$, where p , a predicate symbol, is a sequence of alphanumeric characters (including _) starting with a lower case letter and the t_i are terms. Given this definition for atoms, the rest of the syntax is the same as the propositional case. Some predicate and function symbols can be written using infix notation. For example “X is 4+3*Y” means the same as “is(X, +(4, *(3, Y)))”, where “is” is a predicate symbol and “+” and “*” are function symbols. The operator precedence follows Prolog’s conventions.

The symbol “<=” is defined to be infix, but there are no clauses defining it. This is designed to be used for meta-programming where “<=” can be used as a meta-level predicate symbol defining the object-level implication. (See *Artificial Intelligence* Poole and Mackworth [2009], Chapter 13).

4 Ask and Tell

The general interaction with AILog is to **tell** it clauses and to **ask** it queries (which are bodies). It replies with answers, which are instances of the query that are consequences of its knowledge base. The user can

then either explore how the system derived an answer or why the system didn't find another answer.

To add a clause to the knowledge base you issue the command:

```
ailog: tell clause.
```

Example 1 You could tell the system some genealogy knowledge:

```
ailog: tell parent (X,Y) <- mother (X,Y) .
ailog: tell parent (X,Y) <- father (X,Y) .
ailog: tell grandfather (X,Y) <- father (X,Z) & parent (Z,Y) .
ailog: tell grandmother (X,Y) <- mother (X,Z) & parent (Z,Y) .
ailog: tell father (randy, sally) .
ailog: tell father (randy, joe) .
ailog: tell mother (sally, mary) .
ailog: tell father (joe, sue) .
```

You can ask a query to determine if some body is a consequence of the knowledge base:

```
ailog: ask query.
```

where *query* is a body. The system will respond with either an instance of the query that is a consequence of the knowledge base or "No. *query* doesn't follow from the knowledge base" if there are no answers. When the system have given an answer you can reply with:

ok. to indicate that you don't want any more answers

more. to ask for more answers

how. to investigate how that answer was produced (see Section 6.1)

help. to get a menu of available answers (this option is available at all points where the system is asking for a response)

If you reply with **more**, the system either finds another answer or reports there are no more answers.

Example 2 Given the knowledge base after Example 1, you can ask queries of the knowledge base:

```
ailog: ask grandfather (randy, mary) .
Answer: grandfather (randy, mary) .
[ok, more, how, help]: ok.
```

It has thus told you that *grandfather(randy,mary)* is a consequence of the knowledge base.

```
ailog: ask grandfather (joe, mary) .
No. grandfather (joe, mary) doesn't follow from the knowledge base.
```

which means *grandfather(joe,mary)* is not a consequence of the knowledge base.

You can ask queries with free variables:

```

ailog: ask grandfather(G,C) .
Answer: grandfather(randy,mary) .
      [ok,more,how,help]: more.
Answer: grandfather(randy,sue) .
      [ok,more,how,help]: more.
No more answers.

```

This means there are two instances of the query *grandfather(G,C)* that are consequences of the knowledge base.

5 Loading and Listing Knowledge Bases

You can also tell the system a set of clauses from a file. The command:

```
ailog: load 'filename' .
```

where *'filename'* is the name (enclosed in single quotes) of a file that contains a sequence of clauses (each clause ending with a period), adds the clauses to the knowledge base as though they had been told to the system. Note that the file does *not* contain **tell** commands; it just contains the clauses.

Within files (and at any prompt), **comments** can be included between a “%” and the end of a line, or between “/*” and “*/”.

Loading a file adds the clauses to the database. It does not replace definitions. To clear the knowledge base you can issue the command:

```
ailog: clear.
```

To remove all clauses with head *atom* from the knowledge base, you issue the command:

```
ailog: clear atom.
```

To list the contents of the knowledge base you can issue the command:

```
ailog: listing.
```

This gives a listing of the whole knowledge base that is suitable for copying to a file and loading in a subsequent AILog session.

To get a listing of all clauses in the knowledge base whose head unifies with *atom*, you can issue the command:

```
ailog: listing atom.
```

Note that in listings, and any time atoms with free variables are written, the variables are renamed. The system does not remember the names you gave the variables. It just calls them *A*, *B*, *C*, etc.

