Atelier B

Redaction guide for mathematical rules

Draft

This document is a translation. The author of the original document is Clearsy Ltd. The responsible for this translation to English is David Deharbe. This translation has not yet been submitted to endorsment to Clearsy Ltd.

Introduction

This document is targeted to users of the prover that need to write mathematical rules to facilitate in the verification of the generated proof obligations. As a matter of fact:
the proof in predicate logic is undecidable
the rule base of the prover is finite (it has approximatively 2800 rules)
It may therefore be necessary to add rules, either in a Pmm file or in a PatchProver file.

We draw the attention of the reader on the unfortunate consequences of inadequate rules, that may induce an undesirable behavior of the prover. Particularly, employing false rules enables showing correct false proof obligations and thus jeopardize the development of a correct software.

This document contains advice to write rules that are correct and which verification will be easier to perform.

Bibliography

[1] Redaction guide for mathematical rules (this document)
[2] Reference manual of the B language
[3] Reference manual of the interactive prover

Terminology

Abbreviations

PO: proof obligation
RPT: rule proving tools
TRATL: translator of rules written in the theory language
RB: rule base of the automatic prover

Glossary

Proof obligation: logic predicate produced by Atelier B from a component (machine, refinement, implementation), written in the B language and that needs to be proved to guarantee the soundness of this component.

Rule base: Set of mathematical rules written in the theory language that are necessary for the prover to achieve proofs.

Pmm: A file containing user rules that enrichrd the RB used for a component.

PatchProver: A file containing user rules that enriches the RB used for a project.

Introduction to the theory language

The mathematical rules used by the prover are written in the so-called "theory language". Without getting into the details of this language, which is similar to PROLOG, the following sections expose the fundamental notions that may be employed to use the theory language for the purpose of defining mathematical rules.

What is a wildcard?

A wildcard is a value that can take any value (literal, expression, etc.)
If it is assigned a value, then it is said to be instantiated.

The sole mechanism to represent a variable is the wildcard.
A wildcard is denoted as a single (latin) alphabet letter: one cannot have more that 52 wildcards inside a rule (considering both uppercase and lowercase letters).

For instance, the expression

    a + bb*cc - d

contains wildcards a and d and literals bb and cc.
Wildcards a and d may be instantiated with ee+1 and 3, respectively. We then obtain the expression:

    ee + 1 + bb*cc - 3

What is a formula?

A formula is an expression that is built out of

- wildcards,
- literals and numbers
- logical connectors:
    - conjunction : &
    - disjunction : or
    - implication : =>
    - équivalence : <=>
    - negation : not(a)
- quantifications:
    - universal quantification : !
    - existential quantification : #
- equality : =,
- set membership : :,
- set inclusion : <:
- operators to combine arithmetic, set, boolean, relational, functional and sequence expressions of the B language (see [2]), obeying the syntax of the B language.

For instance,

    0=<aa & 0=<bb => 0<=aa*bb
    a=TRUE or b=TRUE => a : BOOL

are syntactically correct formulas.
According to this definition, formulas need not have a boolean type. This terminology is confusing. -Deharbe 01/04/09 09:06

Matching formulas

A formula f is said to match a formula g, if it is possible to obtain f by substituting, in g, all the occurences of the same wildcards with some formulas . Recall that a wildcard is an atomic formula composed of a single letter . A wildcard is thus a "formula variable". For instance the following formula g:

aa + (bb/ee - (cc + dd)*aa) - bb/ee

matches the following formula f:

x + (y - z*x) - y

An assignment of formulas to wildcards is called a filter. A filter is thus a partial function from wildcards to formulas. Applying a filter to a formula g consists in replacing each wildcard occurence in g, that belongs to the domain of the filter, by the corresponding formula. In summary, a formula f matches a formula g if there is a filter such that the application of that filter to g yields f. In the case of the previous matching, we have the following filter:

{ x |-> aa, y |-> bb/ee, z |-> cc + dd }

What is a rule?

A rule is a formula with the following form A => B. A is called the antecedent of the rule, B is called the consequent of the rule. A and B may be conjunctions of predicates.
A may be omitted, in that case, the rule is said to be atomic.
A rule may be:

- inductive (*backward*)
  If the current goal is B, then to prove B it is sufficient to prove A. A is supposed to be simpler, or easier to prove, than B.
  For instance, with the rule
  `x = FALSE => not(x = TRUE)`
  the goal
  `not (bool(0 <= aa*bb) = TRUE)`In the original text, the relational operator is denoted =< -Deharbe 01/04/09 09:15
  is proved if the derived goal
  `bool(0 <= aa*bb) = FALSE`
  is proved.
  Similarly, the atomic rule
  `not (BOOL = {})`
  is applied to prove immediately the goal with the same form.
- deductive (*forward*)
  If the antecedent A has the following form:
  A1 & A2 &... & An
  and if A1 is an hypothesis that has just been generated, and if
  A2 & ... & An
  are in the hypothesis stack then instances of predicates in the conjunction B are generated and, if they do not appear in the hypothesis stack, are pushed onto that stack.
  For instance, with the rule:
  `not(a=b) & a=c => not(b=c)`
  If the hypothesis not(xx = 3) has just been generated, and the hypothesis xx = aa * bb - cc is in the hypothesis stack, then the hypothesis not(3=aa*bb-cc) is generated.
- rewrite
  In this kind of rules, B has the form C == D.
  If A is satisfied, then C is rewritten to D.
  This kind of rules only applies to subformulas of the current goal, or on the

current goal itself.
For instance, the rule SimplifyIntMaxXY.3:

```
btest(p<=q)                 /* If p and q are integers */
=>                          /* such that p <= q */
max({p}\/{q}) == q          /* then max({p}\/{q}) rewrites to p */
```

The original text uses braces instead of parenthesis for the argument to max -Deharbe 01/04/09 09:29
applies to the goal

```
0 <= max({3}\/{5}) - min(1..4)
```

The original text uses =< and neither <= nor \le -Deharbe 01/04/09 09:30
and rewrites it to

```
0 <= 5 - min(1..4)
```

The rules contain wildcards, whereas hypothesis do not.

