NIFS 10 (2004), 1, 27-36

AN ATTEMPT TO BUILD AN INTUITIONISTIC FUZZY PROLOG MACHINE
Part 1

K. Atanassov*, M. Marinov, Z. Zlatev*****

* CLBME-BAS, Bulgaria, E-mail: krat@bas.bg
** FabLess Ltd (Bulgaria), E-mail: marin.marinov@fab-less.com
*** University of Twente (Holland), E-mail: Z.V.Zlatev@ewi.utwente.nl

The first attempt to build an intuitionistic fuzzy Prolog was made in the end of 80-ies and the beginning of the 90-ies (see [3,9]). For number of reasons the work lead only to a prototype, but it was a finished version as well. The present wok is a new development of an intuitionistic fuzzy Prolog, reflecting the new inventions in the theory of intuitionistic fuzzy sets (see [1,5]) and intuitionistic fuzzy logic. [2,4-8] – the built-in modal, topological and level operators are examples. The present version gives a possibility to interpret an aim in conditions of intuitionistic fuzzy, simple fuzzy or pure Prolog.

The existing till now solutions of the task: to build a means for automatic proof of theorems or to make an automatic inference of a new knowledge from a given knowledge base, already collected, were spread mostly over the classical logics and the normally fuzzy one. The classical Prolog solves problems from the classical logics, but it's helpless in cases of unsure information (something is known with a certain probability) or paradoxes. The fuzzy Prolog is quite successful with this class of tasks, but it's not powerful enough to deal with incomplete information tasks. The surrounding world cannot be successfully modeled even with the "compromises" in the fuzzy Prolog. The knowledge about the surrounding world is quite often unknown to a certain extent. And that is exactly the aim of the present work – to make a Prolog machine, operating with facts and rules, having a degree of uncertainness in their nature. Such a machine should use the achievements of the theory of intuitionistic fuzzy sets and the intuitionistic fuzzy logic. In order to use completely the power of the intuitionistic fuzzy logic and to be able to freely transform and modify the degree of truthfulness of a given statement, it is necessary to build-in operators without analog in the other realizations. These are the modal operators, whose realization and use will be discussed below.

General description of the language

As a starting point for building an intuitionistic fuzzy Prolog have been used the classic version of Prolog [10] (see also [11-14]) and the achievements of the theory of intuitionistic fuzzy logic, i.e. a program language has been defined, called Deep Thought Prolog or DTProlog, on the basis of the classic Prolog with added syntactic possibilities for setting intuitionistic fuzzy values. For the purpose of compatibility it supports all constructions, defining synonym ones as well. The synonyms have been built-in for more flexible programming and to give a possibility to choose symbols abstracting a given logical operation. The structure of the program has been kept, giving an additional liberty in the organization of the blocks – repetitions, empty blocks and rearrangements are allowed. The fundamental setting-up of the classical Prolog consists in the added constructions for associating reliability degrees to the rules and facts. A modifier can be added to each fact, in which truth degrees are set as fractional numbers. The modifier or part of it can be omitted and in this case it is automatically filled by default values. The reliability change of the rules is done during execution with the help of the already mentioned modal operators. The modal

operator is a new syntactic construction which, depending on its type, allows modification of the rule head assessment value during interpretation of the given goal. The two most widely known modal operators are built in the present version: NECESSARY and POSSIBLE. In their essence they change the truthfulness degree transferring it in the terms of purely fuzzy logic (they increase or decrease the truthfulness of a given statement with the value of uncertainty.

Interpreter

A compiling interpreter of the language described above has been made, i.e. a given program is firstly compiled, and then the data received from it are interpreted. The compilation stage allows greater flexibility of the permitted syntactic constructions, guarantees a syntactically correct interpreting program, higher interpretation speed, settingness of the data for interpretation and including of additional elements to a compiler, as pre-processor. The present version can interpret a program in the terms of the classic, fuzzy or intuitionistic fuzzy Prolog. A new compilation is necessary each time the type of the Prolog has been defined, so that correct default values could be associated or to be set such according to the rules of another Prolog. The attempts to satisfy the goal could be repeated several times after the compilation, the interpretation works on specific data, taken from the text of the program. The built-in pre-processor gives the following possibilities for text-processing on the outgoing source of the program:

1. Macroprocessor. It gives a possibility similar to that of the macroprocessor for languages as C and C++. A string is associated to a given string, replacing the first one before the real compilation.

