CHOOSING AN INDUCTION VARIABLE IN UNIVERSALLY QUANTIFIED ATOMIC FORMULAS

Marta Franova, Yves Kodratoff

RAPPORT DE RECHERCHE

L R I

CHOOSING AN INDUCTION VARIABLE IN UNIVERSALLY QUANTIFIED ATOMIC FORMULAS

FRANOVA M / KODRATOFF Y

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

02/2015

Rapport de Recherche N° 1579

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Choosing an induction variable in universally quantified atomic formulas

Marta Fraňová, Yves Kodratoff

Introduction

In logics, a formal proof of a formula consists of an application of axioms and inference rules until the atom TRUE is obtained. When the inference rule called induction principle is applied, we speak of inductive theorem proving. In contrast to usual inference rules that simply look at the pattern of formulas, induction principle involves consideration of elements of the domain over which the given formula is expressed.
For instance, let us consider the inference rule called Modus Ponens. This inference consists of two premises, namely

Premise 1: $A \Rightarrow B$
Premise 2: A

and the logical conclusion is the formula B.

The induction principle is a particular inference rule since it

- requires the domain of the problem formulation to be defined inductively (such as it is the case for natural numbers, lists, etc.)
- requires the problem to be expressed using the universal quantifier (as it is the case for the formula $\forall x\ A(x)$)
- requires that the particular formulas called base and induction steps are proved.

Example:
Natural numbers are defined inductively by the following definition:

- 0 is a natural number;
- if n is a natural number then s(n) is a natural number.

0 and s(n) are called representatives and n is here a sub-representative.
Sometimes this inference is written in the following form:

$$F(0), F(n) \Rightarrow F(s(n)) \vdash \forall x\ F(x).$$

The problem we specifically address here is a study of what happens when the formula to be proved contains several universally quantified variables.

To handle this situation, logicians use the so-called multiple induction.

For instance, for a formula with two variables [Yasuhara, 1971] (p. 104) presents the following double induction:
To prove the formula $\forall n \forall m\ A(n,m)$ perform an 'outer induction' on n and an 'inner induction' on m. That is:

- prove A(0,0)
- assume A(0,b) prove A(0,s(n))
- for all n smaller than or equal to a and for all y
    - suppose F(a,y) and prove F(s(a),0)
    - suppose F(a,y) and F(s(n),b) and prove F(s(a),s(b))

As far as automation of proving theorems by induction is concerned, the first significant work was done by [Boyer and Moore, 1979]. This system relies on backward reasoning and Boyer and Moore realized that the implementation of the induction principle in such a backward reasoning oriented system is not simple. Their particular strategy requires a kind of cleverness the key word of which is 'anticipation'. This means that the system starts with the formula to be proved and then it simplifies it up to the point where the anticipated induction hypotheses can be applied. Such an anticipation is not all straightforward, as shown by the following example.

Example:
Let the function flat (flatten) be defined as follows:

flat(u) = cons(u,nil), if not(listp(u))
flat(u) = app(flat(car(u)),cdr(u))), otherwise

Let the function mc.flat be defined as follows:

mc.flat(u,v) = cons(u,v) if not(listp(u))
mc.flat(u,v) = mc.flat(car(u),mc.flat(cdr(u),v))

Finally, consider the append function

ax15 : append(nil,w) = w
ax16 : append(cons(u,v),w) = cons(u,append(v,w))

Let the formula A(x,y) (where x and y are universally quantified) to be proved is as follows:

mc.flat(x,y) = append(flat(x),y).

Boyer and Moore generate the following steps:

- Base step: assume not(listp(x)) and prove A(x,y)
- Induction step: Assume listp(x) and A(car(x),mc.flat(cdr(x),y)) and A(cdr(x),y) and prove A(x,y).

We can note that the induction hypotheses here are not straightforward.
In contrast to this clever, but somewhat generally difficult solution, we have realized [Franova, 1985a] that a 'lazy', logics inspired solution, can be a convenient compromise… provided we know how to prove by induction theorems of the form ∀ x ∃ z P(x,z), where x and z may be vectors.
Such a 'lazy' solution consists in choosing one variable that becomes the induction variable and the remaining variables are universally quantified in the induction hypotheses. For the previous example of Boyer and Moore it would give the induction hypothesis

∀ p A(car(x),p) & ∀ q A(cdr(x),q).

Then, when our development of the formula A(x,y) makes appear the patterns A(car(x),

t1) or A(cdr(x),t2), we look whether we can create an instantiation of the universally quantified variables in the induction hypotheses that corresponds to t1 and t2.

