Johann Radon Institute for
Computational and Applied Mathematics
Austrian Academy of Sciences (ÖAW)

A Predicate Calculus Prover Based on Completion and Boolean Term Algebra

S.D. Mechveliani

RICAM-Report 2006-16

A Predicate Calculus Prover Based on Completion and Boolean Term Algebra.

A technical report for the Special Gröbner Semester, RICAM, Linz, 2006.

S.D.Mechveliani
Program Systems Institute, Pereslavl-Zalessky, Russia,
e-mail: mechvel@botik.ru

Abstract

In 1985 J.Hsiang introduced a certain generalized completion method with Boolean terms (**BT**), which expressed a refutational proof search for predicate calculus formulae. We revise and adopt this method by 1) bringing to it the *unfailing completion* version, 2) analysing and applying different methods for reducing BT, 3) adding heuristics for selecting terms in the proof by considering 'cases'.

Key words:
automated proof search, term rewriting, unfailing completion, Boolean term.

1 Acknowledgements

Half of this work has been done in March — April 2006, during the Special Semester on Gröbner Bases, organized by RICAM, Austrian Academy of Sciences, Johannes Kepler University.

The author is grateful to the supporters of his travel to Linz and of his work in Linz:
Russian Foundation for Basic Research (grant 06-01-10519),
organizers and sponsors of the Special Gröbner Semester.

This work is also a part of the 'Dumatel' project. And this project is supported in 2006 by the Program of Fundamental Research of the Russian Academy of Sciences Presidium ("Razrabotka fundamentalnykh osnov sozdaniya nauchnoj raspredelennoj informatcionno-vychislitelnoj sredy na osnove tekhnologij GRID").

2 Introduction

Concerning the generic problem of the proof search in the predicate calculus, we keep in mind that it is algorithmically unsolvable. So far, our aim is to create a program having, on average, about the same ability in solving problems as human. So far, we consider for the variety of problems only simply formulated problems in algebra or the ones concerning simple algorithms. This paper provides the three examples for such problems.

For this paper, our goal is:
to provide a correct extension of the *unfailing completion* method for many-sorted term rewriting (**TRW**) to *Boolean terms*, so that this would enable proofs in predicate calculus.

We use the abbreviation: BT — Boolean term (Zhegalkin polynomial).

Aiming at our goal, we continue the line presented by the following works.

(1) The completion method (KBC) by Knuth and Bendix of 1971 [KB].

(2) The refinement of this method to "unfailing completion" by [Hsi:Ru] (1987) and by [B:D:P] (1989).

(3) Certain optimized strategy design for unfailing completion described in [Lo:Hi] (2002).

(4) The extension of the KBC method to BT, with application to proofs in predicate calculus [Hsi] (1985).

Our main effort is in implementing the prover based on the methods (2) — (4), in optimizing the program. But also we search for optimization of these methods and improvements for their interaction.

The important points in our design are as follows.

1. The completion procedure design is optimized by introducing the flags (`Superposable | Unsuperposable`) (Section 3.4) and (`Preferable | Ordinary`) for equations (Section 3.6.2). This provides additional control for the user and in many cases reduces the search space.

2. The completion has additional stop condition: by exhaustion of the resource of 'steps'. This provides more flexibility when completion is applied as a part of the proof strategy, and prevents the prover from running into infinity along unlucky search branch.

3. The old KBC method in the RN+ strategy by [Hsi] is replaced by a certain variation of the unfailing completion algorithm by [Lo:Hi].

4. We apply a stronger method for the BT reduction (Section 3.6, Theorem from 3.6.1).

5. The method includes a strategy for the proof by completion and considering 'cases' for terms with a finitely enumerated value domain (Section 5). It applies certain heuristics for selecting the 'case' terms.

We have implemented this approach in the system Dumatel [Me] written in the Haskell functional language.

'Dumatel' is a joke Russian word taken from the novel "Skazka o troike" by brothers Strugatsky, it can be translated as 'thinker'.

This paper describes the prover's methods related to reduction of BT and proof by generalized completion with BT.

The structure of this paper is as follows.

The beginning of Section 3 describes the scheme of a proof by refutation: negation, converting to disjuncts, to BT, applying generalized completion. The description is done by introducing an example problem.

Section 3.1 explains the BT algebra.

Sections 3.2 and 3.3 give an introduction to the *unfailing completion* method.

Sections 3.4, 3.5, and 3.6 describe what is the generalized completion method (with extension to BT).

Section 3.6.1 suggests and motivates a new reduction method for BT, formulates the theorem about its properties. The proof is given in Appendix: Section 7.

Section 3.6.2 describes several optimizations, and some design features, in our variant of the completion method.

Section 3.7 comments the program print-out of performing the example problem from Section 3. This gives more detail about the completion (and proof search) strategy.

Section 5 explains the necessity for extending completion to a proof by considering 'cases' for terms with a finitely enumerated value domain. It gives the two example problems concerning algorithms for ordering lists.

In Section 6, we are trying to convince the reader and ourself that for the truth in variety, proofs by refutation include in a natural way 'positive' proofs.

3 Proof in predicate calculus by Boolean terms

To illustrate the usage of BT, consider the following example taken from [Hsi].

Let `G` be a group with the operations `e, *, i`, and `P` a non-empty subset in `G` such that if `X` and `Y` belong to `P` then `X*(i Y)` belongs to `P`.

Prove that it follows from these axioms that

(1) `e ∈ P`, **(2)** for all `X (X ∈ P ==> i X ∈ P)`

To solve this, we need to set the corresponding *calculus*, to input the goal formula, and to apply the proof procedure.

First, introduce a predicate symbol `P`, denote `x ∈ P` as `(P x)`, and create a calculus `groupP`, with the

sort `G`, operators `e, i, *, P`, variables `X, Y, Z`,

the three equations for the group laws,

equations for the Boolean connectives (like `x & true = true` and the such — but we do not show them here),

operator precedence `P >> i >> * >> ....`

This calculus is expressed as the following Haskell construct using the Dumatel library.

```
-------------------------------------------------------------------------------
module Main
where
import Dumatel

bool_lpo = bool defaultVariableWeight defaultInitialOpId weightDecls lpo
                                                          where
                                                          weightDecls = []
groupP =
  addDeclarations 0 CheckOn SkolemizeNot Ordinary (\ _ _ _ -> Superposable)
                  (superposabilityToCalculus Unsuperposable bool_lpo)
  $
  CalculusDeclaration
  {Sorts [G],                                      -- domain G <--> sort "G"
   Operators [e   : G,
              i_  : G -> G     [ParsePrecedences 0 7],
              _*_ : G G -> G   [ParsePrecedences 5 5],
              P_  : G -> Bool  [ParsePrecedences 0 6]
             ],
   OperatorPrecedence (P >> i >> * >> e >> false),
   Variables          [X Y Z :  G],

   Equations [ (X*Y)*Z = X*(Y*Z),  e*X = X,  (i X)*X = e ]
  }
-------------------------------------------------------------------------------
```

Let us comment this.

The line `bool_lpo = bool ...` forms the Dumatel library calculus for the Boolean connective operators. We do not display here this calculus.

