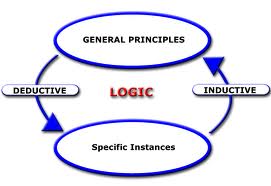
BABAR

Reasoning Engine



Design and API

"[*Cogito ergo sum*](http://en.wikipedia.org/wiki/Cogito_ergo_sum)" (Descartes, 1637)

# Introduction

This document describes a standalone theorem module that is at the core of Babar’s reasoning engine. It is essentially a backward chaining reasoned very much in the Spirit of Prolog. It supports a temporal reasoning facility based on Interval Time Logic (Allen, 86). Statements are implemented as ground clauses and inference rules are represented as Horn clauses.

Unless otherwise noted, all documented entries are in the *:prover* Common Lisp package. In all documented entries, the argument <database> defaults to *\*database\**.

# Initialization

(**INITIALIZE-DATABASE** &key <clause-descriptions> <database>)

(**RESET-DATABASE** &key <database>)

# Atomic Entities

This section documents the API functions for manipulating atomic entities. These include *constants*, *variables*, *predicates* and *temporal relations*.

# Constant Entities

(**MAKE-CONSTANT** <name> &key <value><class><database>)

This generic function returns an object of class constant-entity whose value will be set to <value> which defaults to <name>. The <name> should normally be a common lisp string. The argument <value> should be a common Lisp string, symbol or integer. This function will accept a symbol or integer as the value of the <name> argument. This is really for convenience and in this case the value of the constant will be set to <name> and the string representation of <name> will be used as the name of the constant. Finally this function adds the new constant object to the database of constant object. Constants are unique by name. Creating a new constant with the same name as an existing constant with a different value will effectively replace the existing database entry. See *ensure-constant*.

**Example**

PROVER(17): (make-constant "A")

#<Const: A>

(**FIND-CONSTANT** <name> &key<database>)

This function simply looks up <name> in the constants database and returns the constant named <name> or nil if no such named constant exists.l

**Example**

PROVER(23): (make-constant 'A)

#<Const: A>

PROVER(24): (find-constant "A")

#<Const: A>

(**DELETE-CONSTANT** <name> &key<database>)

This function deletes the constant named <name> and returns t if such a constant exists and was deleted. If no such constant exists, it simply return nil

**Example**

PROVER(25): (delete-constant "A")

T

(**ENSURE-CONSTANT** <name> &key<database>)

This function is really for convenience implements the notion of find-or-make constant. It looks up <name> in the constants database and returns the constant named <name> if such a named constant exists, otherwise it creates a new constant object and adds it to the constants database.

**Example**

PROVER(19): (find-constant 'foo)

NIL

PROVER(20): (setf x (ensure-constant "foo"))

#<Const: foo>

PROVER(21): (setf y (ensure-constant 'foo))

#<Const: foo>

PROVER(22): (eq x y)

T

(**MAP-CONSTANTS** <function> &key <database>)

This function applies <function> to each constant object in the constants database specified by <database>.

(**RESET-CONSTANTS** &key <database>)

This is for debugging purposes . Tis function deletes all constants from the database specified by <database>.

(**PRINT-CONSTANTS** &key <database)

This is for debugging purposes and simply prints all the constants in <database

# Predicate Entities

Predicates occur as the first component of any clause. They typically capture and name relation between the other components in a clause. For binary predicates, they essentially represent the edge label in a labeled graph.

(**MAKE-PREDICATE** <name> &key <value><class><database>)

This generic function returns an object of class constant-entity whose value will be set to <value> which defaults to <name>. The <name> should normally be a common lisp string. The argument <value> should be a common Lisp string, symbol or integer. This function will accept a symbol or integer as the value of the <name> argument. This is really for convenience and in this case the value of the predicate will be set to <name> and the string representation of <name> will be used as the name of the predicate. Finally this function adds the new predicate object to the database of predicate object. Predicates are unique by name. Creating a new predicate with the same name as an existing predicate with a different value will effectively replace the existing database entry. See *ensure-predicate*.

