7

Knowledge Representation in a Proof Checker for Logic Programs

Emmanouil Marakakis, Haridimos Kondylakis and Nikos Papadakis
Department of Sciences, Technological Educational Institute of Crete,
Greece

1. Introduction

Lately the need for systems that ensure the correctness of software is increasing rapidly. Software failures can cause significant economic loss, endanger human life or environmental damage. Therefore, the development of systems that verify the correctness of software under all circumstances is crucial.

Formal methods are techniques based on mathematics which aim to make software production an engineering subject as well as to increase the quality of software. *Formal verification*, in the context of software systems, is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics. *Formal program verification* is the process of formally proving that a computer program does exactly what is stated in the program specification it was written to realize. Automated techniques for producing proofs of correctness of software systems fall into two general categories: 1) *Automated theorem proving (*Loveland, 1986), in which a system attempts to produce a formal proof given a description of the system, a set of logical axioms, and a set of inference rules. 2) *Model checking*, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.

Neither of these techniques works without human assistance. Automated theorem provers usually require guidance as to which properties are "interesting" enough to pursue. Model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model.

Interactive verifiers or *proof checkers* are programs which are used to help a user in building a proof and/or find parts of proofs. These systems provide information to the user regarding the proof in hand, and then the user can make decisions on the next proof step that he will follow. Interactive theorem provers are generally considered to support the user, acting as clerical assistants in the task of proof construction. The *interactive systems* have been more suitable for the systematic formal development of mathematics and in mechanizing formal methods (Clarke & Wing, 1996). *Proof editors* are interactive language editing systems which ensure that some degree of “semantic correctness” is maintained as the user develops the proof. The *proof checkers* are placed between the two extremes, which are the automatic theorem provers and the proof editors (Lindsay, 1988).

In this chapter we will present a proof *checker* or an *interactive verifier* for logic programs which are constructed by a schema-based method (Marakakis, 1997), (Marakakis & Gallagher, 1994) and we will focus on the knowledge representation and on its use by the core components of the system. A *meta-program* is any program which uses another program, the object program, as data. Our proof checker is a meta-program which reasons about object programs. The logic programs and the other elements of the theory represented in the Knowledge Base (KB) of our system are the object programs. The KB is the data of the proof checker. The proof checker accesses and changes the KB. The representation of the underlying theory (object program) in the proof checker (meta-program) is a key issue in the development of the proof checker. Our System has been implemented in Sicstus Prolog and its interface has been implemented in Visual Basic (Marakakis, 2005), (Marakakis & Papadakis, 2009)

2. An Overview of the main components of the proof checker

This verifier of logic programs requires a lot of interaction with the user. That is why emphasis is placed on the design of its interface. The design of the interface aims to facilitate the proof task of the user. A screenshot of the main window of our system is shown in Fig. 1.

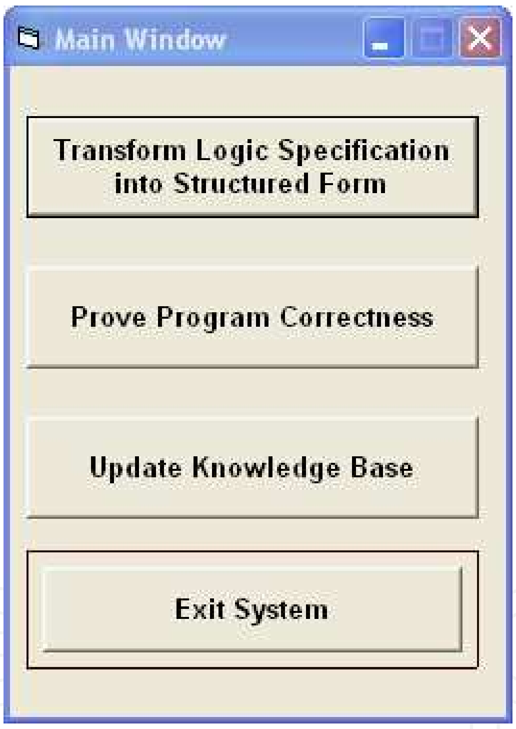

Main Window
Transform Logic Specification into Structured Form
Prove Program Correctness
Update Knowledge Base
Exit System

Fig. 1. The main window of the proof-checker.

Initially, all proof decisions are taken by the programmer. The design of the interface aims to facilitate the proof task of the user. This interactive verifier of logic programs consists of three distinct parts the *interface,* the *prover* or *transformer and the knowledge base (KB).* The interface offers an environment where the user can think and decide about the proof steps that have to be applied. The user specifies each proof step and the prover performs it. A high-level design of our system is depicted in Fig. 2. The main components of the proof checker with their functions are shown in this figure. The prover of the system consists of the following two components. 1) The component “*Spec Transformer*” transforms a specification expressed in typed FOL into structured form which is required by our correctness method (Marakakis, 1997), (Marakakis, 2005). 2) The component "*Theorem Proof Checker* supports the proof task of the selected correctness theorem.

The "*KB Update*" subsystem allows the user to update the KB of the system through a user-friendly interface. The knowledge base (KB) and its contents are also shown in Fig. 2. The KB contains the representation of specifications, theorems, axioms, lemmas, and programs complements. It also has the representation of FOL laws in order to facilitate their selection for application. These entities are represented in ground representation (Hill & Gallagher, 1998). The main benefit of this representation is the distinct semantics of the object program variables from the meta-variables. It should be noted that the user would like to see theorems, axioms, lemmas and programs in a comprehensible form which is independent of their representation. However, the ground representation cannot be easily understood by users. Moreover, the editing of elements in ground representation is error-prone. Part of the interface of the system is the "*Ground-Nonground Representation Transformer*" component which transforms an expression in ground representation into a corresponding one in the standard formalism of FOL and vice-versa. The standard form of expressions helps users in the proof task and for the update of the KB.

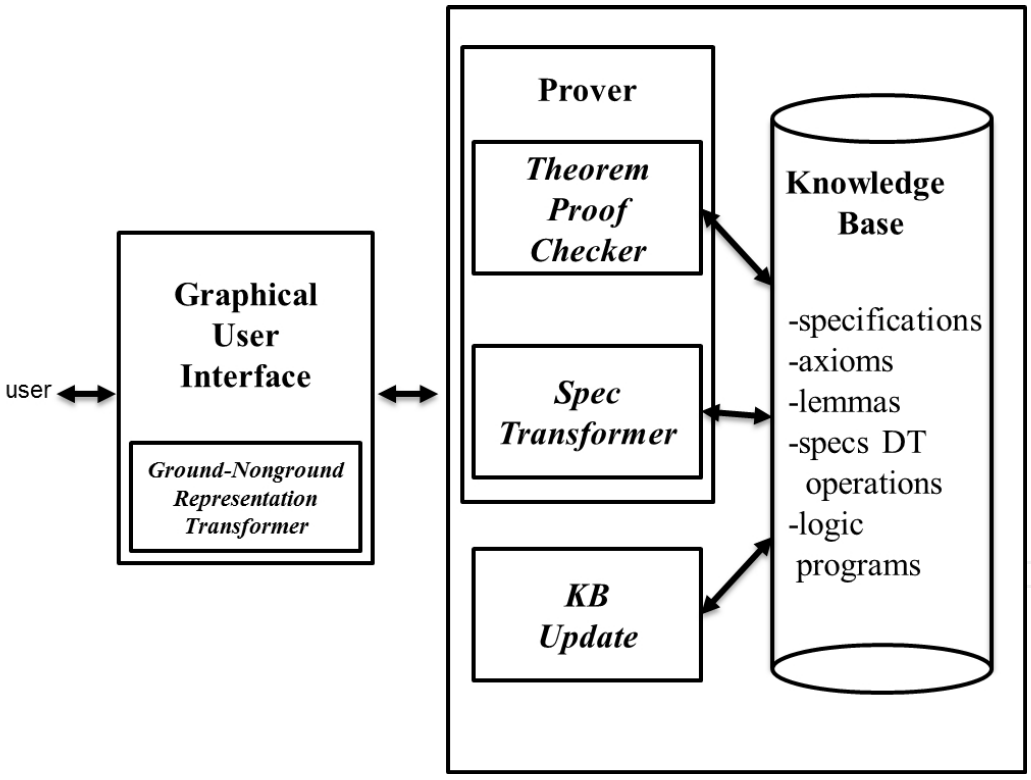


Fig. 2. Main components of the proof-checker.

3. Knowledge representation

Knowledge and *representation* are two distinct concepts. They play a central role in the development of intelligent systems. *Knowledge* is a description of the world, i.e. the problem domain. *Representation* is how knowledge is encoded. *Reasoning* is *how* to extract more information from what is explicitly represented.