The lines

```
groupP = addDeclarations ...
                         (superposabilityToCalculus Unsuperposable bool_lpo)
```

form the calculus `groupP` by adding to the calculus `bool_lpo` the declarations shown above.

A calculus also needs to set a function for an admissible (partial) term ordering ([Hsi:Ru], [B:D:P]), because this adds the notion of simplification related to the completion method. For the examples in this paper, it suffices the lexicographic-path ordering — `lpo` ([Hsi:Ru], [B:D:P]).

The calculus `bool_lpo` is built under the `lpo` ordering. The declaration for `groupP` does not mention any term ordering. This means that the ordering function is taken from the imported calculus `bool_lpo`, and it uses as a parameter the specified operator symbol precedence (`OperatorPrecedence` ...). The calculus `bool_lpo` has ist own operator precedence declaration. And the precedence declaration of `groupP` joins these ordering tables by inserting the declared segment before the operator `false` (from `bool_lpo`).

Currently, Dumatel is based on the *many-sorted TRW*: terms are formed according to the declared operator sortage and sorts of variables. The calculus `groupP` imports the sort `Bool` and declares the sort `G`, the latter expresses the needed algebraic domain.

We use here several 'sugar' denotations which are not yet implemented. For example, the declaration `_*_ : G G -> G` is set in the current Dumatel as

`Op " *  [G] [G] G ".` It declares a binary operator `*` with the result in `G`, with one argument of sort `G` writen on the input on the left, and one argument of sort `G` written on the right.

The goal formula can be set, for example, by parsing (`parse`) from a string and by reduction (`reduce`) by the calculus:

```
rF = reduce groupP $ parse groupP
     "
      (exist [X] (P X))                              &
      (forall [X,Y] (P X & P Y ==> P (X*(i Y))))
      ==>
      ( P e   &   (forall [X] (P X ==> P (i X))) )
     "
```

The prover denotes:
`"not"` — negation, `"&"` — conjunction, `"|"` — disjunction,
`"xor"` — exclusive disjunction, `"==>"` — implication.

In this example, the simplest way to start the proof search is to apply the function `proveByNegationAndCompletion`:

```
res = let rc = 10^6 in  proveByNegationAndCompletion ... rc groupP rF
```

By "..." we have denoted the arguments not material for this presentation. The argument `rc` defines a bound for the number of 'steps' to spend. It is useful when the function `proveByNegationAndCompletion` is applied as a part of some strategy. For this particular example, we could set it as `Infinity`.

The function `proveByNegationAndCompletion` forms the negation of the goal formula `rF` and skolemizes this negation. Then, it converts the skolemization to the *conjunctive normal form*. In our example, this produces the four *disjuncts*:

```
[ P XSk,                 not (P X) | not (P Y) | P (X*(i Y)),

  not (P e) | P XSk0,    not (P e) | not (P i XSk0)          ],
```

where the variable `X` and `Y` are presumed as universally quantified in each disjunct, `XSk` and `XSk0` are the Skolem constants.

Further, each disjunct converts to a certain Boolean term (BT) (BT are explained in the next subsection), with

`not x` converting to `x xor 1`, `true -> 1`, `false -> 0`,

and so on, the BT simplification rules applied ([Hsi] Sections 2, 3).

In our example, there appear the four BT made from disjuncts:

```
[(P XSk)  xor  1,                                    -- converts to equation
 (P (X*(i Y))) & (P Y) & (P X)  xor  (P Y) & (P X),
 (P XSk0) & (P e)  xor  (P e),
 (P i XSk0) & (P e)
]
```

In the further completion process the first of these BT converts to the equation `P XSk = true`. Such simplification always applies to BT of kind `(A &...& B) + 1`.

`"&"` stands for multiplication, `xor` stands for addition `"+"`.

A disjunct `d` has the meaning of the statement `d = 1 (= true)` — with the presumed universal quantifier. This statement is equivalent to `r(d) = 0 (= false)`, where the method `r` converts from disjunct to BT as follows:

```
r [] = 1,   r (l | d') = (l+1) & (r d').
```

Here `l` is a literal, and for negative `l = not l'`, `l+1` is put `l'`.

The Skolem operators (in our example they are constants) and the obtained BT join the calculus, producing a calculus `calc'`.

Then, the proof by refutation applies to `calc'` the generalized completion procedure `ukbb`.

Completion transforms the set of 'facts' (equations and BT) contained in the calculus, step by step, by joining at each step the set of simplest consequences from the current set of facts.

If this algorithm derives the BT `1`, this is equivalent to deriving `1 = 0`, or `true = false`, and this means to derive a contradiction — to obtain a refutational proof for the goal.

For such a refutational proof, the completion algorithm is given the goal equation `true = false`.

Either the algorithm derives this goal (by reducing its sides to the same term by the current set of facts)
or it stops by exhaustion of the resource of steps,
or it stops by forming a 'complete' set of facts.

Now, we need to explain what is a BT and what is the generalized completion.

3.1 Meaning of a Boolean term

A term is called ground if it does not contain variables.

An atom means in this paper a term having at the top a predicate symbol being not a standard Boolean connective.

We call a b-monomial a commutative product of several different atoms, with `"&"` for multiplication. It is actually a finite set of atoms.

Thus, in our example,
`(P e)` is a ground atom, `(P (X*(i X))` is a non-ground atom,
`(P (X*(i X)) & (P e))` is a b-monomial which is not an atom.

The predicate-top restriction:
for this paper, we presume that a b-monomial does not contain any (Boolean) variable as atom. Most examples work under this restriction. And in future, we have to consider lifting this restriction.

BT is a commutative sum of several different b-monomials
(a finite set of b-monomials). For example, the BT

```
(P (X*(i Y))) & (P Y) & (P X)  xor  (P Y) & (P X)
```

consists of two b-monomials and represents the statement

```
(P Y) & (P X)  ==>  P (X*(i Y))
```

(the universal quantifier is presumed).

The coefficients for b-monomials in BT are integers modulo 2.

Also it holds the idempotence law

`A & A = A` for any atom `A`.

The meaning of a BT `f` is the equation `f = false`, or `f = 0`, (compare this with a polynomial equation).

3.2 Term ordering, term reduction, equation superposition

The completion method for TRW was introduced in 1971 by D. Knuth and P. Bendix [KB]. The works [Hsi:Ru] (1987) and [B:D:P] (1989) give a remarkable improvement for this method: unfailing completion.

Here we consider only unfailing completion.

The completion algorithm processes the set of equations `s = t` by superposing and reducing equations as it is described, for example, in [Hsi:Ru].

Reduction of a term to the normal form by a set of equations is an important part of completion. In unfailing completion, equations are bi-directed: potentially, can apply in both directions. And reduction relies on a partial term ordering `">>"`: each equation applies to a term `t` only in such direction, and under such a match, for which the result is smaller than `t` in `">>"`.

In superposition [Hsi], only those unifiers are applied, which superposition result does not contradict the superposed equations under the ordering `">>"`.

Definition. A partial ordering `">"` is well-founded iff there does not exist any infinite chain `x > y >` ... in the domain set of this ordering.