Example 3 Suppose the file “genealogy.cil” contains the rules from Example 1. The following shows how the file can be loaded and listed.

```

ailog: load 'genealogy.cil' .
Ailog theory genealogy.cil loaded.
ailog: listing.
parent(A,B) <- mother(A,B) .
parent(A,B) <- father(A,B) .
grandfather(A,B) <- father(A,C) & parent(C,B) .
grandmother(A,B) <- mother(A,C) & parent(C,B) .
father(randy,sally) .
father(randy,joe) .
mother(sally,mary) .
father(joe,sue) .
ailog:

```

To load a file that uses Prolog syntax, you can issue the command:

```
ailog: prload 'filename' .
```

This assumes the clauses in the file use the Prolog implication `:-` and use a comma for conjunction. This does not allow the use of Prolog extra-logical predicates.

One useful command to check your knowledge base is the command:

```
ailog: check .
```

This lists the rules in the knowledge base with atoms in the body which don't unify with the head of any clause in the knowledge base (or any askable or assumable). This is a simple check that the rule can never be used in a proof. This is useful as this is usually an indication that there is a problem with the knowledge base.

This static check replaces the dynamic “undefined predicate” exception of Prolog, but is more useful in that it can be done before asking any queries, it finds clauses that contain atoms that must immediately fail (even if there are other clauses with the same predicate symbol). The warning can be ignored if that is what you intended (for example if you intend to add appropriate clauses later).

6 Explanation and Debugging

The main power of AILog is in the explanation and debugging facilities. These fall into three classes:

- determining how an answer was proved
- determining why a potential answer wasn't produced
- determining why the search is in a particular state

Each of these is described in the following sections.

6.1 How? Questions

When the system has derived an answer you can ask **how** that answer was produced. This provides a mechanism for interactively traversing a proof tree. At each stage the system either says why the answer was produced immediately (e.g., if it was a fact or a built-in predicate — see Section 8) or produces the instance of the top-level rule that was used to prove the goal.

When you ask how atom h was proved, it produces the instance of the rule in the knowledge base with h as the head that succeeded:

```
h <-
  1: a1
  2: a2
  ...
  k: ak
```

which indicates that the rule $h \leftarrow a_1 \& a_2 \& \dots \& a_k$ was used to prove h . You can then give one of:

how i . where i is an integer $1 \leq i \leq k$. This means that you want to see how a_i was proved.

up. to go back to see the rule with h in the body. If h is the initial query it goes back to the answer interaction.

retry. to ask for a different proof tree.

ok. to exit the how traversal and go back to the answer prompt.

help. for a list of the available commands.

If a_i was not proved via a rule (for example, if it was a fact or a built-in predicate), the reason that a_i was proved is printed and then the rule with a_i in the body is printed again.

Example 4 Given the knowledge base after Example 1, you can ask how a particular instance of the query was proved, as in:

```
ailog: ask grandfather(G,C).
Answer: grandfather(randy,mary).
[ok,more,how,help]: how.
grandfather(randy,mary) <-
  1: father(randy,sally)
  2: parent(sally,mary)
How? [number,up,retry,ok,help]: how 2.
parent(sally,mary) <-
  1: mother(sally,mary)
How? [number,up,retry,ok,help]: how 1.
mother(sally,mary) is a fact
parent(sally,mary) <-
  1: mother(sally,mary)
How? [number,up,retry,ok,help]: up.
grandfather(randy,mary) <-
  1: father(randy,sally)
```

```

    2: parent(sally,mary)
How? [number,up,retry,ok,help]: how 1.
father(randy,sally) is a fact
grandfather(randy,mary) <-
    1: father(randy,sally)
    2: parent(sally,mary)
How? [number,up,retry,ok,help]: up.
Answer: grandfather(randy,mary).
[ok,more,how,help]: more.
Answer: grandfather(randy,sue).
[ok,more,how,help]: ok.
ailog:

```

6.2 Whynot? Questions

Just as how questions let you explore the proof tree for a particular derivation, **whynot** questions let you explore the search tree. This facility is mainly used for determining why there was no proof for a particular query. As such the documentation is written assuming that you are trying to determine why a query failed, when you thought that it should have succeeded.