2. Including files. With the use of the command #include the programmer is given a possibility to structure the program in modules, to improve its readability and to focus on a given problem.

3. Conditional compilation. With the commands for conditional compilation different versions of a program could be supported or the program could be modified depending on given setting constants.

Fuzziness modifiers

This is a language construction allowing associating of a fuzzy value to a give fact. To each statement is appropriated a couple of digits, corresponding to the conditions set by the theory of intuitionistic fuzzy logic. If the modifier or part of it has been omitted, default values are put in the empty places by the compiler. The syntaxes in Backus-Naur notation of the fuzzy modifier is as follows:

[< [decimal fraction constant] , [decimal fraction constant] >]

where [ ] stands for an optional element

The default values are: [<[0.0],[1.0]>]

With these values is obtained a complete compatibility with the classic and fuzzy Prolog.

Declarations and data types

The rules and fact follow a common definition scheme. A fact is defined as follows: a predicate with its corresponding variables and constants (arguments, their number could be zero) is written on a separate line, and its intuitionistic fuzziness is done with the help of a modifier at the end of the declaration.

Examples:

Is("John","boy").

Is("John","handsome").<.5,0.2>

The inference rules are analogous to those in the classic Prolog – they are not modified with fuzzy values. In order to combine more than one predicates in logic functions the simple Prolog connection symbols are used to form the rule, but they have one more duplicate. ‘,’and ‘&’ have the meaning of logical AND. ‘;’ and ‘ | ’ have the meaning of logical OR. ‘:-’ and ‘-’ are used to represent the implication. A rule in DTProlog consists of: a rule head, a symbol for the rule type and a rule body. The head follows the syntactic rules defined for facts. The symbol defining the type of the rule is one of the following: ‘:-’, ‘<-’, ‘<>-’, ‘[]-’. Each one of them defines the way of calculating the intuitionistic fuzzy head value. When using ‘:-’ and ‘<-’ the value of the body is appropriated to the head. In the case of a rule of the type ‘<>-’ – the value representing the falseness of the statement is transferred to the head without changes, but the value representing the truthfulness is 1 minus this of falseness. The modal operator POSSIBLE is realized with the help of this rule. In practice this is a transfer of intuitionistic fuzzy value in the terms of a simple fuzzy one. In the case of a rule of the type ‘[]-’ – the value representing the truthfulness of the statement is transferred to the head without changes, but the value representing the falseness is 1 – this of truthfulness. The modal operator NECESSARY is realized with the help of this rule. The body of the rule is a combination of predicates and symbols for logic operations.

Is(someone,"clever"):-Is(someone,"A-student"),
Likes(someone,"reading") | Is(someone,"genius").

Structure of the program

The DTProlog program is a sequence of sections in a random range and without limitation of the number of appearences (which differs from the classic Prolog where the sections have a fixed range and number of apparitions in a program). The sections can be 6 types. The beginning of a section is marked with a corresponding keyword, and its end is beginning of another section. The sections are:

Section constants. The standard Prolog provides a possibility for parameterization of the program text in a way analogous to the define command. In DTProlog the contents of this section is not taken into account.

Section domain. In the standard Prolog this section gives a possibility to define new types of derivatives of the built-in ones. DTProlog has limited possibilities for work with data types and for that reason the contents of this section is not taken into account.

Sections database и predicates. These sections are also empty, because they suppose "typisation" of the possible constructions in sections clauses and goal, but for DTProlog such limitations are not necessary. There is no need of tracking the parameters type for each predicate, as there is only one type and tracking of the number is enough as condition.

Section goal. It is used to set a goal to the interpreter. The specification is a little bit modified as compared to the classic Prolog in order to include the new possibilities to set a goal including modality. The goal is an inference rule, whose intuitionistic value is represented as a result. One DTProlog can have many goal sections, but only one goal is permitted.