A partial ordering `">>"` is an important parameter, it defines a notion of what expression is simpler. It can be set in many ways, but it needs to be a Simplification Ordering Total (linear) on Ground Terms [Hsi:Ru]. We abbreviate this as SOTG. Its laws are as follows.

(1) `">>"` is a well-founded.

(2) `c[t] >> t` for any proper subterm `t` in `c`.

(3) Monotonicity by the term structure: `s >> t ==> c[s] >> c[t]`
(for example, `u >> v  ==>  u+a >> v+a`).

(4) `s >> t ==> sb s >> sb t` for any substitution `sb`.

(5) It is a total ordering on the set of ground terms.

3.3 Unfailing completion

This is an algorithm ([Hsi:Ru], [B:D:P]) which transforms at each step the current set `E` of equations to the set `E'` by adding superpositions and by reducing equations by other equations. [Lo:Hi] describes an optimized strategy for this method. And we have implemented a certain variant of this strategy.

The completion algorithm takes among its arguments the set of goals: pairs `(s, t)`, expressing the equality `s = t`. When the current set `E` reduces this pair to trivial (the one of kind `(s', s')`), the corresponding goal is proved, and it is removed.

The completion stops when

the goal set is empty `or`

a complete equation set `E` is obtained (that is all its superpositions reduce to trivial by `E`) `or`

the bound `rc` for the number of 'steps' is reached (this is a particular feature of our version).

The works [Hsi:Ru] and [B:D:P] prove the logical completeness of the unfailing completion method and its other remarkable properties.

3.4 Generalized completion

We call such the completion procedure extension for BT. Concernig the proof search problem, it is natural to recall first of

(1) the resolution method [Ro], which processes disjuncts only.

(2) One could apply directly the unfailing completion to disjuncts, or to BT, by treating the boolean connectives as other operators, and by treating a BT as an equation `first monomial = remainder`. Adding the AC unification for `"&"` and `xor` will increase performance, but even without it the unfailing completion is complete.

This approach leads to a greater computational cost than the method RN+ considered below.

In the work [Hsi] (1985), it is described the refutation search strategy RN+, which joins the old KBC method [KB] (for equations) and a special way of superposing BT: BN-critical pairs. It puts a restriction: one of BT in the superposed BT pair must be a b-monomial. And it provides a certain proof scheme for that such a strategy is complete.

It follows then, that any superpositions of a b-monomial `m` with a BT `f` is possible only by superposing of `m` with each monomial in `f` separately.

All this allows to reduce the proof search space.

We prefer the generalized completion (with BT) for the following reasons.

1. It looks like a natural extension of the completion method of TRW. In particular, it uses the term ordering and b-monomial ordering in a fashion close to TRW and computer algebra.

2. For ground terms, this method yields the graded polynomial algebra, even with applicable Gröbner basis method.

The generalized completion processes 'facts': equations and BT.

BT are being *reduced* by equations and by BT (by the function `reduceBT`).

Equation superposes with equations and with BT.

A monomial BT superposes with any BT at various monomials.

The superposability flag for the standard Boolean equations
(like `x & true = true` and such).
In our prover, the standard Boolean equations apply in reduction, but usually are marked as `Unsuperposable`. This is because a) the generalized completion treats the Boolean algebra in a special way, preserving the method completeness, b) the less superpositions are allowed (with preserving completeness) the smaller is the search space.

3.5 Superpositions

They are are of kind "ee", "be", "bb".

We call "ee" a superposition of equations. It is of unfailing completion ([Hsi:Ru], [B:D:P]).

We call "be" a superposition of an equation `s = t` with a BT `f`. It is done by unifying of `s` or `t` with any atom subterm in any monomial in BT and by forming the result BT in an evident way.

A "bb" superposition superposes a monomial `m` with a BT `f` by 'mul-unifying' `m` with some monomial in `f`.

A monomial-to-monomial (mul-)unification is by unifying an atom subset and by applying the complementary factors
(compare this to LCM for polynomial power products).

This is the AC+idempotence unification for the operator `"&"`.
It finds a complete set of mul-unifiers.

Example

For our example with `groupP`, suppose that the completion deals with the BT (1) and (2), as below. Then, the superposition is as follows:

```
(1)  (P Y) & (P i Y) & (P e)  xor  (P Y) & (P e)  |
(2)  (P i XSk0) & (P e)                           |   -->  (P XSk0) & (P e).
```

Because the substitution `[Y := XSk0]` produces the instance `(1')` of BT `(1)`:

```
(1')  (P XSk0)&(P i XSk0)&(P e)  xor  (P XSk0)&(P e).
```

And it also unifies (2) with a submonomial of the head monomial in `(1')`. Now, similar as with polynomials, the submonomial after this substitution is replaced with `0`, and there remains from `(1')` the monomial `(P XSk0)&(P e)`. This is the result of superposition. Then, it is possible reduction: the BT `(P XSk0)&(P e) xor (P e)` from the initial basis *reduces* the result to `(P e)`. This reduction is due to the particular assumed monomial ordering described below.
(compare this reduction to polynomial reduction).

In the general case, there may be needed the complementary factors.
(compare this to s-polynomial of Gröbner bases [Bu]).

What is a refutation procedure:
by superposing 'facts', and by reducing them by each other, try to obtain the unity BT.

The method scheme we have sketched, is a certain version of the `RN+` strategy by [Hsi].

3.6 Reduction for BT

The 'facts' are being reduced by each other. The more is reduced the current fact the smaller search space remains.

The "ee" reduction is as in the unfailing completion [Hsi:Ru].

The "be" reduction is through reduction of each atom by equations.

For the BT reduction, [Hsi] suggests only reduction by b-monomials. With this approach, to reduce a BT `f` by a b-monomial `m` means actually to remove from `f` all monomials `n` which contain any instance of `m`. This restriction may look reasonable, because a) it is clearly a 'reduction', b) there does not exist such a well-agreed total reduction order on b-monomials as it is for polynomials.

Unfortunately, this reduction is rather weak. For example, any set `gs = [f, g]` of binomial BT cannot reduce `f`, despite that `f` belongs to `gs`.

We introduce a stronger reduction, as described below. Let us call it BD-reduction.

3.6.1 BD-reduction for BT

It is based on a special mls-extension of the (partial) term ordering `">>"` to b-monomials. The latter is extended further to the (partial) ordering on BT. And we denote these three orderings with the same symbol `">>"`.

For a BT set `gs`, reduction of a BT `f` by `gs` is presented by the function `reduceBT`. At each step of the `reduceBT` loop, it is applied the reduction `f' = reduce1 g f` to reduce the current `f` by a single BT `g`. If there is zero among the set of such `f'`, then `reduceBT` stops with the zero result. Otherwise, the reduction recurses by `reduceBT gs f'` to the first `f'` such that `f >> f'` — if such exists. If such does not exist, the result is put `f`.

`reduce1 g f` performs by considering BT of kind

```
f' =  f - (q & (sb g)),
```

with all the 'suitable' substitutions `sb`, and the corresponding monomial factor `q`.

We note also that the subtraction `"-"` coincides for BT with the operation `xor`.