Application of a rule to a formula

To realize formal proofs, inference rules are applied. This section explains what is meant by applying an inference rule to a formula.
The result of the application of a rule r to a formula f yields other formulas that are called the SUCCESSors of f. The set of SUCCESSors may be empty. Also note that a rule is not always applicable to a formula. When a rule r is applied to a formula f, r is said to discharge f and produces a number of formulas.

We are now going to state in which conditions a rule is applicable to a formula and, if when it is, define the result of this application. Recall that a rule has the following general form:

```
a1 & a2 & ... * an => c
```

where a1, a2,... , an are called the rule antecedents, and c is the rule consequent. To apply a rule to a formula, one must consider two cases, depending on the nature of the consequent c.

Deduction rule

When the consequent of the rule does not have the form g == d, then the rule is a deduction rule. For instance, the following is a deduction rule:

```
x < z & y < z => x+y < 2*z
```

A deduction rule is applicable to a formula f when f matches the consequent of r, yielding a filter. The result of the application of r to f is then the application of this filter to the different antecedents of r. This result may be empty when the rule has no antecedent.
For instance, the application of the previous rule to the following formula:

```
aa+bb+cc < 2*(cc+dd)
```

produces the two following formulas:
aa+bb < cc+dd
cc < cc+dd
It might be relevant to explain why this filter was chosen, instead of {x |-> aa, y |->bb+cc, z |-> cc+dd} -Deharbe 01/04/09 10:01

Rewrite rule

When the consequent c of the rule has the form g = dd, then the rule is called a rewrite rule. In such rules, formula g and d are called the left hand side and the right hand side. For instance, the following is a rewrite rule (without antecedent):

```
x*(y+z) == x*y + x*z
```

Such rule is applicable to a formula f if there exists a sub-formula h of f such that h matches the left-hand side of the consequent of r, yielding a filter. The result of the application of r to f corresponds first, as in the previous case, to the application of the filter to the antecedents of r, if there is any. The result also contains the formula obtained by replacing, in f, the sub-formula h by the application of the filter to the right-and side of the consequent of r. Consequently, the application of the previous rule to the formula

```
aa + bb + cc*(ee+ff) < cc + dd
```

produces the following formula:

```
aa + bb + (cc*ee + cc*ff) < cc + dd
```

If several subformulas match the left-hand side of the consequent, the "rightmost" sub-formula is chosen. For instance, when the rule

```
a+b == b+a
```

is applied to formula

```
aa+bb+cc = cc+bb+aa
```

four subformulas match the left hand side, namely:

```
aa+bb
aa+bb+cc
cc+bb
cc+bb+aa
```

The selected sub-formula is thus cc+bb**+**aa. The rewrite rule results in the following formula:

```
aa+bb+cc = aa+(cc+bb)
```

What is a theory?

A theory is a group of rules, written in the theory language. Two consecutive rules are separated by a ';' (semi-colon). The rules are (implicitly) named t.n, where
t is the name of the theory
n : is the position of the rule in the theory, the first position being numbered 1.

Example :

```
THEORY th1 IS
    binhyp(a: B) & /* Rule th1.1 */
    binhyp(B<: C)
    =>
    a: C;
    btest(0 <= -t) /* Rule th1.2 */
    =>
    0<=t**2 - 4*t + 1
END
```

The prover first tries to apply rules with a higher index before a rule with a lower index (from the bottom to the top of the theory).
For instance, when using the command ar(th1) (see [3]), the prover always tries to apply rule th1.2 before th1.1. In the case of this command, a rule th1.n only applies when no rule th1.m (such that n < m) applies. When a rule has been applied, the prover repeats its search from the last rule of the theory.

Proof

Proof of a formula

Given a set of rules, a formula f is said to be formally proved under this set of rules, when the repeated applications of these rules to the initial formula f and its SUCCESSors, and the SUCCESSors thereof, and so forth, results in an empty set. Equivalently, the formula f is proved under a set of rules when f and all its descendants are discharged by said these rules.

The initial formula and its descendants are called the goals of the proof. The initial formula is the initial goal, and its descendants are intermediate goals that appear during the course of the proof. By definition, all the goals are proved at the end of the proof.

This definition leaves space to some non-determinism, as we specify neither the order in which the intermediate goals are proved, nor the order in which the rules are chosen to discharge a given goal.

For instance, given the following rules:

```
x < z & x < z => x+y < 2*z
a-a == 0
x < x+1
0 < x+1
```

the goal is the following formula:

```
aa + (bb-bb) < 2*(aa+1)
```

The first rule may be used to discharge the initial goal, producing two intermediate goals:

```
aa < aa+1 bb-bb < aa+1
```

The first goal is discharged with the third rule, without creating any new goal. The second intermediate goal still needs to be proved. It is easy to see that can be done by discharging the second and then the fourth rule. Since neither applications produce new goals, we reach the point where no goal is left to be proved. The initial goal is then formally proved under the enuntiated set of rules.

Special case of conjunctive formulas

When a goal has the form:

```
f1 & f2 & ... & fn
```

the proof of this goal is replaced by the proof of each formula f1, f2, ... fn, that become new intermediate goals.

Tips to write mathematical rules

Order of rule applications

Rules may be separated according to the order of their application: forward (generating new hypothesis) and backward (generating new goals or resolution). When a rule is written, such order must be stated, either in the name of the corresponding theory, or through comments. Rules with different application order shall not be grouped in a same theory.