Different types of knowledge require different types of representation. Different types of knowledge representation require different types of reasoning. The most popular knowledge representation methods are based on *logic*, *rules*, *frames* and *semantic nets*. Our discussion will be focused on knowledge representation based on logic.

Logic is a language for reasoning. It is concerned with the truth of statements about the world. Each statement is either "true" or "false". Logic includes the following: a) *syntax* which specifies the symbols in the language and how they can be combined to form sentences, b) *semantics* which specify how to assign a truth to a sentence based on its meaning in the world and c) *inference rules* which specify methods for computing new sentences from existing sentences. There are different types of logic, i.e. propositional logic, first-order predicate logic, fuzzy logic, modal logic, description logic, temporal logic, etc. We are concerned on knowledge representation and reasoning based on typed first-order predicate logic because our correctness method is based on typed FOL.

Another classification of knowledge representation is *procedural* and *declarative* knowledge representation. *Declarative knowledge* concerns representation of the problem domain (world) as a set of truth sentences. This representation expresses "*what something is*". On the other hand, the procedural *knowledge* concerns tasks which must be performed to reach a particular goal. In procedural representation, the control information which is necessary to use the knowledge is embedded in the knowledge itself. It focuses on "*how something is done*". In the same way, *declarative programming* is concerned with writing down "*what*" should be computed and much less with "*how*" it should be computed (Hill & Lloyd, 1994). Declarative programming separates the control component of an algorithm (the *"how"*) from the logic component (the "*what*"). The key idea of declarative programming is that a program is a theory (in some suitable logic) and computation is deduction from the theory (Lloyd, 1994). The advantages of declarative programming are: a) teaching, b) semantics, c) programmer productivity, c) meta-programming and e) parallelism. Declarative programming in Logic Programming means that programs are theories. The programmer has to supply the intended interpretation of the theory. Control is usually supplied automatically by the system, i.e. the logic programming language. We have followed the declarative knowledge representation for the representation of the knowledge base of our system.

3.1 Meta-programming, ground and non-ground representation

A language which is used to reason about another language (or possibly itself) is called *meta-language* and the language reasoned about is called the *object language*. A *meta-program* is a program whose data is another program, i.e. the *object program*. Our proof-checker is a meta-program which manipulates other logic programs. It has been implemented in Prolog and the underlying theory, i.e. the logic programs being verified and the other elements of the KB, is the object program. An important decision is how to represent programs of the object language (i.e. the KB elements in our case) in the programs of the meta-language, i.e. in the meta-programs. G*round representation* and non-*ground representation* are the two main approaches to the representation of object programs in meta-programs. We have followed the ground representation approach for the representation of the elements of the KB of our system. Initially, ground and non-ground representation will be discussed. Then, we will see the advantages and the drawbacks of the two representations.

In logic programming there is not clear distinction between programs and data because data can be represented as program clauses. The semantics of a meta-program depend on the way the object program is represented in the meta-program. Normally, a distinct representation is given to each symbol of the object language in the meta-language. This is called *naming relation* (Hill & Gallagher, 1998). Rules of construction can be used to define the representation of the constructed terms and formulas. Each expression in the language of the object program should have at least one representation as an expression in the language of the meta-program. The *naming relation* for constants, functions, propositions, predicates and connectives is straightforward. That is, constants and propositions of the object language can be represented as constants in the meta-language. Functions and predicates of the object language can be represented as functions in the language of meta-program. A connective of the object language can be represented either as a connective or as a predicate or as a function in the meta-language. The main problem is the representation of the variables of the object language in the language of the meta-program. There are two approaches. One approach is to represent the variables of the object program as ground terms in the meta-program. This representation is called *ground representation*. The other approach is to represent the variables of the object program as variables (or non-ground terms) in the meta-program. This representation is called *non-ground representation.*

Using non-ground representation of the object program is much easier to make an efficient implementation of the meta-program than using ground representation. In non-ground representation, there is no need to provide definitions for *renaming*, *unification* and *application* of substitutions of object language formulas. These operations which are time consuming do not require special treatment for the object language terms. The inefficiency in ground representation is mainly due to the representation of the variables of the object program as constants in the meta-program. Because of this representation complicated definitions for *renaming*, *unification* and *application* of substitutions to terms are required. On the other hand, there are semantic problems with non-ground representation. The meta-program will not have clear declarative semantics. There is not distinction of variables of the object program from the ones of the meta-program which range over different domains. This problem can be solved by using a typed logic language instead of the standard first-order predicate logic. The ground representation is more clear and expressive than the non-ground one and it can be used for many meta-programming tasks. Ground representation is suitable for meta-programs which have to reason about the computational behavior of the object program. The ground representation is required in order to perform any complex meta-programming task in a sound way. Its inherent complexity can be reduced by specialization. That is, such meta-programs can be specialized with respect to the representation of the object program (Gallagher 1993).

Another issue is how the theory of the object program is represented in the meta-program. There are again two approaches. One approach is the object program to be represented in the meta-program as program statements (i.e. clauses). In this case, the components of the object program are fixed and the meta-program is specialized for just those programs that can be constructed from these components. The other approach is the object program to be represented as a term in a goal that is executed in the meta-program. In this case the object program can be either fixed or it can be constructed dynamically. In this case the meta-program can reason about arbitrary object programs. This is called *dynamic meta-*

programming. The object program in our proof checker is represented as clauses. The underlying theory is fixed for each proof task.

3.2 Ground representation of object programs in the proof-checker

The KB shown in Fig. 2 contains the representation of specifications, theorems, axioms, lemmas and programs complements. It also has the representation of FOL laws in order to facilitate their selection for application. These KB elements are represented in ground representation (Hill & Gallagher, 1998).. The representation of the main symbols of the object language which are used in this chapter is shown below.

Object language symbol	**Representation**
constant	constant
object program variable	term *v(i)*, *i* is natural
function	term *g(i)*, *i* is natural
proposition, formulas of FOL	term *f(i)*, *i* is natural
predicate	term *p(i)*, *i* is natural
connectives $(\vee,\wedge,\sim,\rightarrow,\leftrightarrow)$	\/, /\, ~, -> , <->
exist $(\exists)$	*ex*
for all $(\forall)$	*all*
length of sequence *x1*(#x1)	*len(v(1):Type):nat*
operation plus (+)	*plus*
operation minus (-)	*minus*
type variable	term *tv(i)*, *i* is natural
type sequence	*seq*
empty sequence (<>)	*nil_seq*
sequence constructor (*Head* :: *Tail*)	*seq_cons*(*Head*, *Tail*) where *Head* and *Tail* are defined in ground representation accordingly.
operator / (*Object*/*Type*)	(*Object* : *Type*)
$x1_i$ / *Type* (e.g. x_1/*a1*)	*v(1, i:nat):Type (e.g. v(1, 1:nat):tv(1))*
equality (=)	*eq*
inequality (≠)	*~eq*
less-equal (≤)	*le*
greater-equal (≥)	*ge*
type natural (N)	*nat*
type integer (Z)	*int*
nonzero naturals (N_1)	*posInt*

Predicates are represented by their names assuming that each predicate has a unique name. In case of name conflicts, we use the ground term *p(i)* where *i* is natural. Sum of n elements, i.e. $\sum_{i=1}^{n} x_i$ is represented as the following ground term: *sum(1:nat, v(2):nat, v(3, v(4):nat):Type):Type* where "*Type*" is the type of x_i.

3.3 Representation of variables

3.3.1 Type variables

The type variables are specified by the lower case Greek letter *a* followed by a positive integer which is the unique identifier of the variables e.g. *a1*, *a2*, *a3*, *a4* etc. Each type variable is represented in ground form by a term of the form *tv(N)* or in simplified form *tvN* where *N* stands for the unique identifier of the variable. For example, the ground representation of type variables *a1*, *a2*, *a3* could be *tv(1)* or *tv1*, *tv(2)* or *tv2*, *tv(3)* or *tv3* respectively.