A 'suitable' substitutions `sb` is any *matching* of any monomial `m` from `g` against any subset in any monomial `n` in `f`. For a suitable substitution `sb`, and its monomial pair `(m, n)`, the complementary factor `q` is defined by the condition `n = q & (sb m)`.

All these items are found — modulo associativity, commutativity and idempotence of the operation `&`. It is evident that the set of the above suitable triplets `(m, n, sb)` is finite.

The procedure `reduce1` searches this set through, and if any of `f'` in this set is zero, it is put as the result. Otherwise, it returns the first `f'` such that `f >> f'`, if such exists. If such does not exist, the result is `f`.

Clearly, the usefulness of the BD-reduction depends on the ordering extension to b-monomials and to BT.

Set the *multiset ordering* extension for b-monomials
— let us call it 'mls' and denote it again as `">>"`.

Applying the multiset order definition from [B:D:P] (Section 5), to b-monomials, we have the

Definition. `">>"` is the transitive closure of the relation
`mon & x >preMls> mon & mon'` iff `x >> y` for all `y` in `mon'`,
`x` is not in `mon`, `mon` ∩ `mon' = []`.

`>preMls>` means that any atom `a` in a b-monomial `m` can be replaced with any finite set (empty too) of atoms smaller than `a`, and this is the only way to obtain a b-monomial smaller in `>preMsl>` than `a`.

Evidently, the relation `">>"` on b-monomials can be computed by the algorithm of the following rules.

```
mon >> mon' =  gtm (mon \\ mon') (mon' \\ mon)         -- cancel intersection
  where
  gtm _  [] =  True    -- case of a proper subset
  gtm [] _  =  False   --
  gtm m  n  =  exist [a <- m, b <- n] such that  a >> b  and  m >> (n \\ [b])
```

Lemma (Mls) (another reformulation of mls).
For b-monomials `m, n`, denote `m' = m \ n, n' = n \ m`.
Then, for the mls extension of `">>"` to b-monomials, `m >> n` if and only if (`m'` is non-empty, and for each `b` in `n'` there exists `a` in `m'` such that `a >> b`).

The proof easily follows from the definition of mls.

Theorem.
The multiset extension of any partial ordering is a well-founded partial ordering.

According to [B:D:P], this theorem is proved in [D:Man]. We believe that we have also a proof of our own, but we skip it here.

Analogue example. Compare this well-foundedness property to the corresponding property of the graded polynomial algebra over `Integer`. If we compare polynomials by degree, and extend this comparison to coefficients by comparing absolute values, then, in the result ordering, the set `{g | f >> g}` is always finite.

(also note that a polynomial has not variables, it has indeterminate constants, which we do not consider as variables).

In the following theorem,
`m, n, p` denote b-monomials,
∩ denotes the intersection of b-monomials (as sets of atoms),
`"\"` denotes the difference of b-monomials (as sets of atoms).

Theorem (Mls).
The mls-ordering for b-monomials satisfies the following properties.

(Ms1) On ground b-monomials, mls is a total ordering which coincides with the lexicographic extension of `">>"` from ground atoms.

(Ms2) Any non-identical reduction of any atom in a b-monomial by equations makes this b-monomial smaller.

(Ms3) If `m >> n`, then for any `p` `p & m >> n`.

(Ms4) Invariantness under regular factors:
if `m >> n` and `p ∩ (m \ n) = []`, then `p & m >> p & n`.

(Ms5) Invariantness under substitutions:
if `m >> n`, then for any substitution σ σ `m >>` σ `n`.

The proof for this is easy, still it is provided in Section 7.

As BT is a finite set of b-monomials, the mls-extension of the ordering `">>"` to BT yields a BT ordering with the properties similar to (Ms1) — (Ms5). This makes the BD-reduction both correct and more efficient for completion than the reduction "by monomials only". In particular,
a) the BD-reduction is well-founded (terminates),
b) the reduction by equations and reduction by BT agree: simplify in the same direction,
c) the simplification is not only by monomials, and if a BT `f` is an instance of one of the elements of `gs`, then `gs` reduces `f` to zero.

More example: suppose that some b-monomial `m` in `g` is a multiple of each remaining monomial in `g` (this is so, for example, in a BT representing implication). Suppose also that a substitution `sb` matches `m` against a subset in some monomial `n` in a BT `f`, with the complementary factor `q`.

Then, `f - (q & (sb g))` is a non-trivial reduction by `g`.

3.6.2 Optimizations for reduction and fact selection

1. The mls comparison is expensive and is intensively applied. Therefore we apply the optimization of the "set of maxima". Namely, for each BT from the basis `gs` the BD-reduction finds preliminarily its set `maxMons` of maximal monomials by `">>"`. In the loop of reduction by `gs`, the matches `sb` are searched only for the monomials from `maxMons`.

There are possible other optimizations.

2. 'Dumatel' allows to mark an equation as `Superposable` or `Unsuperposable`.
The explanation for this is given earlier. Thus, in our example `groupP`, the calculus includes the equations from the calculus `bool_lpo` by using the construct of kind

```
addDeclarations ... (superposabilityToCalculus Unsuperposable bool_lpo)
```

This imports the standard Boolean equations with marking them as `Unsuperposable`.

3. Each fact has the mark of `Preferable | Ordinary`.

The completion algorithm selects the current fact for processing by considering its (a) syntactic size, (b) `Preferable` mark, (c) orderedness of the equation parts (for an equation fact).

The condition (c) for an equation `s = t` means `s >> t`. Such an equation is called directed, and also a rewrite *rule*.

A directed equation is selected more eagerly than un-directed (under other equal conditions). This increases, — on average, — the reduction power of the currently active set of facts. Also this reduces, — on average, — the set of the current superpositions, because the unfailing completion puts a certain restriction for a superposition not to break the ordering of the equation instance.

A `Preferable` fact is selected more eagerly than `Ordinary` — under other equal conditions. For example, for a proof by contradiction of a statement `fF`, Dumatel marks the equations and BT originated from the negation of `fF` as `Preferable`. Due to this, the refutation search occurs more directed — on average.

Also the `Preferable` mark can be set by the user. This has a meaning of a hint for the system of the approximate direction of the proof search.

3.7 Performing the example

In the example `groupP`, the function `proveByNegationAndCompletion` yields the result `res` of the proof search by refuting the negation by the generalized completion. This result contains many data fields. In particular, the trace of the proof search can be extracted by applying

```
ukbbTrace $ cpRefInitialCompletion res.
```

From the below print-out of this trace we see, in main, how the completion algorithm works. Its main scheme is by [Lo:Hi].

The initial facts are: the equations from `groupP` and equations and BT originated from negating and skolemizing of the goal formula.

The initial facts are put in the *delayed set* ("passive facts") `pP`.

'Active' facts are the ones by which the current reduction is done, and which are superposed with the selected fact. The set of active facts is initiated as empty.

Each 'step' of the completion loop is as follows. It selects the 'simplest' fact from `pP`, reducing it by the active facts to `f`, and removing trivial facts, when such appear. Active facts reducible by `f` move to `pP`. All the superpositions of `f` with the active facts are being reduced by active facts, and the reduction results insert into `pP`.

Here are some explanations.