Restricting the application domain of a rule

Mathematical rules contain formulas conforming to the syntax of the B language. It is however possible to restrict the application domain of a rule so that it remains valid, or to parameterize it, by using guards of the theory language. Such guards are presented in Chapter 7.
For instance, in the rule

```
bsearch((x: p..q), P, r) &
(SIGMA(x).(P & 0 <= -E | -E)<=SIGMA(x).(P & 0<=E | E))
=>
(0 <= SIGMA(x).(P | E))
```

the bsearch guard ensures that P contains the definition domain of variable x as an interval.

In the rule

```
band(binhyp(n<=size(a)) ,
btest(n>0)) &
(size(a) = 1 => b = c) &
(2<=size(a) => first(a) = c)
=>
(first(a<+{size(a)|->b}) = c)
```

the two guards express that the sequence a has at least one element.

Equivalence rules

The fundamental principle to obey when writing rules is to write equivalence rules, that is, when
writing a rule a => b, it is equivalent to show a to show b. This means that the provability of b is preserved. Otherwise the produced goal may be erroneous and prevent th proof when the rule is applied without control.
For instance, the rule

```
0<=a &
0<=b
=>
0<=a*b
```

is not an equivalence rule. It cannot be used to show the goal 0<=a*b when a<0 and b<0.

However, the rule

```
binhyp(0<=a) &
binhyp(0<=b)
=>
0<=a*b
```

is an equivalence rule.

It is possible to write rules that are not equivalence rules. Such rules may only be employed in interactive proofs, as a single application. There is no risk of showing a false goal; only to reach an unprovable goal.

Rewrite rules

Application of rewrite rules within the scope of a quantifier must be done with care. In the case of rewrite rules with an antecedent, the application is only correct if the context variables appearing on the left-hand side of the consequent are bound at the rewrite position.
For instance, the rule:

```
binhyp(x=0)
=>
(x == 0)
```

transforms the expression !x1.P(x1) into !0.P(0) under the hypothesis x1 = 0. The problem comes from the fact that the variable bound to the wildcard x in the guard may not be the same as that of the rewriting position (the rule matches since they are synonymous).

The combination of guards blvar(Q) & Q\(H) may be used to prevent erroneous rewriting within quantified formulas. Every rewrite rule that needs a restriction with blvar (beware of the instantiation of H). The correct way to write the preceding rule is

```
binhyp(x=0) &
bgetallhyp(H) &
blvar(Q) &
Q\(H)
=>
(x == 0)
```

The paragraph on the use of these guards cannot be understood at this point. This reads as a magical incantation to avoid a well-identified problem. -Deharbe 01/04/09 11:34

Forward rules

The behavior of forward rules is significatively different from that of backward rules; in practice, there is little use of such rules outside of the core of the prover.
A forward rule may only apply when the hypothesis have just been generated and before they are pushed onto the stack hypothesis (and before they are reachable through the guard binhyp).
For instance, applying the backward rule

```
(not(X<=0 & Y<=0) => 0<=X & 0<=Y)
=>
(0<=X*Y)
```

on the goal

```
0<=(aa+bb)*(cc+3)
```

results in the new goal

```
not(aa+bb<=0 & cc+3<=0) => 0<=aa+bb & 0<=cc+3
```

Using the deduction mechanism, then prover then stacks the hypothesis

```
not(aa+bb<=0 & cc+3<=0)
```

and the new goal is:

```
not(aa+bb<=0 & cc+3<=0)
```

It is when the hypothesis is pushed on the stack that forward rules are applicable.

A forward rule has the form:
A1 ∧ A2 ∧ . . . ∧ An ⇒ B1 ∧ B2 ∧ . . . ∧ Bp

where A1 is a newly generated hypothesis and
A2 ∧ . . . ∧ An
are in the hypothesis stack. The hypothesis B1, B2,... Bn are then generated and pushed onto the hypothesis stack, if they are not already there.

Examples:

```
(u\/v = w)
=>
u: POW(w) & v: POW(w)
```

```
not(b: a) &
not(b..c/\a = {})
=>
not(b+1..c/\a = {})
```

Such rule may be restricted by guards that shall obrigatorily be placed after hypothesis A1. For instance,

```
not(a:POW({x})) &
bgetallhyp(H) &
bfresh(zz,H,z)
=>
#z.(z:a & not(z=x))
```

Lists

Lists hall be manipulated with extreme care in rules. For instance, formula

```
[a]
```

may match

```
[aa, bb, cc]
```

and also

```
[aa]
```

It is then important to be able to state in a rule if a wildcard shall match a literal or a list. In the previous case, if one wishes to match sequences with a single element, the following guard may be used

```
bnot(bpattern(a, (u, v))
```

Some rules that apply to lists may be grouped by pairs, where the first rule handles the general case, i.e. when the list has at least two elements, and where the second rule handles the special case of a list with a single element. The latter rule shall not be applied to a list with multiple elements, as it would yield an unprovable goal.
For instance, the prover rules:

```
bnot(bpattern(a, (u,v))
=>
([a]1 == a)
[a,b]1 == [a]1
```

are used to simplify expressions selecting the first element of a sequence. Note the recursive application expressed in the second rule, as the matching always selects the rightmost occurence of a pattern.

Tips for parenthesis

Writing rules requires using parenthesis. Misplacing parenthesis may result in flawed rules. The following tips advise on the correct usage of parenthesis:

- Parenthesis and existential quantifiers
  For compatibility with the Linux port of Atelier B, it is required to over-parenthesize formulas under the scope of an existential quantifier.

#X. P shall be written (#x. P) to be used under Linux.