3.3.2 Object program variables

Object program variables and variables in specifications are expressed using the lower case English letter *x* followed by a positive integer which is the unique identifier of the variables e.g. *x1*, *x2*, *x3*, *x4* etc. Each object variable is represented in ground form by a term of the form *v(N)* where *N* stands for the unique identifier of the variable. For example, the ground representation of the object variables *x1, x2, x3* is *v(1) ,v(2) and v(3)* respectively. Note that, the quantifier of each variable comes before the variable in the formula. Subscripted variables of the form $x1_i$ represent elements from constructed objects. They are represented by a term of the form *v(Id, i:nat):ElementType* where the first argument "*Id*" represents its unique identifier and the second one represents its subscript. "*Id*" is a natural number. This type of variables occurs mainly in specifications. A term like *v(Id, i:nat):ElementType* can be assumed as representing either a regular compound term or an element of a structured object like a sequence. The distinction is performed by checking the types of the elements *x* and *x(i)*. For example, for *i=1* by checking *x1:seq(a1)* and *x1(1:nat):a1*, it can be inferred that *x1(1:nat):a1* is an element of *x1:seq(a1)*.

3.4 Representation of axioms and lemmas

A set of axioms is applied to each DT including the "*domain closure*" and the "*uniqueness*" axioms which will be also presented later on Section 3.7. Each axiom is specified by a FOL formula. Axioms are represented by the predicates "*axiom_def_ID/1*" and "*axiom_def/4*" as follows. The predicate

"*axiom_def_ID(Axiom_Ids)*"

represents the identifiers of all axioms in the KB. Its argument "*Axiom_Ids*" is a list with the identifiers of the axioms. For example, the representation "axiom_def_ID([1,2,3,4])" says that the KB has four axioms with identifiers 1,2,3 and 4.

The specification of each axiom is represented by a predicate of the following form

"axiom_def(Axiom_Id, DT_name, Axiom_name, Axiom_specification)".

The argument "*Axiom_Id*" is the unique identifier of the axiom, i.e. a positive integer. "*DT_name*" is the name of the DT which the axiom is applied to. "*Axiom_name*" is the name of axiom. "*Axiom_specification*" is a list which has the representation of the specification of the axiom.

Example: *Domain closure axiom for sequences*. Informally, this axiom says that a sequence can be either empty or it will consist from head and tail.

Specification:

$$[\forall x1/seq(a2),[x1= < > \vee (\exists x3/a2,\ \exists x4/seq(a2),[x1=x3::x4])]]$$

Representation:

```
axiom_def (1, sequences, 'domain closure',
   [all v(1):seq(tv(1)), (eq(v(1):seq(tv(1)),nil_seq) \/
      [ex v(2):tv(1), ex v(3):seq(tv(1)), eq(v(1):seq(tv(1)),seq_cons(v(2):tv(1),
               v(3):seq(tv(1))):seq(tv(1))) ])]).
```

Similarly, lemmas are represented by the predicates "*lemma_sp_ID/1*" and "*lemma_sp/4*".

3.5 Representation of first-order logic laws

The FOL laws are equivalence preserving transformation rules. Each FOL law is specified by a FOL formula. They are represented by predicates *fol_law_ID/1* and *fol_law/3* as follows. The predicate

"fol_law_ID(FOL_laws_Ids)"

represents the identifiers of all FOL laws in the KB. Its argument "*FOL_laws_Ids*" is a list with the identifiers of all FOL laws. The specification of each FOL law is represented by a predicate of the form

"fol_law(FOL_law_Id, FOL_law_description, FOL_law_specification).".

The argument "*FOL_law_Id*" is the unique identifier of the FOL law, i.e. a positive integer. "*FOL_law_description*" is the name of a FOL law. "*FOL_law_specification*" is a list which has the ground representation of the specification of FOL law.

Example: ($\wedge$ distribution)

Specification:

$$P \wedge (Q \vee R) \leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

Representation:

```
fol_law(2,' ∧ distribution',
      [f1 /\ (f2 \/ f3) <-> (f1 /\ f2) \/ (f1 /\ f3)]).
```

3.6 Representation of theorems

3.6.1 Initial theorem

The theorems that have to be proved must also be represented in the KB. Each theorem is specified by a FOL formula. They are represented by the predicates "*theorem_ID/1*" and "*theorem/4*" as follows. The predicate

"theorem_ID(Theorems_Ids)"

represents the identifiers of all theorems that are available in the KB. Its argument "*Theorems_Ids*" is a list with the identifiers of all theorems. The specification of each theorem is represented by a predicate of the form

"theorem(Theorem_Id, Program_Id, Spec_struct_Id, Theorem_specification)."

The argument "*Theorem_Id*" is the unique identifier of the theorem, i.e. a positive integer. The arguments "*Program_Id*" and "*Spec_struct_Id*" are the unique identifiers of the program and the structured specifications respectively. "*Theorem_specification*" is a list which has the representation of the specification of theorem.

Example: The predicate *sum(x1, x2)* where *Type(sum) = seq(Z)* × *Z* is true iff *x2* is the sum of the sequence of integers *x1*. The correctness theorem for predicate *sum/2* and the theory which is used to prove it is as follows.

$$Comp(Pr) \cup Spec \cup A \ \models \ \forall x1/seq(Z),\ x2/Z\ (sum(x1,x2) \leftrightarrow sum^{S}(x1,\ x2))$$

Pr is the logic program for predicate *sum*/2, excluding the DT definitions. *Comp(Pr)* is the complement of the program *Pr*. *Spec* is the specification of predicate *sum*/2, i.e. $sum^{S}(x1,x2)$. *A* is the theory for sequences, i.e. the underlying DTs for predicate *sum*/2, including the specifications of the DT operations.

Theorem Specification:

$$\forall x1/seq(Z),\ \forall x2/Z\ (sum(x1,x2) \leftrightarrow sum^{S}(x1,x2))$$

Representation:

```
theorem(1, progr1, spec_struct1,
   [all v(1):seq(int),all v(2):int, (sum(v(1):seq(int), v(2):int):int <->
      sum_s(v(1):seq(int), v(2):int))]).
```

3.6.2 Theorems in structured form

The specification of a theorem may need to be transformed into structured form in order to proceed to the proof. The structure form of theorems facilitates the proof task. Each theorem in structured form is specified by a FOL formula. They are represented by the predicates "*theorem_struct_ID/1*" and "*theorem_struct/4*" as follows. The predicate

"theorem_struct_ID(Theorems_Ids)"

represents the identifiers of all theorems available in the KB with the specification part in structured form. Its argument "*Theorems_Ids*" is a list with the identifiers of all theorems in structured form. The specification of each theorem is represented by a predicate of the form

"theorem_struct(Theorem_struct_Id,Program_Id,Spec_struct_Id, Theorem_specification).".

The argument "*Theorem_struct_Id*" is the unique identifier, i.e. a positive integer, of the theorem whose specification part is in structured form. The arguments "*Program_Id*" and "*Spec_struct_Id*" are the unique identifiers of the program and the structured specifications respectively. "*Theorem_specification*" is a list which has the representation of the specification of theorem.

Example

In order to construct the theorem in structured form the predicate specification must be transformed into structured form. The initial logic specification for predicate *sum*/2 is the following.

$$\forall x1/seq(Z),\ x2/Z\ (sum^{S}(x1,x2) \leftrightarrow x2 = \sum_{i=1}^{\#x1} x1_{i}\ )$$

The logic specification of $sum^{S}(x1,x2)$ in structured form and its representation are following.

Theorem Specification:

$$\forall x1/seq(Z),\ \forall x2/Z\ (sum^{S}(x1,x2) \leftrightarrow [x1{=}{<}{>} \wedge x2{=}0 \vee [\exists x4/Z,\ \exists x5/Z,$$
$$\exists x6/seq(Z),[x1{=}x5{::}x6 \wedge x2{=}x5{+}x4 \wedge sum^{S}(x6,x4)]]])$$

Representation:

theorem_struct(1, progr1, spec_struct1,
[all v(1):seq(int), all v(2):int, (sum(v(1):seq(int), v(2):int) <->
((eq(v(1):seq(int),nil_seq:seq(tv(1))) /\ eq(v(2):int,0:int)) \/ [ex v(3):int,
ex v(4):int, ex v(5):seq(int), (eq(v(1):seq(int), seq_cons(v(4):int,
v(5):seq(int)):seq(int)) /\ eq(v(2):int,plus(v(4):int, v(3):int)) /
sum_s(v(5):seq(int), v(3):int))]))]).

3.7 An example theory and theorem

Throughout this Chapter we use the correctness theorem and theory for predicate *sum/2*. That is,

$$Comp(Pr) \cup Spec \cup A\ \ |{=}\ \forall x1/seq(Z),\ x2/Z\ (sum(x1,x2) \leftrightarrow sum^{S}(x1,\ x2))$$

The ground representation of theory is also illustrated.

Theory

The logic program completion *Comp(Pr)* of *Pr* is as follows.