The print-out is a sequence of print-outs of 'steps', embraced by parentheses and separated by comma. In this print-out, we use the following abbreviations:

```
Sup  --- Superposable,  Unsup  --- Unsuperposable,
Od   --- Ordinary,      Pref   --- Preferable.
```

The labels in 'facts' are printed in the square brackets. Labels are used only for referencing facts on the print-out, in order to reduce the text volume.

`(P [l1, l2])` means that the considered fact is obtained by superposing the facts referenced by `l1` and `l2`; `'P'` abbreviates the word `Parents`. For example, the line

```
(... [50] (i X)*(X*Z) -> Z     Sup Od (P [[26], [24]]))
```

means that this fact is labelled as `[50]`, and that it is obtained by superposing of the facts `[26]` and `[24]`.

`(P [])` means that the considered fact is taken from the source data.

Reduction preserves labels.

"`Chosen e`" means: it is chosen an equation from `pP`, and the result of its reduction is printed further in the line. "`Chosen b`" means the same for selecting BT.

The part of a line after '`--`' is our comment. Some of our comments show in additional lines more details on how the current fact is obtained.

Directed equations are printed as `s -> t` (in reduction, a directed equation can apply only "from left to right").

About proof reduction

After the prover finds a proof, it could try to reduce it to a shorter proof. But the problem of such reduction is not so simple as it looks at the first glance. We do not address it, so far.

```
[(Pre-reduction in completion. The number of rules is  26.
The number of equations and BT
in the initial calculus are respectively  31, 5;  after reduction: 31, 10.
The reduced goals are                     ([(true, false)], [])
),
(Chosen e  [29] x ~ x -> true           Unsup Od (P [])),
(Chosen e  [30] true ~ false -> false   Unsup Od (P [])),
(Chosen e  [31] false ~ true -> false   Unsup Od (P [])),
(Chosen b  [42] (P i XSk)&(P e)         Pref (P [])),
(Chosen e  [28] P XSk0 -> true          Sup Pref (P [])),
(Chosen e  [25] e*X -> X                Sup Od (P [])),
(Chosen e  [24] (i X)*X -> e            Sup Od (P [])),
(Chosen e  [26] (X*Y)*Z -> X*(Y*Z)      Sup Od (P [])),
(Chosen e  [50] (i X)*(X*Z) -> Z        Sup Od (P [[26], [24]])), -- first derivation
(Chosen e  [51] (i e)*X -> X            Sup Od (P [[50], [25]])),
(Chosen e  [55] (i i e)*X -> X          Sup Od (P [[51], [50]])),
(Chosen e  [57] i e -> e                Sup Od (P [[55], [24]])
),
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:         [[51], [55]]
),
(Chosen e  [53] (i i X)*e -> X            Sup Od (P [[50], [24]])),
(Chosen b  [41] (P XSk)&(P e) xor (P e)   Pref (P [])),
(Chosen e  [58] (i i i X)*X -> e          Sup Od (P [[53], [50]])),
(Chosen e  [60] i i X -> X                Sup Od (P [[58], [50]])
),
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:        [[53], [58]]
),
(Chosen e  [53] X*e -> X                Sup Od (P [[50], [24]])),
(Chosen e  [63] X*(i X) -> e            Sup Od (P [[60], [24]])),
(Chosen e  [62] X*((i X)*Z) -> Z        Sup Od (P [[60], [50]])),
(Chosen e  [65] X*(Y*(i (X*Y))) -> e    Sup Od (P [[63], [26]])),
(Chosen e  [67] Y*(i ((i X)*Y)) -> X    Sup Od (P [[65], [62]])),
(Chosen e  [74] Y*(i (X*Y)) -> i X      Sup Od (P [[67], [60]])),
```

```
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:         [[67], [65]]
),
(Chosen e [84] (i (X*Y))*X -> i Y      Sup Od (P [[74], [74]])),
(Chosen e [82] i (X*(i Y)) -> Y*(i X)  Sup Od (P [[74], [62]])),
(Chosen e [83] i (X*Y) -> (i Y)*(i X)  Sup Od (P [[74], [50]])
),
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:         [[74], [84], [82]]
),
(Chosen b  [40] (P (X*(i Y)))&(P Y)&(P X) xor (P Y)&(P X)   Pref (P [])),

-- recall  [63] X*(i X) -> e,  apply  Y := X,  and derive
--
(Chosen b  [106] (P X0)&(P e) xor (P X0)          Pref (P [[63], [40]])),

(The following facts have been newly reduced,
or subsumed, and moved to delayed set:        [[42], [44], [41], [43]]
),
(Chosen e  [44] P i XSk01 -> false      Sup Pref (P [])),
(Chosen e  [42] P i XSk -> false        Sup Pref (P [])),

-- it is given  [28]  P XSk0 -> true,
-- recall       [106] (P X0)&(P e)  xor  (P X0),   apply X0 := XSk0  and derive
--
(Chosen e  [111] P e -> true            Sup Pref (P [[28], [106]])  -- goal 1
),
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:        [[106]]
),
(Chosen e  [43] P XSk01 -> true           Sup Pref (P [])),
(Chosen e  [41] P XSk -> true             Sup Pref (P [])),

-- recall  [111] P e -> true,                        put  X := e  in the given
--         [40] (P (X*(i Y)))&(P Y)&(P X)  xor  (P Y)&(P X)         and derive
--
(Chosen b  [117] (P i Y)&(P Y) xor (P Y)   Pref (P [[111], [40]])),

-- recall  [41] P XSk -> true,  put  Y := XSk  in [117],  derive
--                                                          (P i XSk) xor 1,
-- and reduce it by [42] to
--
(Chosen b  [128] 1           Pref (P [[41], [117]])   -- contradiction derived
),
(The following facts have been newly reduced,
or subsumed, and moved to delayed set:        [[117], [40]]),
(Chosen e  [128] false -> true               Sup Pref (P [[41], [117]])),
The number of goals has reduced to  0
]
```

4 Equality predicate

Each *sort* in a calculus in Dumatel is supposed to be provably non-empty. For example, it is sufficient to declare any constant in each sort.

And usually, for each sort, the calculus needs to have the equality operator `"~"` for this sort. This operator must be accompanied, at least, by the equation `x ~ x = true`. Also the prover incorporates the inference rule `s ~ t = true |- s = t`. This all gives the prover possibility to reason adequately about equality/in-equality in algebraic domains. For example, the main *field* axiom is written as

```
forall x (not (x ~ 0) ==>  x*(inverse x) ~ 1)
```

We often write briefly `"~"`, while in the real program input it is denoted as different operators `eq_<S>` for each sort `S`.

5 Proof by completion and considering 'cases'

When searching a proof, a prover often deals with terms who's sort is finitely enumerated. For example, any term of kind Bool can have either the value `true` or `false`. Generally, let some domain `D` contain only the values `a_1, ..., a_n`. The simplest way to express this knowledge is to add to the calculus the disjunct `X ~ a_1 | ... | X ~ a_n` consisting of several equality formulae.

The prover converts the above disjunct to BT, and this BT takes part in completion.