If a query fails when it should have succeeded, either there was a rule defining the query whose body fails when it should have succeeded, or else there is a missing rule for that query.

You can ask a query to determine why some query failed using:

```
ailog: whynot query.
```

where *query* is a body.

If the query is an atom, you can examine each rule whose head unifies with the query. For each such rule, the system asks whether you want to trace this rule. You can reply with:

yes. to determine why this rule failed. You should give this answer when this rule should have succeeded for this query.

no. to ask for another rule. You give this answer if this rule should have failed for this query.

up. to return to a previous choice point (the rule in which the atom appears, or else the top-level if the atom was the initial query).

ok. to return to the top-level.

help. to get a menu of available answers.

If you answer “**no**” to each rule, this means that all of the rules in the knowledge base should have failed, and so the appropriate rule for the query that should have succeeded is missing.

To determine why a rule failed, we determine why the body failed. If the body is atomic, we use the above *whynot* mechanism to determine why the rule failed. If the body of the rule is a conjunction, $\alpha \& \beta$, there are four cases:

- α fails, in which case we should use recursively use the *whynot* mechanism to determine why it failed.

- an instance of α succeeds, but should not have succeeded. In this case we can use the *how* mechanism to debug this proof. This case arises because β may have rightfully failed on the instance of α that succeeded.
- an instance of α succeeds, but this instance should have lead to failure of the conjunction. In this case we should look for another proof for α .
- an instance of α succeeds that should lead to the success of the conjunction, in which case we need to determine why β failed on this instance.

Thus, when there is a conjunctive body, we first try to prove the leftmost conjunct. If it fails, we use the above *whynot* mechanism to determine why it failed. If it succeeds, the user is asked *Should this answer lead to a successful proof?* The user can reply:

yes. this instance should have made the body succeed. Thus you need to debug the rest of the conjunction.

no. this instance should lead to a failure of the body. Thus you need to try another proof for this atom.

debug. this instance is false, debug it. This enters the *how* interaction.

ok. to return to the top-level.

help. to get a menu of available answers.

Example 5 Suppose we had the knowledge base from Example 1. Suppose that, the user knew that Joe had a child called Jane, and wanted to know why the system didn't think that Randy was Jane's grandfather. We could imagine the following dialogue:

```
ailog: whynot grandfather(randy, jane) .
grandfather(randy, jane) <- father(randy, A) & parent(A, jane) .
Trace this rule? [yes, no, up, help]: yes.
The proof for father(randy, sally) succeeded.
Should this answer lead to a successful proof?
[yes, no, debug, help]: no.
The proof for father(randy, joe) succeeded.
Should this answer lead to a successful proof?
[yes, no, debug, help]: yes.
parent(joe, jane) <- mother(joe, jane) .
Trace this rule? [yes, no, up, help]: no.
parent(joe, jane) <- father(joe, jane) .
Trace this rule? [yes, no, up, help]: yes.
There is no applicable rule for father(joe, jane) .
parent(joe, jane) <- father(joe, jane) .
Trace this rule? [yes, no, up, help]: up.
grandfather(randy, jane) <- father(randy, joe) & parent(joe, jane) .
Trace this rule? [yes, no, up, help]: up.
ailog:
```

6.3 Depth Bound

The search strategy used by AILog is a depth-bounded depth-first search. The depth bound is set using the command:

```
ailog: bound n.
```

where n is a positive integer. n is a bound on the depth of any proof tree. The default value is 30.

When a goal is asked and no more answers can be found, if the depth-bound was not reached, the system reports there are no more answers. If the depth-bound was reached, there still may be some answers, although it is more likely that there is a bug in the knowledge base where some recursions do not terminate. AILog allows the user to interactively explore the places where the depth bound was reached.