Of course, this 'lazy' solution is, in general, also longer since it may happen that during an inductive proof with respect to one induction variable we need to start an inductive proof for another variable thus leaving us with a proof inside a proof. Fortunately, such solution consists always of finite steps since there is only a finite number of universally quantified variables.

Goal of the paper

This paper presents an algorithm for the above 'lazy' solution for handling the induction principle automation. Let us recall that it requires the ability to prove theorems containing existentially quantified variables and it is not as clever as the anticipation solution proposed by Boyer and Moore. However, it has the essential advantage of simplifying the automation of induction principle. Moreover it mirrors the standard logics solution.

In Section I we shall show that it is important to choose a suitable induction variable, since an unsuitable variable may lead to a failure.
In Section II we shall present the basic vocabulary and criteria for the choice of the induction variable. Section III presents our algorithm and Section IV presents an artificial example inspired by a real world problem presented in [Boyer and Moore, 1979].
In Conclusion we shall point out further perspectives.

1. Necessity of a suitable induction variable

In this section we present an example that illustrates the importance of a choice of a suitable induction variable.
Let us consider following definitions for functions rt (rotate), lg (length) and app (append):
app: LISTN × LISTN → LISTN
ax15 : app(null,w) = w
ax16 : app(cons(u,v),w) = cons(u,app(v,w))
For app we shall use its associativity and the fact that app(u,app(cons(v,null)),w) = app(u,cons(v,w)).

lg: LISTN → N
ax33 : lg(null) = 0
ax34 : lg(cons(u,v) = s(lg(v))

rt: N × LISTN → LISTN
ax35 : rt(0,w) = w

ax36 : rt(s(n),null) = null
ax3 : rt(s(n),cons(u,v)) = rt(n,app(v,cons(u,null))

Let us consider the formula

rt(lg(x),app(x,y)) = app(y,x) (1)

where the variables x and y are universally quantified. We shall consider the induction steps of the proofs for these two variables.
Let us consider a proof by induction on x.
In the induction step, the variable x is represented by cons(a,b), where a belongs to N and b is from LISTN.
The induction hypothesis suggested in our approach is

rt(lg(b),app(b,y')) = app(y',b), (2)

where y' is universally quantified variable. The goal is to prove

rt(lg(cons(a,b)),app(cons(a,b),y)) = app(y,cons(a,b)). (3)

The evaluations are as follows:

rt(s(lg(b)),cons(a,app(b,y)) = app(y,cons(a,b)) (4)

and then

rt(lg(b),app(app(b,y),cons(a,null))) = app(y,cons(a,b)). (5)

Using now the associativity of app, this leads to the goal to prove

rt(lg(b),app(b,app(y,cons(a,null)))) = app(y,cons(a,b)). (6)

To this formula, the induction hypothesis (2) can be applied with y' equal to app(y,cons(a,null))). This gives the task to prove

app(app(y,cons(a,null))),b)) = app(y,cons(a,b)) (7)

Finally, using the above mentioned property of append, this leads to the formula

app(y,cons(a,b)) = app(y,cons(a,b)) (8)

which is true.
The proof is successful, however we needed to use the properties of append. This is not the case for the variable y.
Consider now the proof of (1) with respect to the variable y. In the induction step, the variable y is represented by cons(a,b) and the induction hypothesis is

rt(lg(x'),app(x',b)) = app(b,x') (9)

with x' universally quantified. The goal is to prove

rt(lg(x),app(x,cons(a,b))) = app(cons(a,b),x). (10)

The evaluation gives

rt(lg(x),app(x,cons(a,b))) = cons(a,app(b,x)). (11)

The induction hypothesis (9) can now be applied with x' equal to x. This leads to the task to prove

rt(lg(x),app(x,cons(a,b))) = cons(a,rt(lg(x),app(x,b)). (12)

Obviously now, the proof of this formula requires an induction on x. b occurs in non-recursive positions and thus it is not considered as a candidate for the induction variable. The proof does not need complementary properties for app.

This example shows that the choice of the induction variable is an important task in a 'lazy' implementation of the induction principle. In the following, we start with the basic vocabulary and criteria for the choice of the induction variable.

2. Vocabulary and Criteria for the choice of the induction variable

A term is represented as a tree the nodes of which are functional symbols. An atomic formula may also be represented by a tree whose apex is a predicate and the other nodes are functional symbols. In both types of trees, the leaves are constants or variables. For the choice of an induction variable in an atomic formula, we will consider the paths leading to universally quantified variables.