But for practice, our prover achieves a cheaper computation by introducing the *finite enumeration* denotation to the data construct of `Calculus`. For example, `bool_lpo` includes the declaration like

```
FiniteEnumeration Bool [true, false].
```

Meaning: for a sort `S` declared as enumerated by a finite set `En` of ground terms, any proof "in variety" is understood only for the calculus models in which the support for `S` is a subset of the value set for `En`.

So far, our prover allows only constants in the finite enumeration construct.

The finite enumeration construct provides correctness for a proof by splitting to *cases*. This kind of search strategy selects several terms `e` for each finitely enumerated sort. Let us call the selected terms *'case' terms*.

Then, each case is defined by the finite set $\{e_1 = c_1, \ldots, e_n = c_n\}$ of equations, where e_i are the 'case' terms, c_i all the enumeration constants of the value domain of e_i. This set of equations adds automatically to the calculus, and the prover searches for the proof (refutation). When it is refuted a logically complete subset of cases, the refutation is done.

For searching a refutation by completion, the prover spends half of the resource to the refutation by completion only (without 'cases'). It this part fails, it starts the loop of completion and considering cases, giving it the remaining resource.

The prover applies a certain strategy to consider sub-cases and to select a new 'case' term. It distributes the resource so that there are considered several cases for the most important 'case' terms, and there remains no resource for the rest cases. The 'case'

terms are selected automatically, by heuristics. The strategy must consider only the cases for a small set of selected terms (for the rest cases the resource is out). This is because the total number of cases is often unfeasible. For example, if the enumeration for each e_i has d constants, then the necessary number of cases may occur d^n for the number n of considered 'case' terms.

'Dumatel' extracts the case-terms out of the last failed proof attempt. A result of the current refutation by completion attempt contains some set of ground terms, containing the Skolem operators. Such terms, — only of finitely enumerated sorts, — are considered in the first turn for 'cases'. Also there is set the preference for them according to their syntactic relation to the calculus equations.

5.1 Example 1. A simple problem for ordered lists

The following specification defines the calculus for lists (of the domain `List`) over any domain `Elt`. This calculus defines, among others, the operators `">"` — for ordering on `Elt`, `isOrdered` — for testing the orderedness of a list, `insert` — for inserting an element to list according to the element order. Naturally, these operators are provided with certain simple laws: rules and equations, which sets can also be treated as programs.

Then, there follows the function `main`, which presents the proof search for the formula

```
forall [X,Y,Z] (isOrdered (insert Z (insert Y (X : nil))))
```

```
------------------------------------------------------------------------------
module Main
where
import Dumatel

bool_lpo = bool defaultVariableWeight defaultInitialOpId weightDecls lpo
                                                          where
                                                          weightDecls = []
(list, eqsFromFormulae, _, _) =

  addFormulae 0 preList SkolemizeNot Ordinary (\ _ _ _ -> Sup) formulae
  where
  formulae =
     map (parseSingleOrBreak preList "list calculus")
     [
      "forall [X,Y] (X eq_Elt Y | X > Y | Y > X)",
      "forall [X,Y] (X > Y ==> not (Y > X))",       -- antisymmetry
      "forall [X,Y] (X > Y ==> not (X eq_Elt Y))"
                                                    -- we skip transitivity
     ]

  preList =
    (\ calc -> addEquations calc $ rulesFromCalculusToEquations calc)
    $
    addDeclarations 0 CheckOn SkolemizeNot Ordinary (\ _ _ _ -> Sup)
                    (superposabilityToCalculus Unsuperposable bool_lpo)
    $
    CalculusDeclaration
```

```
{
 Sorts ["Elt",      -- sort for the list elements
        "List"
       ],
 SortGen "List" [["nil", ":"]],  -- List is generated by  nil and ":"
 Operators
  [nil   :  List,
   _:_   :  Elt  List -> List  [ParsePreceds 5 5],   -- prepend
   _+_   :  List List -> List  [ParsePreceds 5 5],   -- concatenation
   _and_ :  Bool Bool -> Bool  [ParsePreceds 5 5],
   _>_   :  Elt  Elt  -> Bool  [eqPPreceds],

   a, b, c :  Elt,    -- some element constants to make examples

   _eq_Elt_  : Elt  Elt  -> Bool  [eqPPreceds, OpAlgAttrs [Commutative]],
   _eq_List_ : List List -> Bool  [eqPPreceds, OpAlgAttrs [Commutative]],

   insert    : Elt List -> List           [ParsePreceds 0 7],
   ins       : Elt Elt List Bool -> List  [ParsePreceds 0 7],
   isOrdered : List -> Bool               [ParsePreceds 0 7],
   isOrd     : Bool Bool -> Bool          [ParsePreceds 0 7]
  ],
 OpetatorPrecedence
                [insert >> ins >> isOrdered >> isOrd >> eq_List >> + >>
                 : >> nil >> ">>" >> eq_Elt >> and >> a >> b >> c >> false
                ],
 TermComparison (Just lpo),

 Variables [X Y Z X' Y' :  Elt,
            Xs Ys Zs    :  List",    bo : Bool
           ],
 FormulaLaws [],
 Eqauations  [],  -- they are automatically made later, from rules
 Rules
   [X eq_Elt X -> true,
    a eq_Elt b -> false,   a eq_Elt c -> false,
    b eq_Elt a -> false,   b eq_Elt c -> false,
    c eq_Elt a -> false,   c eq_Elt b -> false,

    Xs eq_List Xs -> true,

    false and x     -> false,    x and false -> false,
    true  and x     -> x,        x and true  -> x,

    nil    + Ys = Ys,
    (X:Xs) + Ys = X : (Xs + Ys),
    ----------------------------------------
    X > X -> false,
    a > b -> false,   a > c -> false,
    b > a -> true,    b > c -> false,
    c > a -> true,    c > b -> true,
    ------------------------------------------------------------
    isOrdered nil      -> true,
    isOrdered (X:nil)  -> true,
    isOrdered (X:Y:Ys) -> isOrd (X > Y) (isOrdered (Y:Ys)),
```

```
          isOrd true  bo -> false,
          isOrd false bo -> bo,

        insert X nil      -> X : nil,
        insert X (Y : Xs) -> ins X Y Xs (X > Y),

        ins X Y Xs true  ->  Y : (insert X Xs),
        ins X Y Xs false ->  X : Y : Xs
       ]
    }
    where
    eqPPreceds = ParsePreceds l r  where
                                 (l, r) = defaultEqualityOperatorParsePrecedences

main =
  putStr $ concat
  [
   "Goal formula =  ",    show fF,
   "\n\n",
   "is proved = ",        show $ isProved res,
   "\n",
   "rcRem = ",            show $ cpRefRcRem res,
   "\n",
   "refuted cases =\n",   showsList ",\n" (cpRefRefutedCases res)
   "\n\n",
   "remaining cases =\n", showsList ",\n" (cpRefRemCases res)
   "\n"
  ]
  where
  parse1 = parseSingleOrBreak th "list calculus"
  th     = list

  superpty _ _ _ = Superposable
  jump           = 2

  fF = parse1 " forall [X,Y,Z] (isOrdered insert Z insert Y (X : nil)) "

  rc              = 4*10^5
  rcInitialCompl  = Fin (10^3)
  rcPerSubcase rc = quot rc 64

  res = proveByNegationAndCompletion
        Trace jump (Fin rc) rcInitialCompl (rcPerSubcase rc)
        (Skolemize defaultSkolemParameters) Preferable superpty th fF
```

Comment: the *predicate calculus formula laws* for the operator `">"` (as, for example, anti-symmetry) join the calculus by applying the function `addFormulae`. In this example, this leads to converting these formulae to BT and joining these BT to the BT part of the calculus. Also, as we shall see in the print-out, these three formula laws reduce the set of cases to consider. Because for the indefinite constants `(XSk, YSk, ZSk)` $= (c_1, c_2, c_3)$ appearing in the completion process, these laws put certain evident logical dependencies between the values of
$\{c_i > c_j,\ c_j > c_i,\ c_i \tilde{} c_j \mid i, j \in \{1, 2, 3\}\}$.