When no more answers can be found and the search was cut off due to hitting the depth-bound, the user is informed of this, as is given the option of one of:

i. where i is an integer bigger than the current depth-bound. This lets the user explore larger search trees. Repeatedly increasing the depth-bound lets the user simulate an iterative deepening search.

where. to let the user explore the place where the depth-bound was reached.

ok. to go back to the answer prompt.

help. for a list of the available commands.

When the user given the **where** command, the proof is retried, and it halts at a point where the depth-bound was reached, and shows the user the current subgoal, g . The user can give one of the following commands:

fail. to fail g , and explore other places where the depth-bound was reached.

succeed. to say that g should succeed.

proceed. to fail g and not explore any more subgoals where the depth-bound was reached.

why. to let the user explore why g was called (see Section 6.4).

ok. to go back to the answer prompt.

help. for a list of the available commands.

6.4 Why? Questions

If you find yourself at a particular subgoal, it is often useful to find out **why** that subgoal is being proved.

When you ask why an atom was being asked, AILog produces the instance of the rule in the knowledge base with the atom in the body, such that the head was trying to be proved. This is of the form:

```
h <-
  1: a1
  2: a2
  ...
** j: aj
  ...
  k: ak
```

This means that a_j is the atom that is being asked, the atoms $a_1 \dots a_{j-1}$ have been proved, and $a_{j+1} \dots a_k$ have still to be proved. When this is presented you can ask one of:

how i . where i is an integer $1 \leq i < j$. This means that you want to see how a_i was proved. This enters the how dialog.

how j . This means that you want to see how AILog is trying to prove a_j (the atom you have previously asked *why* about). This returns to the rule with a_j in the body.

why. to see the rule with h in the body.

prompt. to return to the AILog prompt.

help. for a list of the available commands.

Note that you can't ask **how** i for $i > j$ as there is no proof tree for the atoms $a_{j+1} \dots a_k$. Also, AILog does not guarantee that the rule given will have the atoms in the body in the same order as in the knowledge base; AILog is free to change the order of atoms in a body as long as this doesn't introduce an error.

Example 6 The following shows a trace of a looping program that reaches the depth bound. We first increase the depth-bound, and then explore where the depth-bound was reached.

```
ailog: tell a <- b & c & d.
ailog: tell b <- e.
ailog: tell e.
ailog: tell c <- f & d & g.
ailog: tell f <- b & h.
ailog: tell h.
ailog: tell d <- a.
ailog: ask a.
Query failed due to depth-bound 30.
  [New-depth-bound,where,ok,help]: 50.
Query failed due to depth-bound 50.
  [New-depth-bound,where,ok,help]: where.
Depth-bound reached. Current subgoal: e
  [fail,succeed,proceed,why,ok,help]: why.
e is used in the rule
b <-
** 1: e
  [Number,why,help,ok]: why.
b is used in the rule
a <-
** 1: b
   2: c
   3: d
  [Number,why,help,ok]: why.
a is used in the rule
d <-
```

```

** 1: a
[Number,why,help,ok]: why.
d is used in the rule
c <-
  1: f
** 2: d
  3: g
[Number,why,help,ok]: how 1.
f <-
  1: b
  2: h
How? [Number,up,retry,ok,help]: how 2.
h is a fact
f <-
  1: b
  2: h
How? [Number,up,retry,ok,help]: up.
d is used in the rule
c <-
  1: f
** 2: d
  3: g
[Number,why,help,ok]: why.
c is used in the rule
a <-
  1: b
** 2: c
  3: d
[Number,why,help,ok]: ok.
Depth-bound reached. Current subgoal:e
  [fail,succeed,proceed,why,ok,help]: fail.
No more answers.
ailog:

```

7 Ask-the-user

The ask-the-user facility exists for yes/no questions as well as for functional properties. The subgoals have to be ground before being asked of the user. This may be fixed in future implementations.

To to make an atom askable, you can issue the command:

```
ailog:askable atom.
```

Whenever a ground instance of the atom is attempted to be proved, the user is asked if it is true. The system may ask:

```
Is g true?
```

The user can reply with one of:

yes. if g is known to be true. The system will not ask this instance again.

no. if g is known to be false. The system will not ask this instance again.

unknown. if g is unknown. In this case any applicable clauses for g can be used.

fail. to fail the subgoal (but not record an answer). This is only used to test you axiomatization.

why. to see why the system was asking this question. This then enters the **why** interaction described in Section 6.4.

prompt. to return to the ailog prompt.

help. to get a menu of available answers.