- Quantified predicates
  Parenthesizing of a quantified predicate is not guaranteed. For instance, the following rule:
  ```
  Antecedent => (!x.(x: E & P(x)))
  ```
  is too restrictive as it does not apply to goals such as
  ```
  !xx. (xx : E & A(xx) & B(xx))
  ```

- Operator precedence (and implicit parenthesis) in the theory language might be ambiguous to the user. For instance
  bsearch (aa, aa => bb, C)
  is interpreted and implicitly parenthesized as bsearch((aa, aa) => (bb, c)) in the prover.
  Also:
  - rule a == b & c is interpreted as ((a == b) & C)
  - rule a => b => c is interpreted as ((a => b) => c).

To avoid surprises in the application of rules in the prover, it is recommended to be systematic in the use of parenthesis when writing such rules.

Wildcard instantiation

Three types of problems may happen with wildcards:

- Non-instantiated jokers in the rule antecedent, or in the right-hand side of a rewrite rule consequent provoke the inclusion of so-called "dead wildcards". Dead wildcards are wildcards that become identifiers and are no longer part of the pattern matching process. This creates a new variable, not present in the source code
  For instance:
  ```
  e: FIN(s) => card(e): NATURAL
  ```
  s is a not instantiated and will become a dead wildcard.
- Renaming is a special case of a non-instantiated wildcard that happens when two distinct wildcards designate the same mathematic object. For instance,
  ```
  binhyp(s~: E +-> NATURAL) &
  1<=size(f)
  =>
  (tail(s)~: E +-> NATURAL)
  ```
  f represents the same sequence as s.
- The last problem is the name confusion occuring when the same wildcard represent different mathematical objects. In that case, either the rule is badly typed, or the rule is correct but very restrictive.
  ```
  band(binhyp(r~: C +-> B)
  binhyp(C: FIN(D))) &
  (a: POW(r<+{a}))
  =>
  (a: FIN(r<+[a}))
  ```
  both occurences of the wildcard are incompatible from the viewpoint of typing.

When writing a rule, special care must be taken regarding wildcards. "Copy&paste" is specially dangerous regarding to that kind of problem. Coherence in the use of wildcards always need to be checked.

The limited number of available wildcards hinders the readability of rules. However, as long as possible, the following conventions must be applied:

- f , g, h for functions,
- r for a relation,

- A, B, C, D, E, F for sets,
- the same letter, lowercase, for an element of the set,
- s, t for sequences,
- i, j, k, m, n, p for integers,
- P, Q, R for predicates,
- x, y, z for variables.

Ambiguities

Some notations lead to ambiguities:

- - denotes both arithmetic subtraction and set difference;
- * denotes both arithmetic product and cartesian product.

The rules that may lead to false proof due to overloading such operators must be corrected or eliminated. Ambiguous expressions need to be correctly typed and disambiguated. If it is not possible to remove ambiguities, the rule must be eliminated.Here I have interpreted the original text that was too vague. -Deharbe 01/04/09 13:36

Miscellaneous remarks

The exact semantics of guards is defined in Appendix C . Special care must be taken with the following points:

- The guards best(a=b) and (btest a/=b), differently from other uses of btest, do not check that a and b are elements of NAT. Beware, if btest(a=b) passes then a=b, however if btest(a/=b) passes, it is not necessarily the case that a /= b. Indeed if x and y are two different wildcards that match the same expression, btest(x /= y) passes but x = y is true.
- The guard btest(a+b=b+a) fails since a+b is not an identifier.
- The guard bsubfrm is used mainly to guide the proof. This guard shall not be employed for other purposes in rules.
- The guard bsearch only performs one extraction.
  bsearch(a, b, c): b needs to be a list with at least two elements, under the form x op y, where op is an associative and commutative operator. The associative and commutative properties of the operator are fundamental as the give the same importance to the different occurences of a within b. If op does not have these properties, the translation in the language of mathematics is no longer general.what did they mean? -Deharbe 01/04/09 13:43

Using the mechanisms of the prover

This section presents different mechanisms of the prover that may be referred to and employed in the mathematical rules. These mechanisms have been validated and may simplify the conception phase of these rules. For each mechanism, the following points will be presented:

- syntax to use the mechanism
- semantics associated with the mechanism
- a textual description of the mechanism
- an illustrating example.

Trying a proof

Syntax: bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(P | Curr))

Translation: P

Description: This mechanism triggers a sub-proof on P. It is successful if P can be demonstrated by the prover.
P must be correctly typed.

Examples

```
band(binhyp(f: s +-> (t --> u)) ,
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
(t: POW(a) & u: POW(b)) | Curr)))
=>
(f(x): a +-> b)
```

```
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
(not(b) => a) | Curr))
=>
(a or b)
```

```
(n: a) &
not(n: b) &
bguard(attempt_to_prove: a_t_t_e_m_p_t__t_o__p_r_o_v_e(
(a: POW(b))| Curr))
=>
bfalse
```

This last rule is a forward rule. If an hypothesis matching n: a has just been generated, if there is an hypothesis not(n: b) and if the prover is able to show that a: POW(b) is true, then the hypothesis bfalse is generated.

Difference

Syntax: bguard(Fast_Prove_Difference: not(a = b)

Translation: not(a = b)

Description: This mechanism may be employed to prove that two terms are not equal. If the call succeeds, then a and b are not equal.

Examples :

```
bguard(Fast_Prove_Difference: not(a = b)) &
blvar(Q) &
Q\not(a = b)
=>
({a}<<|{b|->c} == {b|->c})
```

```
bguard(Fast_Prove_Difference: not(a = c)) &
blvar(Q) &
Q\(a = c)
=>
({a|->b}<+{c|->d} == {a|->b}\/{c|->d})
```

Order relation

Syntax: bguard(Fast_Prove_Order: m<=n)

Translation: m<=n

Description: If the call to this mechanism is successful , then the relation m <= n is true.

Example:

```
bguard(Fast_Prove_Order: c<=d) &
(a = c) &
(b = d)
=>
(a..b = c..d)
```

Positive value

Syntax: bguard(Fast_Prove_Positif: a)

Translation: 0<=a

Description: If the call to this mechanism succeeds, then the value of the arithmetic expression a is positive or zero.