Section clauses. This is the section where all facts and rules are described. The description follows a syntax given in point Declarations and data types, and each fact or rule is written on a separate line (this is not a requirement of the compiler).

Extensions

The differences compared to the standard Prolog consist in the more complex syntactic diagrams allowing intuitionistic fuzzy representation of the facts and rules. The possibility to associate an intuitionistic fuzzy value to each statement differs not only from the classic Prolog, but from all other fuzzy Prologs. The modal operators used are also without analog in

the other Prolog modifications. The doubling of the logic operations allows a choice of a specific symbol among several possible ones. The preprocessor processing is unique for DTProlog. It is characteristic for the structural compiling programming languages, but because of the presence of a compiling pass in DTProlog, a processor, with the potentialities stated above, has been introduced for the sake of convenience and facilitating the programming.

Limitations

Only one limitation has been imposed – only one data type – string. The limitation itself does not reduce the *community of* validity and functionality, but for some programs it can substantially increase the code and the execution time.

Intuitionistic fuzzy Prolog operation algorithm

The operation algorithm of the intuitionistic fuzzy Prolog follows exactly the same steps, with an additional complication to calculate intuitionistic values at each step. The data structure has been also complicated. To each element from the list representing the goal an additional element has been associated to represent intuitionistic fuzzy value. One more element has been added to stand for the depth (the distance to the root of the tree, representing the resolution) for greater convenience in calculating the value of a given head rule.

Structure and data presentation

As it was already seen Prolog is a language with a list structure as regards each data structure in it, that's why, as shown below, the list has been chosen as the most suitable structure for data presentation. For the needs of the backtracking as a data structure, the stack should be also present, and for optimizing the work with the variables, unique within one rule, fact or aim, we use a hash-table.

You'll find here below detailed information about the data structures used. For greater clarity and exactness examples from the source of DTProlog will be given in Microsoft Visual C++ syntax. One after the other will be presented: The data (knowledge) base, the goal to be satisfied, the backtracking stack, the variables hash-table.

Data (knowledge) base

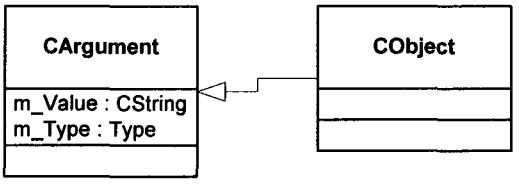
The hierarchic structure of the inserted lists data base will be built, starting from the most inner one. Let's try to present one predicate. It's composed of a head and list of arguments.

Example:

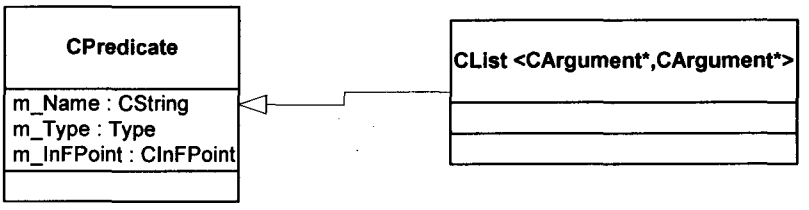
Group_is('Peter", "Ann", "Joni", "Mery").

Lives_in(some_one, "house").<.4,.3>

As an abstraction of the predicate argument we use an object with one data field for value and one for the characteristic value/constant of each argument. The object so formed CArgument is of the following type:



We form an object CPredicate to represent one predicate. It is a list of pointers to objects of the type CArgument plus several data fields to store the name of the predicate and its intuitionistic fuzzy value. A separate object CInFPoint is used to represent the fuzziness, disregarding it. The object CPredicate is of the following type:

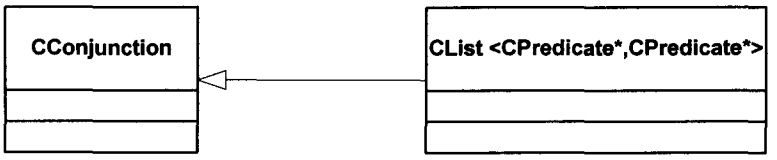


The object CPredicate thus differentiated forms lists CConjunction to represent the possible predicate conjunctions in one clause.