∀x1/seq(Z), ∀x2/Z,[sum(x1,x2) ↔ (p1(x1) ∧ p2(x1,x2) ∨ [∃x3/Z, ∃ x4/seq(Z),
∃ x5/Z,[~p1(x1) ∧ p3(x1,x3,x4) ∧ p4(x1,x3,x5,x2) ∧ sum(x4,x5)]])]
∀x1/seq(Z),[p1(x1) ↔ empty_seq(x1)]
∀ x1/seq(Z), ∀x2/Z,[p2(x1,x2) ↔ neutral_add_subtr_int(x2)]
∀ x1/seq(Z), ∀x2/Z, ∀x3/seq(Z),[p3(x1,x2,x3) ↔ p5(x1,x2,x3) ∧ p6(x1,x2,x3)]
∀ x1/seq(Z), ∀x2/Z, ∀x3/seq(Z),[p5(x1,x2,x3) ↔ head(x1,x2)]
∀ x1/seq(Z), ∀x2/Z, ∀x3/seq(Z),[p6(x1,x2,x3) ↔ tail(x1,x3)]
∀ x1/seq(Z), ∀x2/Z, ∀x3/Z, ∀x4/Z,[p4(x1,x2,x3,x4) ↔ plus_int(x3,x2,x4)]

Representation:

- *progr_clause(progr1, 1,[all v(1):seq(int), all v(2):int, [sum(v(1):seq(int),v(2):int) <-> ((p1(v(1):seq(int)) /\ p2(v(1):seq(int), v(2):int)) \/ [ex v(3):int, ex v(4):seq(int), ex v(5):int,*

[~p1(v(1):seq(int)) /\ p3(v(1):seq(int), v(3):int,v(4):seq(int)) /\ p4(v(1):seq(int),v(3):int, v(5):int, v(2):int) /\ sum(v(4):seq(int),v(5):int)]])]]).

- *progr_clause(progr1, 2, [all v(6):seq(int),[p1(v(6):seq(int)) <-> empty_seq(v(6):seq(int))]]).*
- *progr_clause(progr1, 3, [all v(7):seq(int), all v(8):int,[p2(v(7):seq(int), v(8):int) <-> neutral_add_subtr_int(v(8):int)]]).*
- *progr_clause(progr1, 4, [all v(9):seq(int), all v(10):int, all v(11):seq(int), [p3(v(9):seq(int), v(10):int, v(11):seq(int))<-> p5(v(9):seq(int), v(10):int, v(11):seq(int)) /\ p6(v(9):seq(int), v(10):int, v(11):seq(int))]]).*
- *progr_clause(progr1, 5, [all v(12):seq(int), all v(13):int, all v(14):seq(int),[p5(v(12):seq(int), v(13):int, v(14):seq(int)) <-> head(v(12):seq(int), v(13):int)]]).*
- *progr_clause(progr1, 6, [all v(15):seq(int), all v(17):int, all v(16):seq(int),[p6(v(15):seq(int), v(17):int, v(16):seq(int))<-> tail(v(15):seq(int), v(16):seq(int))]]).*
- *progr_clause(progr1, 7, [all v(18):seq(int), all v(19):int, all v(20):int, all v(21):int,[p4(v(18):seq(int), v(19):int, v(20):int, v(21):int) <-> plus_int(v(20):int, v(19):int, v(21):int)]]).*

The logic specification (*Spec*) is shown in Section 3.6 and its representation in ground form.

The theory *A* of the DT operations including the specification of the DT operations is as follows.

Axioms

Domain closure axiom for sequences

∀ x1/seq(a2),[x1= < > ∀ (∃ x3/a2, ∃x4/seq(a2),[x1=x3::x4])]

Its ground representation is shown in section 3.4.

Uniqueness axioms for sequences

i ∀x1/a2, ∀x3/seq(a2),[~[x1::x3/a2= < >]]
ii ∀x1/a2, ∀x3/a2, ∀x4/seq(a2), ∀x5/seq(a2),[x1::x4=x3::x5 ∧ x1=x3 ∧ x4=x5]

Representation:

- *axiom_def(2, sequences, "uniqueness i", [all v(1):tv(1), all v(2):seq(tv(1)), [~[eq(seq_cons (v(1):tv(1), v(2):seq(tv(1))):tv(1),nil_seq:seq(tv(1)))]]]]).*
- *axiom_def(3, sequences, "uniqueness ii", [all v(1):tv(1), all v(2):tv(1), all v(3):seq(tv(1)), all v(4):seq(tv(1)),[eq(seq_cons(v(1):tv(1), v(3):seq(tv(1))):seq(tv(1)), seq_cons(v(2):tv(1), v(4):seq(tv(1))):seq(tv(1))) -> (eq(v(1):tv(1), v(2):tv(1)) /\ eq(v(3):seq(tv(1)), v(4):seq(tv(1))))]]]).*

Definition of summation operation over 0 entities

$\forall x1/seq(Z),[x1= < > \rightarrow \Sigma((i=1 \ to\ \#x1\ )\ x1_i\ )=0]$

Representation:

- *axiom_def(4, sequences, "summation over 0 entities", [all v(1):seq(int), [eq(v(1):seq(int),nil_seq) -> eq(sum(1:int,len(v(1):seq(int)):int, v(1,v(3):nat):int), 0:int)]]).*

Lemmas

∀x1/seq(a2),[x1 ≠ < > ↔ [∃x3/a2, ∃x4/seq(a2),[x1=x3::x4/a2]]]

∀x1/a2, ∀x3/seq(a2), ∀x4/seq(a2),[x3=x1::x4→(∀x5/N,[2≤x5≤#x3→x3(x5)=x4(x5-1)])]

∀x1/seq(a2), ∀x3/seq(a2), ∀x4/a2,[x1=x4::x3→#x1=#x3+1]

∀x1/a2, ∀x3/seq(a2), ∀x4/seq(a2),[x3=x1::x4/a2→x1=$x3_1$/a2]

Representation:

- lemma_sp(1, sequences, "Non-empty sequences have at least one element",[all v(1):seq(tv(1)), [~eq(v(1):seq(tv(1)),nil_seq:seq(tv(1))) <-> [ex v(2):tv(1), ex v(3):seq(tv(1)), [eq(v(1):seq(tv(1)), seq_cons(v(2):tv(1), v(3):seq(tv(1))):tv(1))]]]]).
- lemma_sp(2,sequences,"if sequence s has tail t then the element si is identical to the element ti-1",[all v(1):tv(1), all v(2):seq(tv(1)), all v(3):seq(tv(1)),[eq(v(2):seq(tv(1)), seq_cons(v(1):tv(1), v(3):seq(tv(1))):seq(tv(1))) -> (all v(4):nat, [le(2:nat, v(4):nat) /\ le(v(4):nat, len(v(2):seq (tv(1))):nat) -> eq(v(2, v(4):nat):tv(1), v(3, minus(v(4): nat,1:nat)))])]]).
- lemma_sp(3, sequences, "If sequence s has tail t then the length of s is equal to the length of t plus 1", [all v(1):seq(tv(1)), all v(2):seq(tv(1)), all v(3):tv(1), [eq(v(1):seq(tv(1)), seq_cons(v(3):tv(1), v(2):seq(tv(1))):seq(tv(1))) -> eq(len(v(1):seq(tv(1))): nat,plus(len(v(2): seq(tv(1))):nat,1:nat))]]).
- lemma_sp(4, sequences, "If sequence s is non-empty then its head h is identical to its first element", [all v(1):tv(1), all v(2):seq(tv(1)), all v(3):seq(tv(1)),[eq(v(2):seq(tv(1)), seq_cons(v(1):tv(1), v(3):seq(tv(1))):tv(1)) -> eq(v(1):tv(1), v(2, 1:int):tv(1))]]).