Example :

```
binhyp(s: POW(a..b)) &
bguard(Fast_Prove_Positif: a)
=>
(s: POW(NATURAL))
```

Non membership

Syntax: bguard(Fast_Prove_Distinction: not(x: s))

Translation: not(x: s)

Description: If the call to this mechanism is successful , then x does no belong to s.

Example :

```
Fast_Prove_Distinction(not(a: dom({c|->d})))
=>
not({a|->b} = {c|->d})
```

Membership to INTEGER

Syntax: bguard(Fast_Prove_Num: n)

Translation: n: INTEGER

Description: A call to this mechanism is successful entails that n is an integer.

Example :

```
bguard(Fast_Prove_Num:(x))
=>
(succ(x) == x+1 )
```

Guards

This chapter presents succinctly the different kinds of guards that are available to write mathematical rules (for further details, see Appendix C). For each guard, the following elements are presented:

- syntax,
- possibly its semantics,
- a textual description of its behaviour.

It must be noted that certain guards have a mathematical semantics only under special circumstances (e.g. the bsearch guard).

The following guards may be employed to obtain one or several hypothesis from the stack, by processing first the most recently inserted hypothesis.

Syntax	Semantics
bgetallhyp(H)	H
bgethyp(H)	H
binhyp(H)	H

The following guard checks if s is an element of STRING.

Syntax	Semantics
bstring(s)	s : STRING

The following guard checks if P is a conjunction.

Syntax	Semantics
bpattern(P, (Q & R))	P <=> (Q & R)

The bsearch guard has a mathematic semantics when the list in which the search is performed is the argument list of an associative and commutative operator. This is the case for the operators &, or, \/ and /\. This is also the case of the operator , (comma) for a list of quantified variables. One must thus check that the application of the rule remains within such cases.

Syntax	Semantics
bsearch(N, (P, Q), M)	(P, Q) = (M, N)
bsearch(N, (P & Q), M)	(P & Q) <=> (M & N)
bsearch(d, (a or b), c)	(a or b) <=> (c or d)
bsearch(d, (a \/ b), c)	a \/ b = c \/ d
bsearch(d, (a /\ b), c)	a /\ b <=> c /\ d

The following guard succeeds if t is true and has the form a op b.

Syntax	Semantics
btest(t)	t

The following guard checks if n is a numeric value both positive and smaller or equal to maxint.

Syntax	Semantics
bnum(n)	n : NATURAL

The following guard succeeds if both arguments succeed.

Syntax	Semantics
band(P, Q)	Semantics of P and semantics of Q

The role of the following guards is to verify the relationship between wildcards representing variables and wildcards representing expressions.

Syntax	Semantics
bvrb(x)	x
blvar(x)	x
x\P	x such that x \ P
bfresh(x, P, y)	y (y\P)

I am not sure how is this supposed to be helpful for the reader. -Deharbe 01/04/09 14:32

A Expression normalization

In order to limit the number of rules in the prover base, expressions are normalized. Every expression that the prover manipulates must have been first normalized. Thus every user rule found in a Pmm file is automatically normalized when it is loaded. However the rules that are stored in the PatchProver need to be manually normalized, otherwise they may induce an anomalous behavior of the prover.

The normal forms are the following:

Expression	Normal form
n > m - 1	m <= n
m < n	m <= n - 1
a <=> b	(a => b) & (b => a)
a <: b	a : POW(b)
a <<: b	a : POW (b) & not (a = b)
a /: b	not (a : b)
a /= b	not (a = b)
a /<: b	not (a : POW(b))
a /<<: b	a : POW(b) => a = b
a : NATURAL	a : INTEGER & 0 <= a
NATURAL1	NATURAL - {0}
NAT1	NAT - {0}
FIN1(A)	FIN(A) - {}
POW1(A)	POW(A) - {}
seq1(A)	seq(A) - {}
iseq1(A)	iseq(A) - {{}}
perm(E)	iseq(E) /\ (NATURAL - {0} +->> E)
<>	{}
{x, y}	{x} \/ {y}
{x | P}	SET(x).P

When writing a rule, it is necessary to check that this rule is indeed normalized. Otherwise, the rule will be normalized when it is loaded and may no longer be applicable as desired.

For instance, the following rule:

```
btest(0<x)
=>
0<=x**2-1
```

is normalized into

```
btest(0+1<=x)
=>
0<=x**2-1
```

But the guard btest only accepts as argument expression of the form a op b, where a

and b are integers. This rule will thus never be applicable. The following rule should be thus preferred:

```
btest(1<=x)
=>
0<=x**2-1
```

B Common pitfalls

B.1 What is an infinite loop of the prover?

For instance, consider applying the rule:
a * a == a * a * a / a
on the following goal
cc(v) = vv * vv
We will obtain SUCCESSively the following goals:
cc(v) = vv * vv * vv / vv
cc(v) = (vv * vv * vv / vv) * vv / vv
...

The kernel of the prover then produces the following messageskrt: sequence memory short
krt: asking for 1500000 int, waiting for system reply...
krt: OK, memory obtained, we continue.
krt: sequence memory short
krt: asking for 2249997 int, waiting for system reply...
krt: OK, memory obtained, we continue.
krt: sequence memory short
krt: asking for 3374992 int, waiting for system reply...
krt: OK, memory obtained, we continue.
...

The messages krt: sequence memory short are generated by the kernel when it dynamically claims additional memory.

This example is simple. Infinite loops may be caused by combination of rule applications and be more difficult to detect a priori.

C Guards of the theory language

This chapter presents guards (operators) that may be used to write rules. Such guards may be used to constrain the domain of application of a rule, using informations related to the goal and the hypothesis. To help writing and verifying the rules, all the guards of the theory language are documented thereafter.