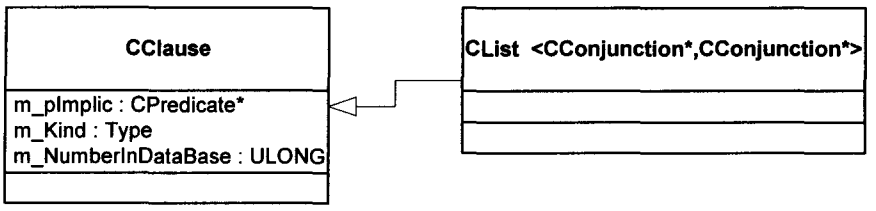
Example:

Clause<-Group_is('Peter", "Ann", "Joni", "Mery")| Lives_in(some_one, "house").

CConjunction is list of pointers to predicates and is of the type:



One clause, which is a deduction rule or fact, is again presented as a list, this time, of pointers to conjunctions (objects CConjunction). Some fields are added to the list forming the object CClause: head of the clause – a separate pointer to a predicate; type of the clause – ordinary with built-in modality; number of the clause in the entire data base. It is of the type:

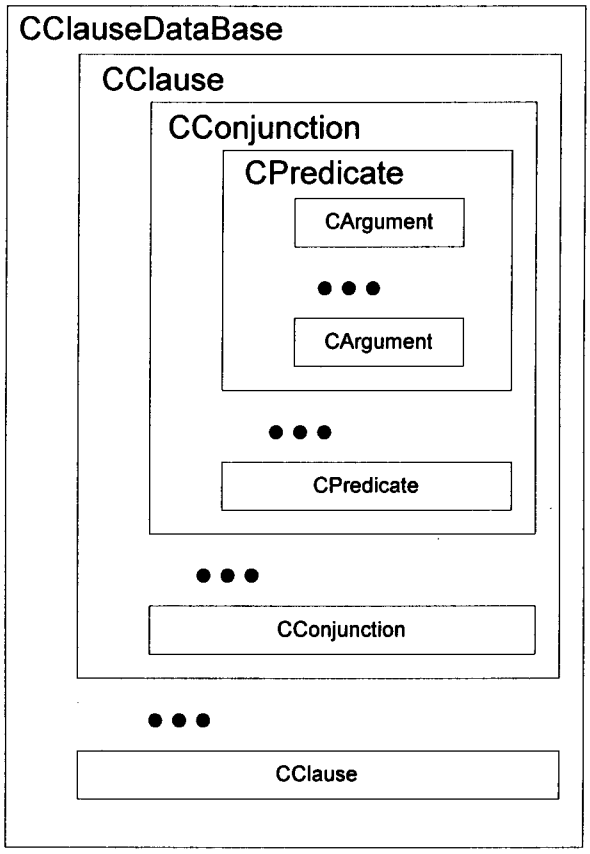


The last level of abstraction of the lists is the whole data base, represented by list of pointers to clauses (objects CClause).

The hierarchic structure of the data (knowledge) base is shown on figure below.

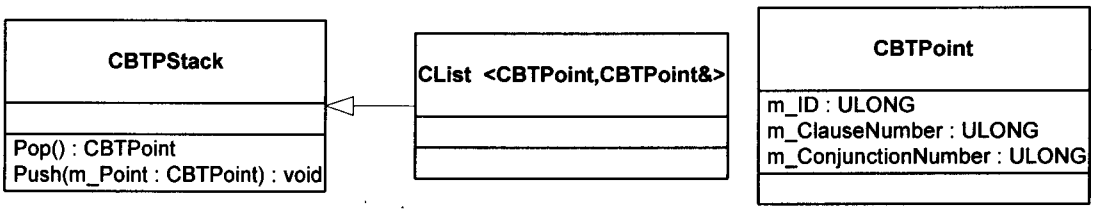
Goal for interpreting

The goal for interpreting is also represented by a list – a list of conjunctions to be satisfied, and the best representation is by an object CClause, although not all of its potentialities are used.