**Example**

PROVER(17): (make-predicate "A")

#<Predicate: A>

(**FIND-PREDICATE** <name> &key<database>)

This function simply looks up <name> in the predicates database and returns the predicate named <name> or nil if no such named predicate exists.l

**Example**

PROVER(23): (make-predicate 'A)

#<Predicate: A>

PROVER(24): (find-predicate "A")

#<Predicate: A>

(**DELETE-PREDICATE** <name> &key<database>)

This function deletes the predicate named <name> and returns t if such a predicate exists and was deleted. If no such predicate exists, it simply return nil

**Example**

PROVER(25): (delete-predicate "A")

T

(**ENSURE-PREDICATE** <name> &key<database>)

This function is really for convenience implements the notion of find-or-make predicate. It looks up <name> in the predicates database and returns the predicate named <name> if such a named predicate exists, otherwise it creates a new predicate object and adds it to the predicates database.

**Example**

PROVER(19): (find-predicate 'foo)

NIL

PROVER(20): (setf x (ensure-predicate "foo"))

#<Predicate: foo>

PROVER(21): (setf y (ensure-predicate 'foo))

#<Predicate: foo>

PROVER(22): (eq x y)

T

(**MAP-PREDICATES** <function> &key <database>)

This function applies <function> to each predicate object in the predicates database specified by <database>.

(**RESET-PREDICATES** &key <database>)

This is for debugging purposes . Tis function deletes all predicates from the database specified by <database>.

(**PRINT-PREDICATES** &key <database)

This is for debugging purposes and simply prints all the predicates in <database

# Variable Entities

(**MAKE-VARIABLE** <variable-name> &optional <value>)

This generic function returns an object of class variable-entity whose value will be set to <value> if specified. The <variable-name> argument should normally be a common lisp string. This function will accept a symbol the value of the <variable-name> argument if which case the symbol name of the symbol will be used as the variable name.

Variables must start with a question mark character. If <variable-name> does not start with a question mark, this function will prefix the name with a question mark.

Additionally because variables with the same name across different clauses need to be distinguished, this function also as a side effect increments the database’s variable counter and this number is cached in the variable and is used in the printed representation of the variable name.

**Note:** Need to consider including the number in the actual variable’s name

**Examples**

PROVER(100): (make-variable "?X")

#<Var: ?X1>

PROVER(101): (make-variable 'X)

#<Var: ?X2>

(**BIND-VARIABLE** <variable-entity> <value>)

This function sets the value of the variable entity <variable> to be <value>. It also sets the variable’s bound status to true. The argument <variable-entity> should be an object of class variable-entity.

Note: A variable can be bound to another variable.

**Example**

PROVER(107): (setf x (make-variable '?X))

#<Var: ?X3>

PROVER(108): (bind-variable x "Elephant")

"Elephant"

PROVER(109): (variable-value x)

"Elephant"

(**UNBIND-VARIABLE** <variable-entity>)

This function clears any binding that <variable-entity> currently has and sets its bound status to false.

**Example**

PROVER(109): (variable-value x)

"Elephant"

PROVER(110): (unbind-variable x)

NIL

PROVER(111): (variable-value x)

NIL

(**VARIABLE-BOUND-P** <variable-entity>)

This predicate function returns true if <variable-entity> currently has a binding and its bound status is true.

(**VARIABLE-VALUE** <variable-entity> &key <deepest>)

This returns the current binding of <variable-entity> if any. If <deepest> is true and <variable-entity>is bound to another variable then the variable-value of the variable that <variable-entity> is bound to, is returned when <deepest> is true.