Logic specifications of DT operations

∀x1/seq(a2),[empty_seq(x1) ↔ x1= < >]

∀x1/Z,[neutral_add_subtr_int(x1) ↔ x1=0]

∀x1/seq(a2), ∀x3/a2,[head(x1,x3) ↔ [x1≠ < > ∧ [∃x4/seq(a2),[x1=x3::x4/a2]]]]

∀x1/seq(a2), ∀x3/seq(a2),[tail(x1,x3) ↔ [∃x4/a2,[x1≠< > ∧ x1=x4::x3/a2]]]

∀x1/Z, ∀x2/Z, ∀x3/Z,[plus_int(x1,x2,x3) ↔ x3=x2+x1]

Representation:

- dtOp_sp(empty_seq, 1, "seq: empty", [all v(1):seq(tv(1)), [empty_seq(v(1):seq(tv(1))) <-> eq(v(1):seq(tv(1)),nil_seq:seq(tv(1)))]]).
- dtOp_sp(head, 2, "seq: head", [all v(1):seq(tv(1)), all v(2):tv(1),[head(v(1):seq(tv(1)), v(2):tv(1)) <-> [~eq(v(1):seq(tv(1)),nil_seq:seq(tv(1))) /\ [ex v(3):seq(tv(1)), [eq(v(1):seq (tv(1)),seq_cons(v(2):tv(1), v(3):seq(tv(1))):tv(1))]]]]]).
- dtOp_sp(tail, 3, "seq: tail", [all v(1):seq(tv(1)), all v(2):seq(tv(1)), [tail(v(1):seq(tv(1)), v(2):seq(tv(1))) <-> [ex v(3):tv(1),[~eq(v(1):seq(tv(1)), nil_seq:seq (tv(1))) /\ eq(v(1), seq_cons(v(3):tv(1),v(2):seq(tv(1))):tv(1))]]]]]).
- dtOp_sp(neutral_add_subtr_int, 8, "int: neutral_add_subtr_int", [all v(1):int, [neutral_add_subtr_int(v(1):int) <-> eq(v(1):int,0:int)]]).
- dtOp_sp(plus_int, 9,"int: plus_int", [all v(1):int, all v(2):int, all v(3):int, [plus_int(v(1):int,v(2):int,v(3):int) <-> eq(v(3):int,plus(v(2):int,v(1):int))]]).

4. Schematic view of the Interaction of the main components

In this section, a schematic view of the proof checker and the interaction of its main components will be shown. In addition, the functions of its components will be discussed. An example of a proof step will illustrate the use of the KB representation in the proof task.

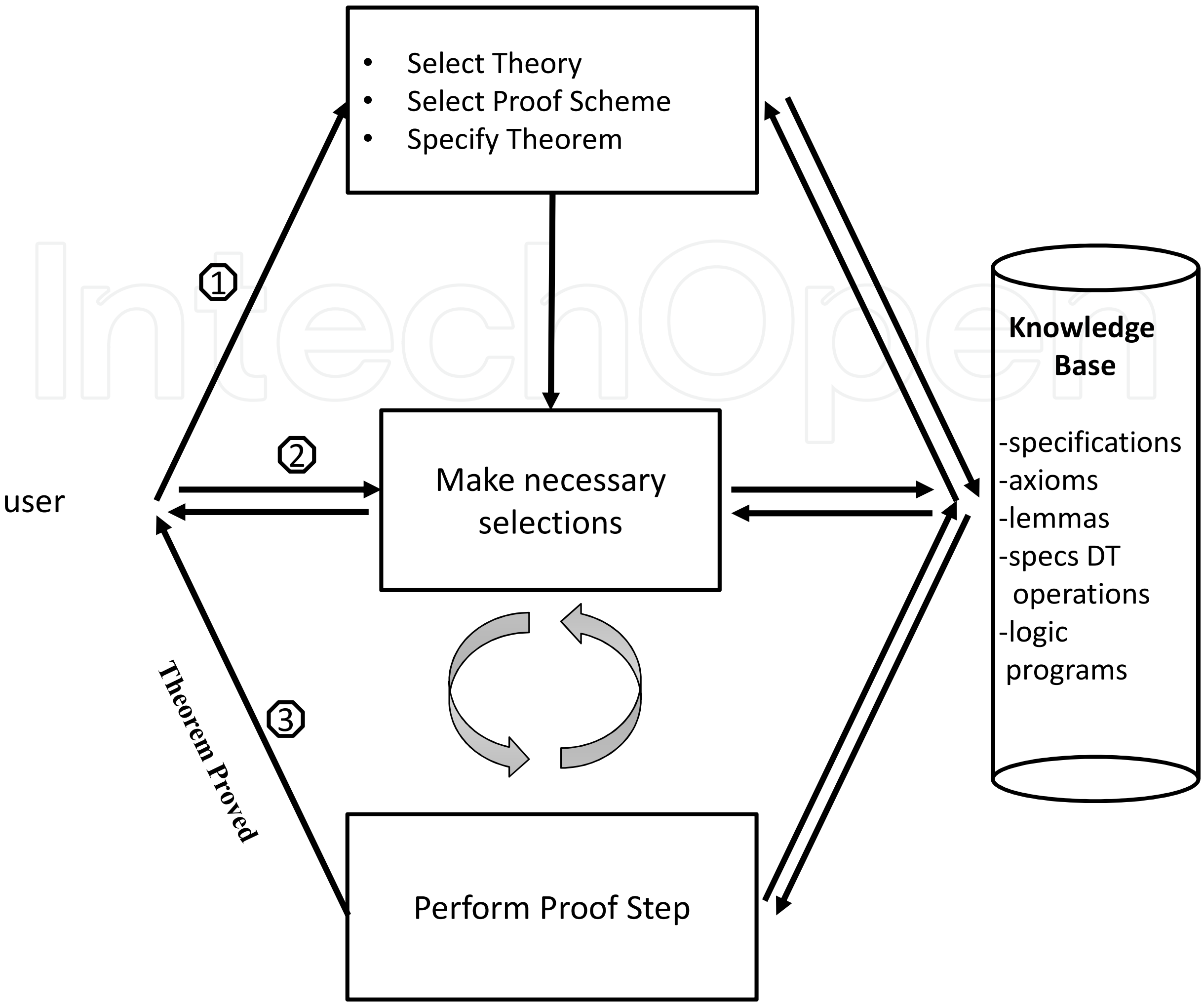


Fig. 3. Schematic View of the Theorem Proof Checker.

4.1 Schematic view of the theorem proof checker

The process of proving a theorem is shown in Fig. 3 and consists of three steps.

- Step 1: In order to prove the correctness of a theorem the user initially has to specify the theorem that is going to be proved and to select the theory and the proof scheme that will be used for the proof. The theory is retrieved from the *KB* and it is presented to the user for selection. It consists of a *program complement*, a *logic specification*, *axioms* and *lemmas.* The corresponding window of the interface which allows the user to make these selections is shown in Fig. 6.
- Step 2: After the selection the user proceeds to the actual proof of the specific theorem. In order to do that he has to select specific parts from the theorem, the theory and the transformation rules that will be applied. The transformation rules that can be applied are first order logic (FOL) laws, folding and unfolding.
- Step 3: In this step the selected transformation is applied and the equivalent form of the theorem is presented to the user. The user can validate the result. He is allowed to approve or cancel the specific proof step.

The last two steps are performed iteratively until the theorem is proved.

4.2 Schematic view of specification transformer

Fig. 4 depicts the procedure for transforming a specification in the required structured form, which is similar to the previous case. In this case however, the underlying theory consists of *Spec* U *Axioms* U *Lemmas*. Initially, the user selects a specification, then the rest elements of the theory are automatically selected by the system. Next, in step 2, the user has to select specific theory elements and transformation rules. In step 3, the selected transformation rule is performed. Step 2 and step 3 are performed iteratively until the specification is transformed in the required structured form.