The definitions below are simple adaptations of the classical definitions of a distance and a path in a tree.

If f(t1,t2,…,tn) is a term, the index i corresponding to the term ti will be called the 'number' of the argument ti. We can formulate this definition in a more general way as follows:
We define the 'number' of a an argument u of a term f(x, y, ..., t) or of an atomic formula P (x, y, ..., t) by 1 + the number of arguments that separate argument u from f or P. When n is the number of the argument u, we shall also say that u is the n-th argument.

We will note a path by the list

((symb1 arg_symb1) ... (symbn arg_symbn) var).

In this list symbi is a symbol and arg_symbi is the number of the argument symbi that leads to the variable var. Obviously, the length of the path to a symbol symbi is i in the notation above.

Example :
Let f be a function of three arguments and h a function of two arguments. Consider the term f(x, h(y,x), z), where x, y and z are variables f and h are at functional symbols. The number of the variable z is 3 because it is third in the definition of f and therefore the path to z is ((f 3) z), while the path to x in the second argument of f is ((f 2) (h 2) x).

Definition :
We shall say that an object o is defined recursively for its n-th argument if it is defined in the argument n for a base constructor of the considered inductively defined domain and that in the recursive call o, the value of this argument decreases with respect to a well-founded relation defined on the considered inductively defined domain.

Limitation 1.

In our work, we limit ourselves to the well-founded relations induced by the constructors of the considered constructible domain (see Constructible Domains [Franova, 1995]).

Example :
app: LISTN × LISTN → LISTN
ax15 : app(null,w) = w
ax16 : app(cons(u,v),w) = cons(u,app(v,w))
The function app (append) is defined recursively with respect to the first argument. v is smaller than cons (u, v) in LISTN.

In the case of recursive definitions set on constructible domains, the algorithm for choosing the induction variable takes into account that in the general case of proof, recursive axioms, that is to say those where the object defined is expressed in terms of himself, several arguments can be expressed using recursive representatives (as defined in [Franova, 1985b]). We shall call these arguments 'evaluable' because they are evaluated during the proof. Note that, by definition of constructible domains, their constructors are always evaluable. The arguments that are not evaluable are called 'non-evaluable.'

By convention, the objects that are not defined recursively will be considered evaluable in all their arguments because their evaluation has no direct impact on the complexity of a proof by induction.

Example :
Consider the predicate
P< : N × N → BOOL
ax30 : P<(0,w), if true
ax31 : P<(u,0), if false
ax32 : P<(s(u),s(v)), if P<(u,v)
The predicate P <is evaluable in its two arguments because the axiom ax32 is defined as ~~to~~ P<(s(u),s(v)), where the first and the second argument are evaluable.

Example :
app: LISTN × LISTN → LISTN
ax15 : app(null,w) = w
ax16 : app(cons(u,v),w) = cons(u,app(v,w))
The function app (append) is evaluable in its first argument. It is non-evaluable in its second argument because it is not defined in terms of constructors for its second argument.

Definition :
We shall say that a symbol symb is evaluable in a path
(... (symb n) … var)
if symb is evaluable in its n-th argument.

Example :
Let the function rev (reverse) be defined as follows
rev : LISTN → LISTN
ax17: rev(null) = null
ax18: rev(cons(u,v)) = app(rev(v),cons(u,null))
Consider the term rev(app(cons(p,q),y),z) with the variables p, q, y, z. app is evaluable in the path ((rev 1) (app 1) (cons 1) p)) but it is not evaluable in the path ((rev 1) (app 2) y)).

Example :
car : LISTN → N
ax013 : car(cons(u,v)) = u
Here, the function car is evaluable in its argument.

Definition :
We shall say that a path of a tree leading to the variable var is evaluable if all the symbols of this path are evaluable. In such a case we shall also say that the variable var is evaluable.

Example :
Let the function rev (reverse) be defined as
rev : LISTN → LISTN
ax17: rev(null) = null
ax18: rev(cons(u,v)) = app(rev(v),cons(u,null))
Consider the term rev(app(cons(p,q),y)) with the variables p, q, y. The path ((rev 1) (app 1) (cons 2) q) is evaluable for q. The path ((rev 1) (app 2) y) is not evaluable for y.

We shall say that a variable var is **evaluable** in an expression exp, if there is at least one path of exp leading to var where all the symbols are evaluable. In this case, we will also say that it has an evaluable occurrence in exp. It will be called **non-evaluable** if all instances of non-evaluable.
We will say that a variable is **purely evaluable** in an expression exp, if it is evaluable in all its occurrences in exp.