**Example**

PROVER(134): (setf x (make-variable '?x))

#<Var: ?X7>

PROVER(135): (setf y (make-variable '?y))

#<Var: ?Y8>

PROVER(136): (bind-variable x y)

#<Var: ?Y8>

PROVER(137): (bind-variable y "Hello, world")

"Hello, world"

PROVER(138): (variable-value x :deepest-p nil)

#<Var: ?Y8>

PROVER(139): (variable-value x :deepest-p t)

"Hello, world"

# Advanced Constant Entities

## Skolem Constants

(**MAKE-SKOLEM-CONSTANT** <name>)

## Builtin Predicates

(**MAKE-BUILTIN-PREDICATE** <name>)

# Clauses

## Clause Entity

(DATABASE-CLAUSES <database>)

## Clause Constructors

(MAKE-CLAUSE <class> <predicate> <arguments> &rest <rest>)

## Clause Accessors

(CLAUSE-PREDICATE <clause>)

(CLAUSE-ARGUMENTS <clause>)

(CLAUSE-CONSTRAINT <clause>)

(CLAUSE-BINDINGS <clause>)

(CLAUSE-DESCRIPTION <clause>)

## Clause Parser Functions

### Public API Functions

(**PROCESS-CLAUSE-DESCRIPTION** <clause-description> &key <database>)

**Example 1**

PROVER(3): (setf c (process-clause-description '(P ?x A ?y ?x)))

#<Clause: (P ?X A ?Y ?X)>

PROVER(4): (describe c)

#<Clause: (P ?X A ?Y ?X)> is an instance of #<STANDARD-CLASS CLAUSE-ENTITY>:

The following slots have :INSTANCE allocation:

UTILITIES::NAME "P ?X A ?Y ?X"

UTILITIES::VALUE NIL

PREDICATE #<Predicate: P>

ARGUMENTS (#<Var: ?X1> #<Const: A> #<Var: ?Y2> #<Var: ?X1>)

CONSTRAINT NIL

VARIABLES (#<Var: ?Y2> #<Var: ?X1>)

DESCRIPTION (P ?X A ?Y ?X)

BINDINGS NIL

ARITY 4

**Example 2**

PROVER(46): (setf hc (process-clause-description '(HORN (P ?x)(Q ?)(R ?x))))

#<Horn: #<Clause: (P ?X)> #<Clause: (AND (Q ?) (R ?X))>>

PROVER(50): (horn-clause-lhs hc)

#<Clause: (P ?X)>

PROVER(51): (horn-clause-rhs hc)

#<Clause: (AND (Q ?) (R ?X))>

### Private API Functions

(**PARSE-CLAUSE** <clause-description> &key <variables> <database>)

(**PARSE-CONJUNCTIVE-CLAUSE** <clause-description> &key <variables> <database>)

(**PARSE-DISJUNCTIVE-CLAUSE** <clause-description> &key <variables> <database>)

(**PARSE-HORN-CLAUSE** <clause-description> &key <variables> <database>)

(**PARSE-HOLDS-CLAUSE** <clause-description> &key <variables> <database>)

(**PARSE-TEMPORAL-CLAUSE** <clause-description> &key <variables> <database>)

# Automated Reasoning

This section documents the automated reasoning facilities of Babar’s reasoning engine. This is implemented as a backward chaining Horn Clause theorem prover. The following documented entities operate on the various types of clauses described earlier. A proof tree is constructed, as a result of the theorem proving efforts and can subsequently be used to provide an explanation or trace of the proof itself.

(**PROVE** <clause> &key <database>)

This function attempts to find a proof of <clause> in the context of <database> It returns as multiple values a clause object and a list of variable bindings if a proof of <clause> was found, otherwise it returns nil. The argument <clause> can be a clause-entity object or a declarative clause description. In this latter case, the parse-clause function be invoked on the declarative clause description.

**Example 1**

PROVER(60): (add-clause '(P A ?x))

#<Clause: (P A ?X)>

PROVER(61): (setf c (parse-clause '(P A B)))

#<Clause: (P A B)>

PROVER(62): (prove c)

#<Clause: (P A B)>

((#<Var: ?X1> #<Const: B>))

Exa