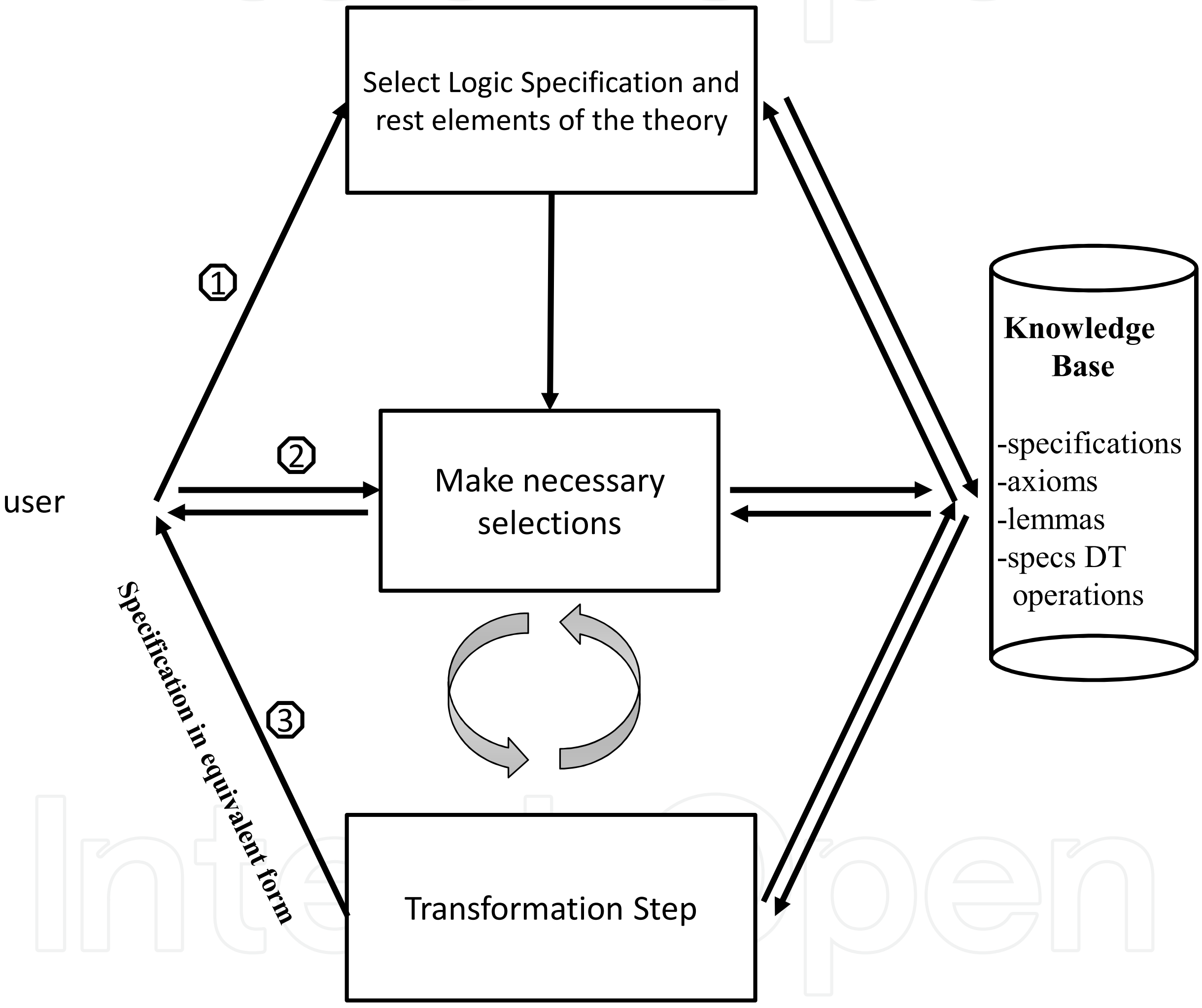

Select Logic Specification and rest elements of the theory
1
2
user
Make necessary selections
Knowledge Base
-specifications
-axioms
-lemmas
-specs DT operations
Specification in equivalent form
3
Transformation Step

Fig. 4. Schematic View of Specification Transformer.

4.3 Illustration of a proof step

The "*Transformation Step*" procedure is actually a sub-procedure of the "*Perform Proof Step*" procedure and that is why we will not present it. The schematic view of the main algorithm for the procedure which performs a proof step, i.e. "*performProofStep", is* shown in Fig. 5. It is assumed that the user has selected some theory elements, and a transformation rule that should be applied to the current proof step.

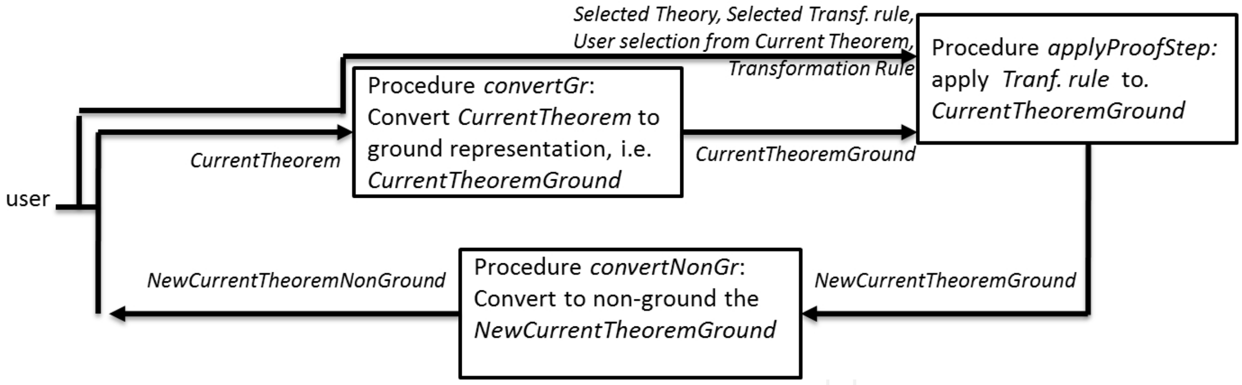

Selected Theory, Selected Transf. rule,
User selection from Current Theorem,
Transformation Rule
Procedure applyProofStep: apply Tranf. rule to. CurrentTheoremGround
Procedure convertGr: Convert CurrentTheorem to ground representation, i.e. CurrentTheoremGround
CurrentTheorem
CurrentTheoremGround
user
NewCurrentTheoremNonGround
Procedure convertNonGr: Convert to non-ground the NewCurrentTheoremGround
NewCurrentTheoremGround

Fig. 5. Schematic view of the "performProofStep" procedure.

The function block diagram of the algorithm "*performProofStep*" shown in Fig. 5 will be discussed through an example. Consider that our theorem has been transformed and its current form is the following:

$$\forall x1/seq(Z),\ \ \forall x2/Z\ (sum^{S}(x1,x2)\ \leftrightarrow$$
$$(x1= <> \wedge x2=0 \vee [\exists x3/Z,[\ \exists x4/seq(Z),false \wedge x2=x4+x3 \wedge sum^{S}(x4,x3)]]))$$

The user has selected the following FOL law to be applied to the above theorem:

$$P \wedge false \leftrightarrow false$$

Initially, the current theorem is converted to the corresponding ground representation by the procedure "*ConvertGr*" and we get:

```
[all v(1):seq(int),[all v(2):int,sum_s(v(1):seq(int),v(2):int)<->
   (eq(v(1):seq(int),nil_seq:seq(int))/\eq(v(2):int,0:int) \/
         [ex v(3):int,[ex v(4):int,[ex v(5):seq(int),
           false/\eq(v(2):int,plus(v(4):int,v(3):int):int)/\sum_s(v(5):seq(int),v(3):int)]]])]]
```

Then, the procedure "a*pplyProofStep*" applies the transformation rule to the current theorem and derives the new theorem. In order to do that, this procedure constructs and asserts a set of clauses which implement the selected transformation rule. Then, it applies this set of clauses and derives the new theorem in ground representation. That is,

```
[all v(1):seq(int),[all v(2):int,sum_s(v(1):seq(int),v(2):int)<->
(eq(v(1):seq(int),nil_seq:seq(int))/\eq(v(2):int,0:int) \/
[ex v(3):int,[ex v(4):int,[ex v(5):seq(int),
                  false /\ sum_s(v(5):seq(int),v(3):int)]]])]]
```

The new theorem is then converted to the corresponding non-ground form in order to be presented to the user. That is,

$$[\ \forall x1/seq(Z),[\ \ \forall x2/Z,sum^{S}(x1,x2) \quad \leftrightarrow \quad (x1=<> \quad \wedge x2=0 \quad \vee \quad [\exists x3/Z,[\ \exists x4/seq(Z),$$
$$false \wedge sum^{S}(x4,x3)]]])]]$$

5. System interface

To enable users to guide this proof checker it is necessary to provide a well-designed user interface. The design of the interface of an interactive verifier depends on the intended user. In our verifier we distinguish two kinds of users, the "*basic users*" and the "*advanced* or *experienced users*". We call "*basic user*" a user who is interested in proving a theorem. We call an "*advanced user*" a user who in addition to proving a theorem he/she may want to enhance the KB of the system in order to be able to deal with additional theorems. Such a user is expected to be able to update the KB of axioms, lemmas, predicate specifications, specifications of DT operations and programs. We will use the word "*user*" to mean both the "*basic user*" and the "*advanced user*". Both kinds of users are expected to know very well the correctness method which is supported by our system (Marakakis, 2005).

Initially, the system displays the main, top-level window as shown in Fig. 1. This window has a button for each of its main functions. The name of each button defines its function as well, that is, "*Transform Logic Specification into Structured Form*", "*Prove Program Correctness*" and "*Update Knowledge Base*". The selection of each button opens a new window which has a detailed description of the required functions for the corresponding operation. Now we will illustrate the "*Prove Program Correctness*" function to better understand the whole interaction with the user.

5.1 Interface illustration of the "Prove Program Correctness" task

If the user selects the button "*Prove Program Correctness*" from the main window, the window shown in Fig. 6 will be displayed. The aim of this window is to allow the user to select the appropriate theory and proof scheme that he will use in his proof. In addition, the user can either select a theorem or define a new one.