A guard is a special antecedent in a rule. In fact, each guard in a rule is interpreted directly before the rule being effectively applied or not. Evaluating a guard never results in the creation of a SUCCESSor. The evalution of a guard may succeed or fail. For a rule to be effectively applicable, all the associated guard must be successful .

The following table summarizes the guard constructs:

band	conjunction of two guards
bfresh	construction of a fresh variable
bgetallhyp	obtains all the hypothesis
binhyp	tests the presence of a formula in (as ???) an hypothesis
blvar	lists the quantified variables
bmatch	identity by matching
bnot	negation of a guard
bnum	numerality test
bpattern	tests formula matching
bsearch	tests presence in a list
bstring	tests if is a character string
bsubfrm	finds sub-formulas
btest	numeric comparison
bvrb	variable test

band(g1,g2)

Parameters

g1 : garde
g2 : garde

Nature

Guard

Summary

Coordinates several guards.

Evaluation

The evaluation strategy differs according to the nature of the guards g1 and g2 :

- if g1 is binhyp(H), or binhyp(n,H), or binhyp(m,n,H), and if, in the last two cases, n is a wildcard, then the resulting guard succeeds if there exists an hypothesis h that matches H and that is such that g2 is successful
- if g1 is a bsubfrm, bsearch or brule guard, then the resulting guard is successful if there exists a wildcard instantiation such that g1 successful , and such that g2 is successful .
- in all other cases, the resulting guard is successful if both guards g1 and g2, evaluated in sequence are successful . If g2 fails, the evaluation does not backtrack to g1 as in the preceding cases.

Example

```
THEORY tentative IS
    band(binhyp(aaa(x)), band(binhyp(ccc),binhyp(bbb(x))) ) &
    bcall(WRITE: bwritef("x: %\n",x))
    =>
    H;
    bcall((DED~;ess):((bbb(1) => (aaa(2) & bbb(2) & ccc & aaa(1) => titi))))
    =>
    foobar
END
```

Result

```
x: 1
```

bnot(g)

Parameters

g : guard

Nature

Guard

Summary

Used to simplify expression of guards in case of failure.

Evaluation

The evaluation of bnot(g) is successful if the evaluation of g is a failure.

Example

```
THEORY tentative IS
        bcall(WRITE: bwritef("OK \n"))
    =>
        aaa(n);

        breade("Write ?>? (substituting the ? with numbers): ",a>b) &
        bnot(btest(a>b)) &
        bcall(WRITE: bwritef("Hello??? \n")) &
        aaa(n+1)
    =>
        aaa(n)/*(a,b)*/;

        bcall((ARI;ess)~: aaa(1))
    =>
        foobar
END
```

Result

```
Write ?>? (substituting the ? with numbers): 3>5
Hello???
Write ?>? (substituting the ? with numbers): 4>9
Hello???
Write ?>? (substituting the ? with numbers): 5>1
OK
```

bfresh(v, f, V)

Parameters

v: variable or variable list
f: formula
V: wildcard

Nature

Guard

Summary

To create fresh variables.

Evaluation

The evaluation is always successful . The wildcard V is instantiated with as many variables as there are in v. Moreover the new variables are not free in f. If v is not free in f, then V is the same as v.

Example

```
THEORY tentative IS
        bfresh((xx,yy,zz),aaa+xx$0-yy*zz,V) &
        bcall(WRITE: bwritef("%\n",V))
=>
        foobar
END
```

Result

```
xx$1,yy$1,zz$1
```

bgethyp(h), bgetallhyp(h)

Parameters

h : formula

Nature

Guard

Summary

This guard gets the hypothesis in a proof.

Evaluation

The evaluation of the guard is successful when the proof contains hypothesis and that the conjunction of these hypothesis matches with formula h. These hypothesis also depend on the type of guard:

- Guard bgethyp only yields the main hypothesis;
- Guard bgetallhyp yields the main hypothesis as well as the hypothesis derived from these main hypothesis.

In any case, all the wildcards of h are instantiated.

Example

```
HEORY tentative IS
        bgethyp(H) &
        bgetallhyp(G) &
        bcall(WRITE: bwritef("Main hypothesis: %\n",H)) &
        bcall(WRITE: bwritef("Main and derived hypothesis: %\n",G))
    =>
        P;

        bcall((DED;ess),fwd: (aaa & bbb => ggg))
    =>
        foobar
END

&

THEORY fwd IS
    aaa => ccc;
    ccc & bbb => ddd & eee
END
```

Result

```
Main hypothesis: aaa & bbb
Main and derived hypothesis: aaa & bbb & ccc & ddd & eee
```

binhyp(h), binhyp(n, h), binhyp(m, n, h)

Parameters

h: formula
n: formula
m: number

Nature

Guard

Summary

To access an hypothesis.

Evaluation

The guard binhyp(h) is successful when there exists an hypothesis that matches h. When there are several hypothesis, the last one is picked. All wildcards of h are instantiated.
The evaluation of the guard binhyp(n,h) depends on the nature of n:
When n is a number, the evaluation is successful when the hypothesis of rank n exists and matches hypothesis h. All wildcards of h are instantiated.
When n is a wildcard, the evaluation is successful if there exists an hypothesis that matches hypothesis h. n is then instantiated with the rank of h. All wildcards of h are instantiated.
In all other cases, the evaluation fails.
The evaluation of the guard binhyp(m,n,h) is similar to that of the guard binhyp(n,h), but in the case where n is a wildcard, the selected hypothesis is the last that matches h and with rank smaller or equal to ı m.
Note the similarities between these last two forms of the binhyp guard and the guards brule and lemma.