In this example, the prover searches for the proof as follows. It adds to the calculus the skolemization of the negated goal statement and tries to refute the obtained calculus `list'` by completion. First, it applies completion only. And this attempt fails by exhausting its part of resource. Its result contains certain terms in which it occurs the subterm `(YSk > XSk)`, where `YSk` and `XSk` are the Skolem constants for certain elements of sort `Elt`. They are indefinite, so there it is not known their order under the ordering of `">"`. With ignoring the `FiniteEnumeration` construct, the prover does not 'understand' that the term `(YSk > XSk)` can equal either `true` or `false` (!). On the other hand, it depends on the above term value (`true` or `false`) how there will simplify the terms `(insert YSk (XSk : nil))` and `(isOrdered insert YSk (XSK : nil))`.

This is why the first refutation attempt has failed.

Then, the prover spends the resource remainder to considering 'cases'. According to its heuristics, the most perspective term for 'cases' occurs `(YSk > XSk)`. Again, the refutation by considering cases for this term fails by exhausting its part of resource. The result contains the subterm `(ZSk > XSk)`. Again, it depends on its value (`true` or `false`) how there will simplify the terms `(insert ZSk insert YSk (XSk : nil))` and `(isOrdered insert ZSk insert YSk (XSK : nil))`.

The chosen 'case' term `(ZSk > XSk)` joins the vector for 'cases', and the refutation search continues with the remaining resource.

In this way, the prover comes to the triple `[YSk > XSk, ZSk > XSk, ZSk > YSk]`. The case set for this triple is sufficient to simplify the goal terms to the point of contradiction.

The following print-out shows in a more detail how the sub-cases are processed.

```
------------------------------------------------------------------------
Pre-reduction done. Initial numbers of equations and BTs are
57 and 3,  after reduction:  57 and 6.

Completion stops by resource exhaustion.

Starting final mutual reduction for UKBBResult.
The number of rules is  56.
Numbers of equations and BTs before reduction are  23 and 0,
after reduction:                                   23 and 0.
Non-trivial jump reduxes are  [].

Initial refutation by completion has failed.
But there are some 'cases' to consider for ground terms.
Starting to refute by cases.

-----------------------------------------------
refuteByCompletionAndCasesForGroundTerms Trace 2 superpty 6250 calc state,
rc = (Fin 396465)

enumPairs =  [(YSk > XSk,  [true, false])]

refuteCases:  refuting the case  [(YSk > XSk, true)].
Resource per subcase is  6250,
the total remaining resource is  Fin 396465.
```

```
Failed to refute the subcase. Additional support =  {(ZSk > XSk)}.
Failed to refute the subcase. Additional support =  {(ZSk > YSk)}.

The case of   [(YSk > XSk, true), (ZSk > XSk, true), (ZSk > YSk, true)]
is refuted!
refuteCases:  refuting the case
              [(YSk > XSk, true), (ZSk > XSk, true), (ZSk > YSk, false)].

Resource per subcase is  6250,
the total remaining resource is  Fin 338485.

Failed to refute the subcase. Additional support =  {}.
Failed to refute the subcase. Additional support =  {}.

The case of  [(YSk > XSk, true),(ZSk > XSk, true),(ZSk > YSk, false)]
is refuted!
refuteCases:  refuting the case
              [(YSk > XSk, true),(ZSk > XSk, false),(ZSk > YSk, true)].

Resource per subcase is  6250,
the total remaining resource is  Fin 289508.

Failed to refute the subcase. Additional support =  {}.

The case of  [(YSk > XSk, true), (ZSk > XSk, false)]
is refuted!
refuteCases:  refuting the case
              [(YSk > XSk, false), (ZSk > XSk, true), (ZSk > YSk, true)].

Resource per subcase is  6250,
the total remaining resource is  Fin 261757.

Failed to refute the subcase. Additional support =  {}.
Failed to refute the subcase. Additional support =  {}.

The case of  [(YSk > XSk, false),(ZSk > XSk, true),(ZSk > YSk, true)]
is refuted!
refuteCases:  refuting the case
              [(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, true)].

Resource per subcase is  6250,
the total remaining resource is  Fin 213664.

Failed to refute the subcase. Additional support =  {}.
Failed to refute the subcase. Additional support =  {}.

The case of  [(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, true)]
is refuted!
refuteCases:  refuting the case
              [(YSk > XSk, false), (ZSk > XSk, true), (ZSk > YSk, false)].

Resource per subcase is  6250,  t
he total remaining resource is  Fin 168281.

Failed to refute the subcase. Additional support = {}.
```

```
Failed to refute the subcase. Additional support = {}.

The case of  [(YSk > XSk, false), (ZSk > XSk, true), (ZSk > YSk, false)]
is refuted!
refuteCases:  refuting the case
             [(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, false)].

Resource per subcase is  6250,
the total remaining resource is  Fin 120300.

Failed to refute the subcase. Additional support = {}.
Failed to refute the subcase. Additional support = {}.

The case of  [(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, false)]
is refuted!


Goal formula =  forall [X,Y,Z] (isOrdered (insert Z (insert Y (X:nil))))

is proved     = True
rcRem         = Fin 73439
refuted cases =
[
[(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, false)],
[(YSk > XSk, false), (ZSk > XSk, true ), (ZSk > YSk, false)],
[(YSk > XSk, false), (ZSk > XSk, false), (ZSk > YSk, true )],
[(YSk > XSk, false), (ZSk > XSk, true ), (ZSk > YSk, true )],
[(YSk > XSk, true ), (ZSk > XSk, false)],
[(YSk > XSk, true ), (ZSk > XSk, true ), (ZSk > YSk, false)],
[(YSk > XSk, true ), (ZSk > XSk, true ), (ZSk > YSk, true )]
]

remaining cases = []
```

5.2 Example 2 for lists

In the inductive proof of the statement

```
forall [Xs, X] (isOrdered Xs  ==>  isOrdered (insert X Xs))
```

for the *initial model* of the above calculus, the prover applies the refutational proof for one of the sub-goals, and comes to considering cases for the term `(XSk > YSk)` containing Skolem constants. This enables it to accomplish the proof by induction by construction of the list `Xs`.