After the appropriate selections, the user can proceed to the actual proof of the theorem by selecting the button "*Prove Correctness Theorem*". The window that appears next is shown in Fig. 7. The aim of this window is to assist the user in the proof task. The theorem to be proved and its logic specification are displayed in the corresponding position on the top-left side of the window. This window has many functions. The user is able to choose theory elements from the KB that will be used for the current proof step. After selection by the user of the appropriate components for the current proof step the proper inference rule is selected and it is applied automatically. The result of the proof step is shown to the user. Moreover, the user is able to cancel the last proof step, or to create a report with all the details of the proof steps that have been applied so far.

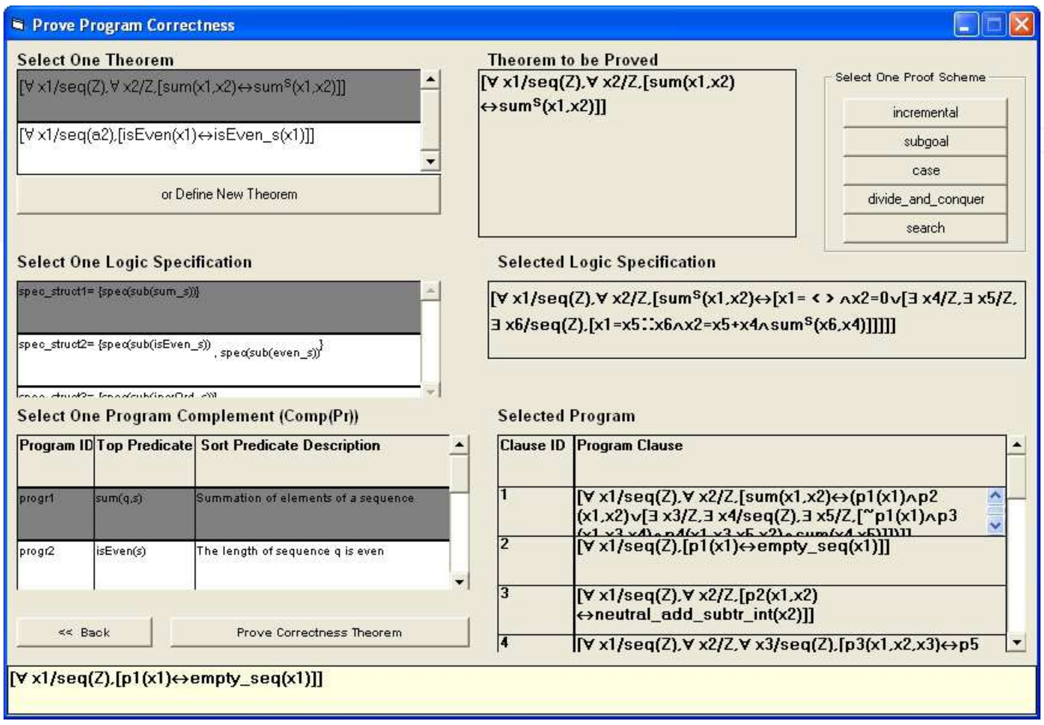
5.1.1 Illustration of a proof step

Let's assume that the user has selected a theorem to be proved, its corresponding theory and a proof scheme. Therefore, he has proceeded to the verification task. For example, he likes to prove the following theorem:

$$Comp(Pr) \cup Spec \cup A \;|=\; \forall x1/seq(Z),\, x2/Z\ (sum(x1,x2) \leftrightarrow sum^S(x1,x2))$$

The user has selected the "*Incremental*" proof scheme which requires proof by induction on an inductive DT. Let assume that the correctness theorem has been transformed to the following form:

$$\forall x1/seq(Z),\ \forall x2/Z,\ sum(x1,x2)\ \leftrightarrow [\exists x3/Z,\ \exists x4/Z,\ \exists x5/seq(Z),$$
$$x1 \neq <> \wedge x1=x4::x5 \wedge x1 \neq <> \wedge x1=x4::x5 \wedge x2=x4+x3$$
$$\wedge\ sum(x5,x3)]$$



Select One Theorem

[∀ x1/seq(Z),∀ x2/Z,[sum(x1,x2)↔sum^{S}(x1,x2)]]

[∀ x1/seq(a2),[isEven(x1)↔isEven_s(x1)]]

Theorem to be Proved

[∀ x1/seq(Z),∀ x2/Z,[sum(x1,x2) ↔sum^{S}(x1,x2)]]

Select One Logic Specification

spec_struct1= {spec(sub(sum_s))}

spec_struct2= {spec(sub(isEven_s)) , spec(sub(even_s))}

Selected Logic Specification

[∀ x1/seq(Z),∀ x2/Z,[sum^{S}(x1,x2)↔[x1= < > ∧x2=0∨[∃ x4/Z,∃ x5/Z, ∃ x6/seq(Z),[x1=x5::x6∧x2=x5+x4∧sum^{S}(x6,x4)]]]]]

Select One Program Complement (Comp(Pr))

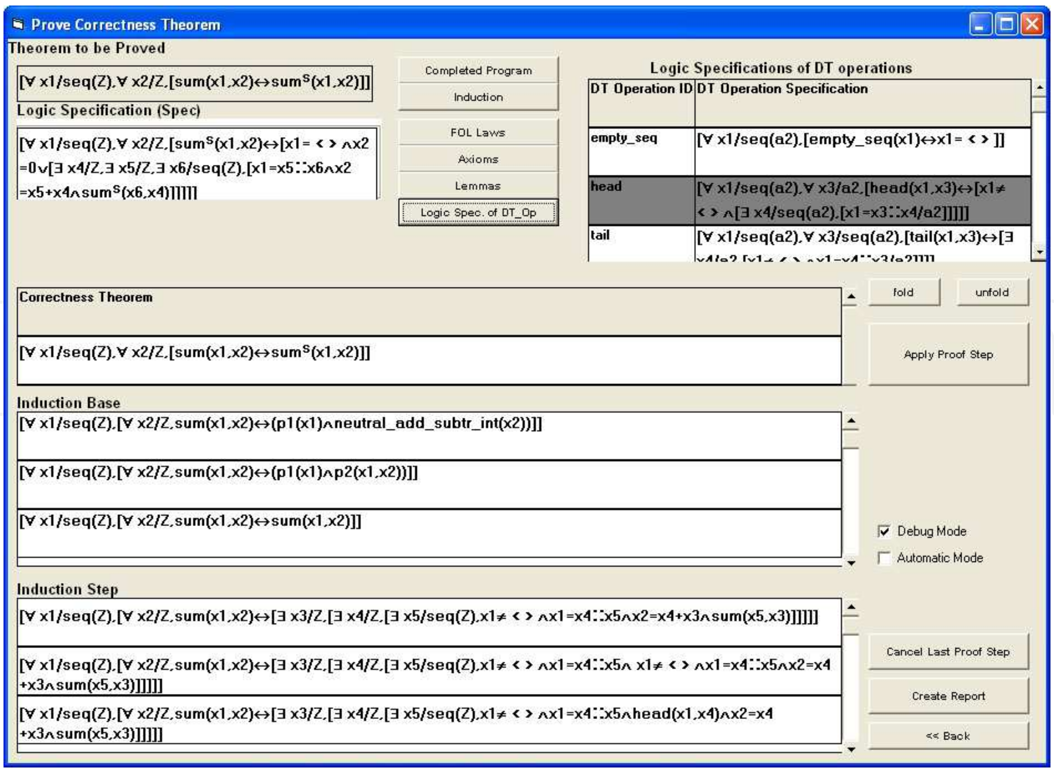
Program ID	Top Predicate	Sort Predicate Description
progr1	sum(q,s)	Summation of elements of a sequence
progr2	isEven(s)	The length of sequence q is even

Selected Program

Clause ID	Program Clause
1	[∀ x1/seq(Z),∀ x2/Z,[sum(x1,x2)↔(p1(x1)∧p2 (x1,x2)∨[∃ x3/Z,∃ x4/seq(Z),∃ x5/Z,[~p1(x1)∧p3 (x1,x3,x4)∧p4(x1,x3,x5,x2)∧sum(x4,x5)]]]]
2	[∀ x1/seq(Z),[p1(x1)↔empty_seq(x1)]]
3	[∀ x1/seq(Z),∀ x2/Z,[p2(x1,x2) ↔neutral_add_subtr_int(x2)]]
4	[∀ x1/seq(Z),∀ x2/Z,∀ x3/seq(Z),[p3(x1,x2,x3)↔p5