Example

```
THEORY tentative IS
    foo(i);

    binhyp(i,n,H) &
    bcall(WRITE: bwritef("hyp_%: %\n",n,H)) &
    foo(n-1)
    =>
    foo(i);

    binhyp(n,H) &
    bcall(WRITE: bwritef("hyp_%: %\n",n,H)) &
    foo(n-1)
    =>
    bar;

    binhyp(1,H) &
    bcall(WRITE: bwritef("hyp_1: %\n",H)) &
    bar
    =>
    baz;

    binhyp(H+G) &
    bcall(WRITE: bwritef("hyp: %\n",H+G)) &
    baz
    =>
    P;
```

```
      bcall((DED;(ARI;ess)~):((aaa & bbb+ccc & ddd) => pp))
      =>
      foobar
END
```

Result

```
hyp: bbb+ccc
hyp_1: aaa
hyp_3: ddd
hyp_2: bbb+ccc
hyp_1: aaa
```

blvar(l)

Parameters

l : list of variables

Nature

Guard

Summary

To get the list of quantified variables at the current rewrite position.

Evaluation

The guard is always successful .

If there is at least one quantified variable at the current rewrite position, l contains the list of quantified variables. Othewise l contains ?. For instance, if the current goal is

(aa,bb,cc).0<=aa & 0<=bb & 0<=cc => 0<=aa+bb+cc

and the rule

binhyp(x=0) &
blvar(Q) &
x\Q
=>
x == 0

is applied, then Q will be (aa,bb,cc).

This guard is used within rewrite rules. It provides the guarantee that one does not mix quantified with non-quantified variables. It is often used in conjunction with the absence of freedom guard bnfree x \ E.

? \ E is always true, whatever E is.

Example

```
THEORY tentative IS
    bnot(blvar(?))
    =>
    qq == pp;

    !qq.(qq+1>qq)
    =>
    %ii.(ii: NAT | uu) = oi$5;

    blvar(t$i)
    =>
    (t$i) == ii;

    blvar(Q) &
    Q\!yx.(yx<yx+oo) &
    bcall(WRITE: bwritef("free Q is %\n",Q)) &
    %(tt$0).(tt$0: NAT | uu) = oi$5
    =>
    !yx.(yx<yx+oo);

    blvar(Q) &
    Q\(aa+2) &
    bcall(WRITE:bwritef("aa transforms into oo\n"))
    =>
    aa == oo;

    blvar(Q) &
```

```
        Q\(yx+2) &
        bcall(WRITE:bwritef("yx transforms into za\n"))
        =>
        yx == za;

        !yx.(yx<yx+aa)
        =>
        !(zz$0,uu,hh$9).(zz$0+uu+hh$9>0);

        blvar(Q) &
        bcall(WRITE:bwritef("Q is %\n",Q))
        =>
        vv == hh;

        !(zz$0,uu,vv$9).(zz$0+uu+vv$9>0)
        =>
        !yy.(yy<yy+1);

        blvar(Q) &
        bcall(WRITE:bwritef("Q is %\n",Q))
        =>
        xx == yy;

        !xx.(xx < xx+1)
        =>
        foobar

END
```

Result

```
Q is xx
Q is xx
Q is xx
Q is zz$0,uu,vv$9
Q is zz$0,uu,vv$9
aa transforms into oo
free Q is ?
EXECUTION ABORTED ON GOAL: !pp.(pp+1>pp)
```

bmatch(x, p, q e)

Parameters

x: variable
p: formula
q: formula
e: formula

Nature

Guard

Summary

Checks if a quantified formula may be instantiated.

Evaluation

For the guard to be successful , it is first necessary that the formula p does not contain quantifiers. Then it is required that there exists a formula f such that the substitution of x by f in p is the same as formula q. Finally, formula f must match e. All wildcards in e are instantiated.

Example

```
THEORY tentative IS
	binhyp(!x.(H=>P)) &
	bmatch(x,P,Q,E) &
	bcall((SUB;WRITE):
	bwritef("P: %\nx: %\nE: %\n[x:=E]P: %\nQ: %\n",P,x,E,[x:=E]P,Q))
	=>
	Q;

	( !(xx$1,yy).(qq(xx$1,yy) => pp(xx$1,ff(xx$1),yy))
		=>
		pp(aa,ff(aa),bb)
	)
	=>
	ccc;

	bcall((ess;DED;ess):ccc)
	=>
	coco
END
```

Result

```
P: pp(xx$1,ff(xx$1),yy)
x: xx$1,yy
E: aa,bb
[x:=E]P: pp(aa,ff(aa),bb)
Q: pp(aa,ff(aa),bb)
```

bpattern(f, g)

Parameters

f: formula
g: formula

Nature

Guard

Summary

Tests if a formula matches another formula.

Evaluation

The guard is successful if formula f matches formula g. All wildcards in g are instantiated.

Example

```
THEORY tentative IS
        bnot(bpattern(p,q)) &
        bcall(WRITE: bwritef("FAILURE"))
        =>
        baz(p,q);

        bpattern(aaa+bbb,a+b) &
        bcall(WRITE: bwritef("% %\n",a,b)) &
        baz(aaa+bbb,k+l)
        =>
        foobar
END
```

Result

```
aaa bbb
FAILURE
```

bsearch(p, l, r) bsearch(p, l, r, s)

Parameters

p: formula
l : non-atomic formula
r: formula
s: formula

Nature

Guard

Summary

To search, and possibly modify an element in a list.

Evaluation

The formula l has the form l1 op ... op li op ... op ln where n is greater than or equal to 2 and op is a binary operator.
The guard bsearch(p,l,r) is successful when there exists a sub-formula li that matches p, and when the formula obtained by removing li from l matches formula r.
The guard bsearch(p,l,r,s) is successful when there exists a sub-formula li that matches p, and when the formula obtained by substituting li in l with the corresponding instance matches formula r.
All wildcards in p, r and s are instantiated.