Note the assumption that AIlog makes about the interaction between asking the user about g and using clauses for g . It assumes that if the user knows whether g is true, the user's answer should be used. Any rules for g are used only if the user doesn't know whether g is true or not.

Example 7 The following gives the example of the electrical example with askables (see Poole et al. [1998], page 214):

```
ailog: load 'ailog_code/ch07/elect_askable.pl'.
Ailog theory ailog_code/ch07/elect_askable.pl loaded.
ailog: ask lit(l1).
Is up(s2) true? [yes,no,unknown,why,help]: yes.
Is up(s1) true? [yes,no,unknown,why,help]: no.
Is down(s2) true? [yes,no,unknown,why,help]: no.
Is up(s3) true? [yes,no,unknown,why,help]: yes.
Answer: lit(l2).
[ok,more,how,help]: ok.
```

If f is functional (i.e., if there can only be one value for f), you can use the special syntax $f = V$ to mean that f has value V . Then when you issue the command:

```
ailog:askable $f=V$.
```

Then a goal of the form $f = X$ will ask the user for the value of f once, and remember the value. For example:

```
ailog: askable age(P)=V.
ailog: tell young(P) <- age(P)=A & A<13.
ailog: ask young(sally) & young(bruce).
What is the value of age(sally)? [value,unknown,why,help]: 3.
What is the value of age(bruce)? [value,unknown,why,help]: 87.
No. young(sally) & young(bruce) doesn't follow from the knowledge base.
Runtime since last report: 0.0 secs.
ailog: ask young(al) & young(sally).
```

What is the value of age(al)? [value,unknown,why,help]: **10.**

Answer: young(al) & young(sally).

Runtime since last report: 0.0 secs.

[ok,more,how,help]: **how.**

yes <-

1: young(al)

2: young(sally)

How? [Number,up,retry,ok,prompt,help]: **1.**

young(al) <-

1: age(al)=10

2: 10<13

How? [Number,up,retry,ok,prompt,help]: **1.**

You told me age(al)=10 is true.

8 Built-in predicates

There are a few built-in predicates. These cannot be redefined. The following arithmetic predicates are predefined:

- $X < Y$, where X and Y must be ground arithmetic expressions when called, is true if the value of X is less than the value of Y .
- $X > Y$, where X and Y must be ground arithmetic expressions when called, is true if the value of X is greater than the value of Y .
- $X =< Y$, where X and Y must be ground arithmetic expressions when called, is true if the value of X is less than or equal to the value of Y .
- $X >= Y$, where X and Y must be ground arithmetic expressions when called, is true if the value of X is greater than or equal to the value of Y .
- $X =\neq Y$, where X and Y must be ground arithmetic expressions when called, is true if the value of X is different to the value of Y .
- $V \text{ is } E$ where E must be a ground arithmetic expression when called. This is true if expression E evaluates to V .
- `number(N)`, where N is ground is true if N is a number.

Arithmetic expressions can either be numbers or of the form $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 * E_2)$, (E_1/E_2) , $\sin(E)$, $\log(E)$, etc.

Example 8 The following shows a simple use of `is` and `>`. It shows how we can use variables in the expressions, as long as they are bound before (to the left) they are used.

```
ailog: ask X is 3+4*5 & X*2 > 11*3.
```

```
Answer: 23 is 3+4*5 & 23*2>11*3.
```

The restriction that the arithmetic expressions must be ground when called is severe, and probably won't be fixed in future implementations.

There is one non-arithmetic comparison:

- $X \neq Y$ true if X and Y denote different objects under the unique names assumption.

X and Y must be bound enough to determine whether they can be unified. (Delaying is not implemented). This may be fixed in future versions.

9 Assumables

Assumables can be used for proof by contradiction or for abduction. Integrity constraints (rules that imply false) are handled specially in abduction. The interaction between assumables and negation as failure is not as general as it could be; for the goal $\sim g$, any proof or partial proof of g should not depend on any assumables.