[∀ x1/seq(Z),[p1(x1)↔empty_seq(x1)]]

Fig. 6. The window for selecting Theory, Theorem and Proof Scheme



Theorem to be Proved

[∀ x1/seq(Z),∀ x2/Z,[sum(x1,x2)↔sum^{S}(x1,x2)]]

Logic Specification (Spec)

[∀ x1/seq(Z),∀ x2/Z,[sum^{S}(x1,x2)↔[x1= < > ∧x2 =0∨[∃ x4/Z,∃ x5/Z,∃ x6/seq(Z),[x1=x5::x6∧x2 =x5+x4∧sum^{S}(x6,x4)]]]]]

Logic Specifications of DT operations

DT Operation ID	DT Operation Specification
empty_seq	[∀ x1/seq(a2),[empty_seq(x1)↔x1= < >]]
head	[∀ x1/seq(a2),∀ x3/a2,[head(x1,x3)↔[x1≠ < > ∧[∃ x4/seq(a2),[x1=x3::x4/a2]]]]]
tail	[∀ x1/seq(a2),∀ x3/seq(a2),[tail(x1,x3)↔[∃ x4/a2,[x1≠ < > ∧x1=x4::x3/a2]]]]

Correctness Theorem

[∀ x1/seq(Z),∀ x2/Z,[sum(x1,x2)↔sum^{S}(x1,x2)]]

Induction Base

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔(p1(x1)∧neutral_add_subtr_int(x2))]]

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔(p1(x1)∧p2(x1,x2))]]

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔sum(x1,x2)]]

☑ Debug Mode

☐ Automatic Mode

Induction Step

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔[∃ x3/Z,[∃ x4/Z,[∃ x5/seq(Z),x1≠ < > ∧x1=x4::x5∧x2=x4+x3∧sum(x5,x3)]]]]]

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔[∃ x3/Z,[∃ x4/Z,[∃ x5/seq(Z),x1≠ < > ∧x1=x4::x5∧ x1≠ < > ∧x1=x4::x5∧x2=x4 +x3∧sum(x5,x3)]]]]]

[∀ x1/seq(Z),[∀ x2/Z,sum(x1,x2)↔[∃ x3/Z,[∃ x4/Z,[∃ x5/seq(Z),x1≠ < > ∧x1=x4::x5∧head(x1,x4)∧x2=x4 +x3∧sum(x5,x3)]]]]]

Fig. 7. The window for proving a correctness theorem

In order to proceed to the next proof step the following steps should be performed:

- First, the user selects "*Logic Spec. of DT_Op*" and then he selects the "*Head*" DT operation:

$\forall x1/seq(a2),\ \forall x3/a2,[head(x1,x3)\ \leftrightarrow [x1\neq <\ > \wedge [\exists x4/seq(a2),[x1=x3::x4/a2]]]]$

- Then he selects the button "*Apply Proof Step*" and the result is shown in the next line of the "*Induction Step*" area:

$\forall x1/seq(Z),\ \forall x2/Z,\ sum(x1,x2)\ \leftrightarrow [\exists x3/Z,\ \exists x4/Z,\ \exists x5/seq(Z),$
$x1\neq<> \wedge x1=x4::x5 \wedge head(x1,\ x4) \wedge x2=x4+x3 \wedge sum(x5,x3)]$

The user continues applying proof steps until to complete the proof of the theorem.

6. Results

The results of this research work involve the development of a proof checker that can be used efficiently by its users for the proof of correctness theorems for logic programs constructed by our schema-based method (Marakakis, 1997). The system has been tested and allows the verification of non-trivial logic programs. Our proof checker is highly modular, and allows the user to focus on proof decisions rather than on the details of how to apply each proof step, since this is done automatically by the system. The update of the KB is supported by the proof-checker as well. The overall interface of our system is user–friendly and facilitates the proof task.

The main features of our system which make it to be an effective and useful tool for the interactive verification of logic programs constructed by the method (Marakakis, 1997) are the following.

- The proof of the correctness theorem is guided by the logic-program construction method (Marakakis, 1997). That is, the user has to select a proof scheme based on the applied program schema for the construction of the top-level predicate of the logic program whose correctness will be shown.
- Proof steps can be cancelled at any stage of the proof. Therefore, a proof can move to any previous state.
- The system supports the proof of a new theorem as part of the proof of the initial theorem.
- The update of the theories stored in the KB of the system is supported as well.
- The overall verification task including the update of the KB is performed through a user-friendly interface.
- At any stage during the verification task the user can get a detailed report of all proof steps performed up to that point. So, he can get an overall view of the proof performed so far.

7. Conclusions

This chapter has presented our proof checker. It has been focused on the knowledge representation layer and on its use by the main reasoning algorithms. Special importance on the implementation of the proof checker has been given on flexibility so the system being developed could be enhanced with additional proof tasks. Finally, the main implementation criteria for the knowledge representation are the support for an efficient and modular implementation of the verifier.

In our proof checker, a proof is guided by the selected proof scheme. The selection of a proof scheme is related with the construction of the top-level predicate of the program that will be verified. The user-friendly interface of our system facilitates the proof task in all stages and the update of the KB. Its modular implementation makes our proof checker extensible and amenable to improvements.

The natural progression of our proof checker is the addition of automation. That is, we intend to move proof decisions from the user to the system. The verifier should have the capacity to suggest proof steps to the user. Once they are accepted by the user they will be performed automatically. Future improvements aim to minimize the interaction with the user and to maximize the automation of the verification task.

8. References

Clarke, E. & Wing, J. (1996). Formal Methods: State of the Art and Future Directions, *ACM Computing Surveys*, Vol. 28, No. 4, pp. 626-643, December, 1996.

Gallagher, J. (1993). Tutorial on Specialization of Logic Programs, *Proceedings of PERM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 88-98, ACM Press, 1993.

Hill, P. M. & Gallagher, J. (1998). Meta-programming in Logic Programming, *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, edited by D. Gabbay, C. Hogger, J. Robinson, pp. 421-497, Clarendon Press, Oxford, 1998.

Hill, P. M. & Lloyd, J. W. (1994). The Gödel Programming Language, The MIT Press, 1994.

Lindsay, P. (1988). A Survey of Mechanical Support for Formal Reasoning, *Software Engineering Journal*, vol. 3, no. 1, pp.3-27, 1988.

Lloyd, J.W. (1994). Practical Advantages of Declarative Programming. In Proceedings of the Joint Conference on Declarative Programming, GULP-PRODE'94, 1994.

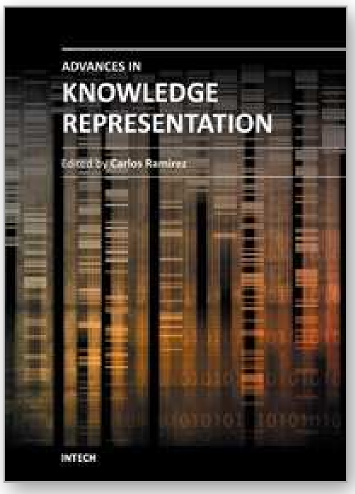
Loveland, D. W. (1986). Automated Theorem Proving: Mapping Logic in AI, *Proceedings of the ACM SIGART International Symposium on Methodologies for Intelligent Systems*, pp. 214-229, Knoxville, Tennessee, United States, October 22-24, 1986.

Marakakis, E. (1997). Logic Program Development Based on Typed, Moded Schemata and Data Types, *PhD thesis*, University of Bristol, February, 1997.

Marakakis, E. (2005). Guided Correctness Proofs of Logic Programs, *Proc. the 23th IASTED International Multi-Conference on Applied Informatics*, edited by M.H. Hamza, pp. 668-673, Innsbruck, Austria, 2005.

Marakakis, E. & Gallagher. J.P. (1994). Schema-Based Top-Down Design of Logic Programs Using Abstract Data Types, *LNCS 883, Proc. of 4th Int. Workshops on Logic Program Synthesis and Transformation - Meta-Programming in Logic*, pp.138-153, Pisa, Italy, 1994.

Marakakis, E. & Papadakis, N. (2009). An Interactive Verifier for Logic Programs, *Proc. Of 13th IASTED International Conference on Artificial Intelligence and Soft Computing* , pp. 130-137, 2009.



Advances in Knowledge Representation
Edited by Dr. Carlos Ramirez

ISBN 978-953-51-0597-8
Hard cover, 272 pages
Publisher InTech
Published online 09, May, 2012
Published in print edition May, 2012

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821