Definition : 'mutilating' function

A definition of a recursive function f is **mutilating** when, for the case of recursive definition, some sub-representatives are displaced from an evaluable argument to a non-evaluable argument in the recursive call.

Example :
Let us consider the following recursive axiom for rev:
rev(cons(u,v)) = append(rev(v),cons(u,nil)))
In this case, the sub-representative u is moved to the non-evaluable argument of append

function. Therefore, rev is mutilating.

Let us consider the recursive axiom for qrev:

qrev(cons(u,v),w) = qrev(v,cons(u,w))

In this case, the sub-representative u is moved to the non-evaluable argument of the function qrev itself. Therefore, qrev is mutilating.

Criteria for the choice of the induction variable:

Criterion 1. (observation of the variables evaluable character)
This criterion expresses a known property in inductive theorem proving, namely that for an atomic formula Th, the difficulty of application of induction hypothesis generated for a given variable var depends on the non-evaluable and evaluable positions of var in Th. Initially, by default, all universally quantified variables of Th are candidates to become the induction variable. For the choice of the induction variable, if a variable of Th is purely non-evaluable, it is eliminated from the set of candidates. In other words, the set of candidates is represented by variables that are universally quantified and which have at least one evaluable occurrence in Th.

Criterion 2. (path length measurement)
Here, we consider the variables that are covered by criterion 1, that is to say, they have at least one evaluable occurrence in Th.
Consider variable var located in Th in n evaluable positions. Let c1 be path of length lg1, c2 of length lg2, … and cn of length lgn. For comparison of the variable var with other candidates, it is the path of the maximum length, that is to say, the path that is max(length(lg1, lg2, ..., lgn)) will be considered for comparison of var with other candidates. We speak of this procedure such as reduction the number of possibilities for var.

After making such a reduction, consider the variables x1, ..., xm to which are associated the respective maximum lengths of paths lg1, ..., lgm. Any of these variables with the minimal path length is selected as the best candidate.

Criterion 3. (Case of recursive 'mutilating' functions)
We will process variables that have in their paths mutilating functions as follows. We introduce for each mutilating functional symbol a **penalty** that is equal to the depth of this symbol in this path increased by 1.
We define the global penalty of a variable as the sum of all the penalties of mutilating functional symbols encountered on the path to this variable. For example, consider the term

i + strops(cons(nth(str,i),nil),rev(pat)),

where i, str and pat are universally quantified variables. The variable pat occurs in the term

i + strops(…,rev(pat)).

On the path to pat occurs the mutilating function rev which, by definition, has a penalty

of 4. The overall penalty of variable pat is 4.

To take into account the fact that several mutilating symbols may occur on the path to a variable, we introduce the concept of **penalty coefficient**, which is the number of all mutilating symbols on the path considered.
For each evaluable path of variable var in formula Th, the algorithm calculates first the **profile** of this variable on this path, that is to say, the triple

(depth global_penalty coeff_penalty)

where 'depth' is the length of the path, 'global_penalty' is the overall penalty of var in this path and coeff_penalty is the coefficient of the var penalty.
If var appears in several paths, the algorithm chooses its "worst representative", that is to say, the profile that has the worst value (depth global_penalty coeff_penalty) by first judging their penalty coefficient and if this does not distinguish profiles of this variable, continuing with the overall penalty and if some profiles are still not distinguished, judging them by their depth. The following selection algorithm implements the intuitive description we have just given.

3. Algorithm for the choice for an atomic formula F

Overview of the algorithm:
If F contains variables that are purely evaluable in F, choose only among these variables.

- First, calculate the value of (depth global_penalty coeff_penalty) for all evaluable variable of F.
- Next, choose the worst representative of all these variables, that is to say the one with the coefficient of the maximum penalty in (depth global_penalty coeff_penalty). If the variable has multiple paths with the same maximum coefficient, choose the one with the maximum total penalty. Among the paths with the same coefficient of maximum and maximum global penalty, the worst is the one that has the maximum depth.
- Finally, among the worst representatives of recursive variables of F, choose the variable with the smallest penalty coefficient. If several variables have the same coefficient, choose one that has the smallest total penalty. If several variables have the same global penalty, choose the variable with the smallest depth. If there is still ambiguity take any variable.

Let CVI (Candidates for Variables of Induction) be the set of all the universally variables of the formula F to be proven.