Example

```
THEORY tentative IS
        bsearch((a-{x}),(aaa \/ (bbb-{xx}) \/ ccc \/ (ddd \/ {xx}) \/ eee),r,a) &
        bsearch((b\/{x}),r,s,b) &
        bsearch((b\/{x}),(aaa \/ (bbb-{xx}) \/ ccc \/ (ddd \/ {xx}) \/ eee),h) &
        bcall(WRITE: bwritef("r: %\na: %\nb: %\ns: %\nh: %\n",r,a,b,s,h))
        =>
        coco
END
```

Result

```
r: aaa\/bbb\/ccc\/(ddd\/{xx})\/eee
a: bbb
b: ddd
s: aaa\/bbb\/ccc\/ddd\/eee
h: aaa\/bbb-{xx}\/ccc\/eee
```

bstring(f)

Parameters

f: formula

Nature

Guard

Summary

To test that a formula is a character string.

Evaluation

The evaluation of the guard is successful if f is a character string (i.e. a sequence in characters between double-quotes). A double-quote may be present inside the string if it is preceded by a backslash.

Example

```
THEORY tentative IS

        bcall (WRITE: bwritef("test3: FAILURE\n"))
        =>
        test3;

        bstring(aa+bb);
        bcall (WRITE: bwritef("test3: SUCCESS\n"))
        =>
        test3;

        bcall (WRITE: bwritef("test2: FAILURE\n"))
        test3
        =>
        test2;

        bstring(aaa) &
        bcall (WRITE: bwritef("test2: SUCCESS\n")) &
        test3
        =>
        test2;

        bcall (WRITE: bwritef("test1: FAILURE\n")) &
        test2
        =>
        test1;

        bstring("Hello \"Sir\"") &
        bcall (WRITE: bwritef("test1: SUCCESS\n")) &
        test2
        =>
        test2;

        bcall(ess: test1)
        =>
        coco

END
```

Result

```
test1: SUCCESS
test2: FAILURE
test3: FAILURE
```

bsubfrm(g, d, p, q) bsubfrm(g, d, p, (q, v))

Parameters

g: formula
d: formula
p: formula
q: formula
v: formula

Nature

Guard

Summary

Tests the presence of a sub-formula in a formula and substitute it with another one.

Evaluation

For the evaluation of the first type of guard to be successful , it is first necessary that a formula f of p matches g (not instantiated). We then consider p obtained by substitution, in p, by the instantiated sub-formula f. Such formula p must match q (not instantiated). Most of the time, q is a simple wildcard.
In the second type of guard, the list of quantified variables on which f depends is also considered. If there is no such variable, then the list contains a single element: ?. This list must match v, not instantiated. Most of the time v is also a simple wildcard.
All remaining unmatched wildcards are instantiated.

Example

```
THEORY tentative IS
bsubfrm(x/:s,not(x:s),#(y,z).!(xxx$1,x,bbb).(x/:s => aaa),(q,v)) &
bcall((SUB;WRITE): bwritef("q: %\nv: %\n",q,v))
=>
coco
END
```

Result

```
q: #(y,z).!(xxx$1,x,bbb).(not(x: s) => aaa)
v: xxx$1,x,bbb,y,z
```

btest(m op n)

Parameters

m: formula
n: formula
op: comparison operator

Nature

Guard

Summary

To compare two numeric values.

Evaluation

The evaluation of the guard is successful when m and n related by the specified operator. The comparison operators:
– Equal: =
– Different: /=
– Smaller: <
– Smaller or equal: <=
– Greater: >
– Greater or equal: >=
When the operator is equality or difference, the evaluation of the guard is also successful when m and n are both identifiers that are related by the operator

Example

```
THEORY tentative IS
btest(bb/=aa) &
bcall(WRITE: bwritef("test3: SUCCESS\n"))
=>
test3;
bnot(btest(8=aa)) &
bcall(WRITE: bwritef("test2: FAILURE\n")) &
test3
=>
test2;
btest(8=8) &
bcall(WRITE: bwritef("test1: SUCCESS\n")) &
test2
=>
foobar
END
```

Result

```
test1: SUCCESS
test2: FAILURE
test3: SUCCESS
```

bvrb(f)

Parameters

f : formula

Nature

Guard

Summary

To test that a formula is a variable.

Evaluation

The evaluation is successful if f is a variable. Recall that a variable is either a letter (wildcard) or an identifier that do not start with an underscore, or one of the two previous possibilities followed by a $ and a number smaller than 10000, or a list composed of distinct elements that fall into the previous cases.

Example

```
THEORY tentative IS
    bcall(WRITE: bwritef("test5: FAILURE\n"))
    =>
    test5;
    bvrb(a+b) &
    bcall(WRITE: bwritef("test5: SUCCESS\n"))
    =>
    test5;
    bcall(WRITE: bwritef("test4: FAILURE\n")) &
    test5
    =>
    test4;
    bvrb(_a) &
    bcall(WRITE: bwritef("test4: SUCCESS\n")) &
    test5
    =>
    test4;
    bcall(WRITE: bwritef("test3: FAILURE\n")) &
    test4
    =>
    test3;
    bvrb(a$10000) &
    bcall(WRITE: bwritef("test3: SUCCESS\n")) & test4
    =>
    test3;
    bcall(WRITE: bwritef("test2: FAILURE\n")) &
    test3
    =>
    test2;
    bvrb(a,a,bbb) &
    bcall(WRITE: bwritef("test2: SUCCESS\n")) &
    test3
    =>
    test2;
    bcall(WRITE: bwritef("test1: FAILURE\n")) &
    test2
    =>
    test1;
    bvrb(aaaa_3,xxx,xx$2,y) &
```

```
        bcall(WRITE: bwritef("test1: SUCCESS\n")) &
        test2
        =>
        test1;
        bcall(ess: test1)
        =>
        foobar
END
```

Result

```
test1: SUCCESS
test2: FAILURE
test3: FAILURE
test4: FAILURE
test5: FAILURE
```