To to make an atom assumable, you can issue the command:

```
ailog: assumable atom.
```

When the atom is encountered, it is assumed to be true. The assumptions used to make a proof go though are collected and presented to the user when the answer is returned.

```
ailog: tell a(X) <- b(X) & c(X) .
ailog: assumable c(X) .
ailog: tell b(X) <- d(X) & e(X) .
ailog: askable e(X) .
ailog: tell d(a) .
ailog: tell d(b) .
ailog: tell d(c) .
ailog: ask a(X) .
Is e(a) true? [yes,no,unknown,why,help]: yes .
Answer: a(a) .
Assuming: [c(a)] .
    [more,ok,how,help]: more .
Is e(b) true? [yes,no,unknown,why,help]: no .
Is e(c) true? [yes,no,unknown,why,help]: yes .
Answer: a(c) .
Assuming: [c(c)] .
    [more,ok,how,help]: more .
No more answers.
ailog:
```

An explanation has to be consistent with the knowledge base. To say that some body is not true, you can use:

```
ailog: tell false <- body.
```

Here *false* is a special atom that is false in all models. Thus, this clause provides a constraint specifying that the body is false.

From the set of constraints, you need to create a set of nogoods to ensure that all explanations are consistent. To do this, you can issue the command:

```
ailog: create_nogoods.
```

The nogoods correspond to the sets of assumables that imply *false*.

To list all of the nogoods, you can issue the command:

```
ailog: nogoods.
```

To debug the nogoods, you can ask to prove *false* or ask whynot *false*.

10 Negation-as-failure

Negation as finite failure is implemented. Use $\sim G$ to mean the negation of G . G can be any body but must be ground (i.e., contain no variables) when $\sim G$ is called; this may be fixed in future implementations. The interaction of negation as failure and the depth-bound is handled correctly.

There is a problem with negation and **why** questions; sometimes the rule written out omits some negation symbols. This only occurs when non-atomic formulae are negated. The interaction of negation as failure with assumables isn't as general as possible; it is only sensible if you make sure that any proof for a negated atom does not include assumables. Negation as failure does work with probabilities.

11 Probabilities

AIlog contains support for first-order probabilistic reasoning. You can have probabilities associated with atoms that have no associated rules defining them. These atoms are grouped into disjoint sets that correspond to random variables.

You can define these probabilistic assumptions using either:

```
ailog: prob atom:probability.
```

where *atom* is an atomic symbol and *probability* is a number between 0 and 1. *atom* can contain variables, in which case every ground instance is a proposition with the corresponding probability. All instances of the same atom, and all atoms defined in different *prob* declarations are assumed to be probabilistically independent.

The alternative syntax is to write:

```
ailog: prob  $a_1 : p_1, a_2 : p_2, \dots, a_k : p_k$ .
```

Where the p_i are non-negative numbers that sum to 1 and the a_i are atoms that share the same variables. In this case every ground instantiation of the a_i s forms a random variable: the a_i s are mutually exclusive and covering.

The atoms defined in a *prob* statement cannot be used in the head of rules, but can be used in the body. When called, the instance of these atoms must be ground.

We also allow statements where the probability is free, for example in:


```
ailog: prob heads_happens (E,P) :P.
```

This is useful when we want to learn parameters from observations. Note that when this is called, P must be bound.

There are two main interfaces to the probabilistic reasoning:

```
ailog: observe obs.
```

where *obs* is a body. This declares that *obs* has been observed. This returns the probability of *obs* conditioned on all previous observations. Observations accumulate, so that subsequent queries and observations are with respect to all previous observations.

You can ask the posterior probability of a query (this is the conditional probability given all of the current observations) using:

```
ailog: predict query.
```

where *query* is a body. This gives the posterior probability of *query*, conditioned on all previous observations.

Given any prediction or observations you can inspect the explanations of the query. An explanation corresponds to a proof of the query based on a set of the probabilistic atoms. For each of these explanations you can examine the proof tree, and ask how that explanation was computed. This can either be done after a proof, or as a stand-alone command:

```
ailog: explanations.
```

which returns the explanations of all of the observations.