Step 1 : Pinpoint the set CVI_F of all evaluable variables in F. That is, these variables are in evaluable positions in all the terms of F. Note that a variable of F can be evaluable in one term of F while non-evaluable in another term of F. Such a variable will not be included in CVI_F.
Step 2 :

- **if** CVI_F is empty
- **then**

  Step 1. pinpoint the list *TF* of couples (term_i, variables purely evaluable in the same term_i)
  - **Step 2.**
    - if *TF* is empty (i.e. all the positions of all the variables are non-evaluable)
    - then failure
    - else
      - Step 1.
        - apply the procedure BB to *TF*, thus creating **TF**
        - apply the procedure CC to **TF** thus providing the chosen variable
- **else** (if there are some variables purely evaluable in F)
  - **Step 1.** determiner the list *TF** of all the couples (term_i, variables purely evaluable in term_i)
  - **Step 2.** eliminate from *TF** all the terms in which do not occur the variables of CVI_F, thus providing the list *TF*
  - **Step 3.** apply the procedure BB to *TF* ; this provides the list **TF**
  - **Step 4.** apply the procedure CC to **TF** which returns the variable chosen by the whole procedure

Procedure BB

- takes as argument the set *TF*, i.e. the list of couples (term_i, list_var_pur_rec_in_term_i), where list_var_pur_rec_in_term_i is the list of variables purely evaluable in term_i. This gives the result **TF** i.e. the list of triples (term_i var (depth global_penalty coeff_penalty)) for each variable var purely evaluable in term_i.

Procedure CC

- takes as argument the list **TF**, i.e. a list of triples (term_i var (depth global_penalty coeff_penalty)) for each variable var purely evaluable in term_i
- analyses the paths of these variables comparing their depths, their penalty et the penalty coefficient
- returns any variable among those that have the same best value.

The paths analysis is done using the following criterion:

- Step 1. for all variables in **TF**, chose the ‘worst representative’, i.e. the variable that has the worst value of the triple (depth global_penalty coeff_penalty). By default, a variable that occurs in one path and in one term of the given formula is considered as its worst representative.
- Step 2. select the best of these worst representatives.

4. Example

We present here an example of execution that illustrates our algorithm applied to the formula th12. This artificial formula is inspired by a real problem given by Boyer and Moore ([Boyer and Moore, 1979], p. 197-199). We slightly increased the problem a complexity by considering four arguments for the predicate BID4, instead of considering Boyer and Moore's binary predicate LESS THAN.

First of all, let us give the definitions of functions that occur in th12.
BID4 is an artificial predicate the definition of which is not given except the information that it is evaluable in its four arguments. Such an information is sufficient, as the algorithm does not analyze the definitions except their evaluable character. 'null' and 'cons' are the constructors of LISTN (lists of natural numbers). The remaining definitions are given as follows.

+ : N × N → N
ax7 : 0 + v = v
ax8 : s(w) + v = s(w + v)
ax13 : v + 0 = v
ax14 : w + s(v) = s(w + v)
This function is evaluable in both its arguments.

lg : LISTN → N
ax33: lg(null) = 0
ax34: lg(cons(u,v)) = s(lg(v))
lg is evaluable in its argument.

nth : LISTN × N → LISTN
ax27 : nth(cons(u,v),0) = u
ax28 : nth(cons(u,w),s(v)) = nth(w,v)
nth is evaluable in its both arguments.

app: LISTN × LISTN → LISTN
ax15 : app(null,w) = w
ax16 : app(cons(u,v),w) = cons(u,app(v,w))
app is evaluable in its first argument.

rev : LISTN → LISTN
ax17: rev(null) = null
ax18: rev(cons(u,v)) = app(rev(v),cons(u,null))
rev est is evaluable in its argument.

strpos: LISTN × LISTN → N
ax21 : strpos(u,w)=0, if match(u,w)
ax22 : strpos(u,null) = 0
ax23 : strpos(u,cons(w1,w2)) = s(strpos(u,w2))
strpos is evaluable in its second argument.

In the picture that follows, *chiv*(x) is the function that chooses induction variable in ‘x’. th12 is atomic formula

(bid4 (+ i (strpos (cons (nth str i) null) (app (rev str) (rev pat))))
(+ (lg pat) (strpos pat (rev str)))
(+ i (strpos (cons (nth str i) null) (+ (rev pat) (rev str))))
(+ (lg **str**) (strpos pat (rev (rev **str**))))
)

for which the program *chiv* seeks the induction variable.

TF is a list of couples

(term_j, variables purely evaluable in the same term_j).