6 Why only proof by refutation?

Consider, for example, any problem of kind

```
calculus  |--in variety--     forall X exist Y ( F )
             ("by completion")
```

with some correct formula `F(x,y)`.

Why our prover pretends to do the search only by refutation? In the mathematical practice, a 'direct' proof is provided more often.

Our answer is as follows:

(a) Sometimes a 'direct' proof is much more difficult to find than the proof by contradiction. Such is, for example, the problem
"for any natural number n there exists a prime natural greater than n".

(b) A refutational search by the generalized completion is complete, and it is not clear how to achieve completeness for a 'direct' proof search.

(c) Technically, to apply completion, we need to get free of the existential quantifier. And we do not see how to do this without negation and skolemization.

(d) If a direct proof is possible for a given goal, then it can be extracted from the proof by negation and completion by setting respectively the strategy for choosing current fact in completion (this strategy is a parameter of the completion procedure).

Let us comment this.

First, the prover applies a 'positive' completion on the laws of `calculus`, and reduces the subterms in the goal formula by the current `calculus'`. This lasts a certain number `rcP` of steps. For some problems, this simplifies `F` so that a positive proof is achieved. If the proof is not achieved, there applies the search by negation and completion for the reduced formula from `calculus'`.

Second, consider the following simple example. Perform the scheme of negation and completion for the goal

```
nat |--  forall X (exist Y (Y > X)),
```

where `nat` the calculus for natural numbers, as it is usually defined, with the operator `s` for the successor number, with the operator `">"` for ordering. And let it have an additional axiom

```
(s X) > X = true
```

Then, the search by negation negation and completion transforms the goal as follows.

```
nat  union  (exist X forall Y (not (Y > X)))  |--  true = false
--->
nat  union  (factsFromFormula (not (Y > c)))  |--  true = false
--->
nat  union  [Y > c = false]                   |--  true = false
```

Here `c` is the Skolem constant for `X`, and the universal quantifier for `Y` is presumed. Among superpositions, the completion method must process the ones for the equations `Y > c = false` and `s X > X = true`. This derives `s c > c = false`, and `nat` reduces this to `true = false`.

In general, we have the transformed goal

```
calculus union negFacts  |--  true = false,
```

where `negFacts = factsFromFormula (skolemize negate goalFormula)`.

Suppose, that the strategy for selecting facts (Section 3.7) delays selection from the set `negFacts`. As far as such is delayed, the proof is direct. The usage of negation can be delayed, and this imitates the part of a direct proof search.

References

[B:D:P] L. Bachmair, N. Dershovitz, D. Plaisted. *Completion without failure.* In Ait-Kaci (Editor): Resolution of Equations in Algebraic Structures, Volume 2, pages 1-30, Academic Press, 1989.

[Bu] B. Buchberger. *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.* CAMP. Publ. No.83–29.0 November 1983.

[D:Man] N. Dershovitz, Z. Manna. *Proving termination with multiset orderings.* Commun. ACM **22**, 1979, pages 465 – 476.

[Hsi] J. Hsiang. *Refutational theorem proving using term-rewriting systems.* Artificial Intelligence, 1985, v.25, p.255-300.

[Hsi:Ru] J. Hsiang, M. Rusinowitch. *On word problems in equational theories.* In Th. Ottman (ed.), Proceedings of the Fourteenth International Conference on Automata, Languages and Programming, Karlsruhe, West Germany, July 1987, Springer Verlag, Lecture Notes in Computer Science **267**, (54–71), 1987.

[KB] D. Knuth, P. Bendix: *Simple word problems in universal algebras.* In John Leech, editor, "Computational Problems in Abstract Algebra", (263–297), Pergamos Press, 1970.

[Lo:Hi] B. Löchner, T. Hillenbrand: *A Phytography of Waldmeister.* AC Communications **(15)** (2,3) (2002) (127–133).

[Me] S. Mechveliani. *The Dumatel program system and book* (manuscript). `http://www.botik.ru/ mechvel/`, click at 'Dumatel', `http://www.haskell.org/dumatel/`

[Ro] J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle.* Journal of the ACM (JACM), Volume 12, Issue 1, 1965, pages 23 – 41, ACM Press, New York, USA.

7 APPENDIX. Proof of Theorem (Mls).

Below, the ordering ">>" on terms is a SOTG ordering (Section 3.2).

Theorem (Mls).
The mls-ordering for b-monomials satisfies the following properties.

(Ms1) On ground b-monomials, mls is a total ordering which coincides with the lexicographic extension of ">>" from ground atoms.

(Ms2) Any non-identical reduction of any atom in a b-monomial by equations makes this b-monomial smaller.

(Ms3) If `m >> n`, then for any `p` `p & m >> n`.

(Ms4) Invariantness under regular factors:
if `m >> n` and `p ∩ (m \ n) = []`, then `p & m >> p & n`.

(Ms5) Invariantness under regular substitutions:
if `m >> n`, then for any substitution σ σ `m >>` σ `n`.

Proof for (Ms1):
it follows easily from that ">>" is total on ground terms and from the definition of mls.

Proof for (Ms2):
any non-identical reduction `a'` for an atom `a` in a monomial `a & m` produces a monomial `m1 = {a'} ∪ m`. If `a'` is in `m`, then `a & m >> m1` by the 'subset' rule of the mls definitiom. Otherwise, this holds by the second rule of the mls definition

Proof for (Ms3):
Let `m >> n`, and `p` be any monomial. We need to prove that `p & m >> n`. By induction, it is sufficient to prove this for any atom `p = a`.

If `a` is in `m`, then `p & m = m >> n`.

If `a` is not in `m`, then `p & m >> m` by the definition of mls, and `p & m >> n` by transitivity of mls.

Proof for (Ms4):
Let `m >> n` and `p ∩ (m \ n) = []`. We need to prove that `p & m >> p & n`.

Evidently, it is sufficient to prove this for `p` being an atom `a`.

Let `m >> n` and `a` be not in `(m \ n)`. We need to prove the property

`a & m >> a & n` `(1)`.

If `a` is in `m`, then, by the condition, `a` is also in `n`. Then `(1)` is equivalent to `m >> n`, which is given.

Suppose `a` is not in `m`.

If `a` is not in `n`, then, by cancellation, `(1)` is equivalent to `m >> n`, which is given.

It remains the case of `a` is not in `m` and is in `n`. Here, `(1)` is equivalent to `a & m >> n`. In this case, `a & m >> m` by the definition of mls. So, by the condition and by transitivity, `a & m >> n`.

Proof for (Ms5):
Let `m >> n`, and σ be any substitution. We need to prove that σ `m >>` σ `n`.

For `p = m ∩ n`, denote `m = p & m', n = p & n'`. By Lemma (Mls), `m'` is non-empty, and also we can choose for each `b` in `n'` such `a` in `m'` that `a >> b`. Applying the substitution, we have: σ `m >>` σ `n` iff σ `m' >>` σ `n'`. By the SOTG property for terms, σ `a >>` σ `b` for each atom `b` in `n'` and its chosen supremum `a`. Hence, by Lemma (Mls), σ `m' >>` σ `n'`, hence, σ `m >>` σ `n`.