You can also ask for the worlds in which the query and all of the previous explanations are true. This gives a set of descriptions of possible worlds where the possible worlds are described in terms of the probabilistic atoms that are true in them, and the descriptions are mutually exclusive. This can be used for computing the probability of the query (by summing over all of the worlds where it is true). The worlds are computed from the explanations by making sure that they are mutually exclusive. This can either be done after a prediction or observation or by issuing the command

```
ailog: worlds.
```

Which returns the worlds in which all of the observations are true.

Note that the **ask** command ignores all previous observations. A future version may allow you to ask in the context of previous explanations.

You can undo observations using

```
ailog: unobserve.
```

which undoes the last observation or using

```
ailog: unobserve all.
```

which undoes all observations.

The command

```
ailog: probs.
```

lists all probabilistic assertions.

Note that probabilities integrates cleanly with negation as failure and naively with the depth-bounded search. (It fails if the probability cannot be computed due to the depth-bound. A more sophisticated version may give bounds on the probability.) Probabilistic hypotheses and non-probabilistic hypotheses (using *assumable*) are not integrated and should not be used together.

Example 9 This may seem like not a very powerful probabilistic inference system, however, arbitrary Bayesian belief nets can be represented. For example, a belief network with Boolean variable *A* as a parent of variable *B* can be represented as:

```
ailog: tell b <- a & bifa.
ailog: tell b <- ~a & bifna.
ailog: prob a:0.6.
ailog: prob bifa:0.9.
ailog: prob bifna:0.3.
```

You can ask the prior on *a* using:

```
ailog: predict a.
Answer: P(a|Obs)=0.6.
Runtime since last report: 0.03 secs.
[ok,more,explanations,worlds,help]: ok.
```

And conditioning on *b*.

```
ailog: observe ~b.
Answer: P(~b|Obs)=0.34.
Runtime since last report: 0 secs.
[ok,more,explanations,worlds,help]: ok.
ailog: predict a.
Answer: P(a|Obs)=0.176471.
Runtime since last report: 0 secs.
[ok,more,explanations,worlds,help]: ok.
ailog:
```

Thus $P(a|b) = 0.176471$.

Note that Ailog does not find all explanations, but only those within a threshold. This threshold can be queried and varied with the `prob_threshold` command.

11.1 Experimental Probabilistic Features

The features in this section are only for specialized applications and may be changed in future applications. Do not use these unless you know what you are doing.

Sometimes we may observe arbitrary choices that we don't want to model the probability of. For example, that the rooms in an apartment are r_{17} , r_{53} and r_5 . We may have a probability that there are three, but the names may be arbitrary. Once these have been observed, then queries about what are the rooms should return these.

As another example, we may want to count rooms. Which is the first room and which is the second room is arbitrary, but we have to record which ones we have already counted so we don't count them again. We don't want to search over the choices of which is first and which is second.

```
ailog: arbitrary f=V.
```

means that the values V assigned to f are arbitrary, but f can only have one such value. Note that $f = V$ can be used as any other atom. The command `arbitraries` list all of the arbitrary declarations.

12 Copyright

AILog is “free”. This means that everyone is free to use it and free to redistribute it on certain conditions. AILog is not in the public domain, it is copyright, as follows:

Copyright © 1997–2007 David Poole.

AILog is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the license, or (at your option) any later version.

This program is distributed in the hope that it may be useful, but *without any warranty*; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more particulars. A copy of the GNU General Public License can also be obtained from the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

This documentation was written using Hyperlatex.

References

Poole, D., Mackworth, A., and Goebel, R. (1998). *Computational Intelligence: A Logical Approach*. Oxford University Press, New York.

Poole, D. (2008). The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton (Eds.), *Probabilistic Inductive Logic Programming: Theory and Application*, LNAI 4911. Springer Verlag. URL <http://www.cs.ubc.ca/spider/poole/papers/ICL-Beyond.pdf>.

Poole, D.L. and Mackworth, A.K. (2009). *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, New York, NY. URL <http://www.cs.ubc.ca/spider/poole/aibook/>.