TF is a list of triples (term_j var (depth global_penalty coeff_penalty)) for each variable var purely evaluable in term_j.

STR, I et PAT are variables universally quantified and CONS and NULL are the constructors of LISTN.

Let us note that this example is purely artificial in the sense that the type of the expressions is not being considered.

For instance, there is an addition of two lists: ‘(+ (rev pat) (rev str))’.

Here is the picture of the execution:

```
>(*chiv* th12)

"Formula considered is "
(BID4 (+ I (STRPOS (CONS (NTH STR I) NULL) (APP (REV STR) (REV PAT))))
      (+ (LG PAT) (STRPOS PAT (REV STR)))
      (+ I (STRPOS (CONS (NTH STR I) NULL) (+ (REV PAT) (REV STR))))
      (+ (LG STR) (STRPOS PAT (REV (REV STR)))))
"CVI_F is the list"
NIL
"*TF* is"
(((+ (LG PAT) (STRPOS PAT (REV STR))) (STR))
 ((+ I (STRPOS (CONS (NTH STR I) NULL) (+ (REV PAT) (REV STR)))) (PAT))
 ((+ (LG STR) (STRPOS PAT (REV (REV STR)))) (STR)))
"**TF** is "
(((+ (LG PAT) (STRPOS PAT (REV STR))) STR (4 4 1))
 ((+ I (STRPOS (CONS (NTH STR I) NULL) (+ (REV PAT) (REV STR)))) PAT
  (5 5 1))
 ((+ (LG STR) (STRPOS PAT (REV (REV STR)))) STR (5 9 2)))
"The chosen variable is "
PAT
PAT

>
```

Consider the results displayed:

The list of variables purely evaluable in th12 is CVI_F, it is NIL. For instance, in the first term of th12, i.e. in

(+ i (strpos (cons (nth str i) null) (app (rev str) (rev pat))))

there is no purely evaluable variable. In *TF*, one can see that variable STR is purely evaluable only in the terms

(+ (lg pat) (strpos pat (rev **str**)))

and

(+ (lg **str**) (strpos pat (rev (rev **str**)))).

The variable PAT is purely evaluable only in the term

(+ i (strpos (cons (nth str i) null) (+ (rev **pat**) (rev str)))).

The first element of **TF** is

((+ (lg pat) (strpos pat (rev **str**))) str (4 4 1))

(4 4 1) is here the profile of the variable STR in the term
(+ (lg pat) (strpos pat (rev str))),
i.e. the first '4' is the depth of STR in this term, the second '4' is its global penalty and '1' is the penalty coefficient. The same convention applies for all the other elements of **TF**.

The algorithm *chiv* returns PAT as the induction variable for th12 as its worst representative, i.e. the profile (5 5 1) is better that the worst representative of the variable STR, i.e. the profile (5 9 2).

Conclusion

In this paper we present an algorithm for the choice of the induction variable for 'lazy' implementation of the induction principle. We have illustrated that such a task is important and differs from the clever and not straightforward solution proposed in [Boyer and Moore, 1979].
Presently, our algorithm deals with universally quantified atomic formulas only. It will be extended and, if necessary, modified in order to include also formulas that contain existential quantifiers and other logical connectives. Only then we shall compare our final solution with those proposed by the other existing approaches.

Acknowledgments

We thank to Michèle Sebag and Dieter Hutter for their suggestions and remarks in the research part of this work.

References

[Boyer and Moore, 1979] R. S. Boyer, J S. Moore: A Computational Logic; Academic Press, 1979.

[Franova, 1985a] M. Franova: A Methodology for Automatic Programming based on the Constructive Matching Strategy; in: B. F. Caviness, (ed): EUROCALL'85; Lecture Notes in Computer Science 204, Springer-Verlag, 1985, 568-570.

[Franova, 1985b] M. Franova: CM-strategy : A Methodology for Inductive Theorem Proving or Constructive Well-Generalized Proofs; in: A. K. Joshi, (ed): Proceedings of the Ninth International Joint Conference on Artificial Intelligence; August, Los Angeles, 1985, 1214-1220.

[Franova, 1995] M. Franova: A Theory of Constructible Domains - a formalization of inductively defined systems of objects for a user-independent automation of inductive theorem proving, Part I; Rapport de Recherche No.970, L.R.I., Université de Paris-Sud, Orsay, France, Mai, 1995.

[Yasuhara, 1971] A. Yasuhara: Recursive Function Theory and Logic; Academic Press, New York, 1971.