Backtracking stack

The operation algorithm of a Prolog machine requires a return backwards to make an attempt to re-satisfy the goal. In order to obtain synchronization between the goal proving process and the variables status at each point of the prolog machine operation, the list is again used as a structure, but this time organized in a stack. A new value is written (i.e. added) in it each time when the goal or the hash-table has been changed. An element of the stack is taken out each time a return back is necessary and depending on the "write" in this element the information in the goal and the links in the hash-table are updated. The elements of this stack are CBTPoint objects and carry information about the event's uniqueness, number of modifications in the goal (quantity and number of conjunctions and predicates), unique number for changes in the hash table. Appearance of the steak and CBTPoint:



Hash-table

In order to optimize the representation of the variables characteristics in a Prolog program they are kept in a hash-table. As one variable is one and the same thing only within one clause, when numbering the clauses the names of the variables in them are modified with the number of the clause. The result are unique variables' names and these names can be used as keys in the hash-table. The elements in the hash-table are objects, in which is kept

information about: state of the variable – free, connected(dependent), unified; value of the connected one, name of the unified; moment of connection (the moment of connection is presented by object CBTPoint –a "momentary shot" of the Prolog machine state. This structure allows the organization of inner, non-interfering with each other, lists in the hash-table. If there is a sequence of unifications, i.e. var1 is unified with var2, and var2 with var3, a list of unifications will be generated within the hash-table itself.

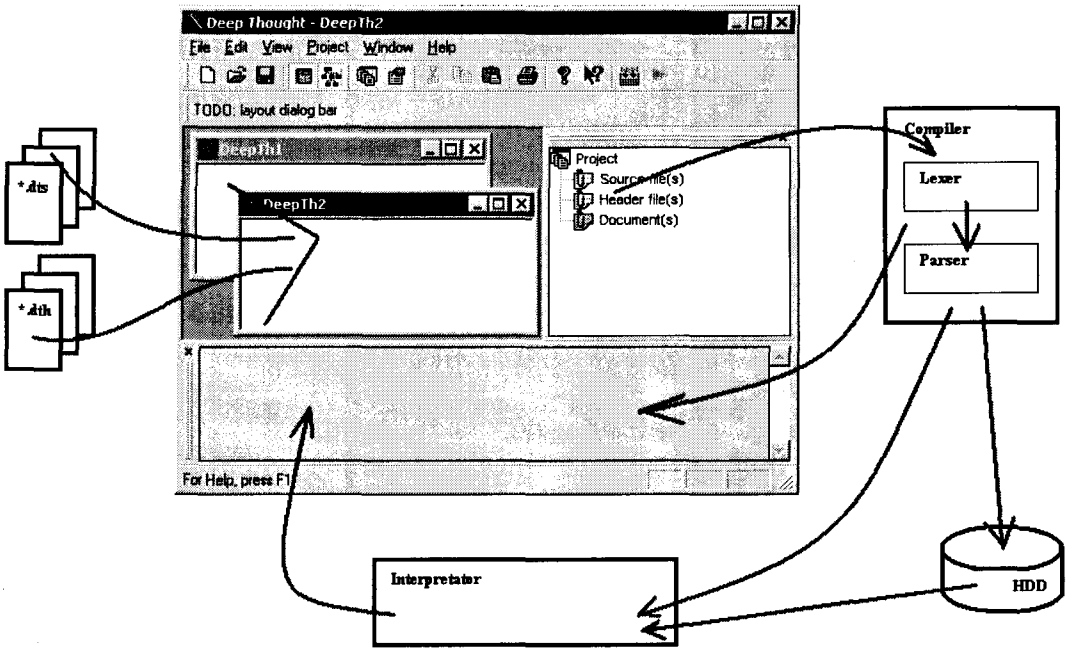
Additional data

The intuitionistic fuzzy values are real numbers within the interval [0,1], but to avoid the float numbers in order to accelerate the work of the Prolog, they are presented as numbers with fixed point, starting from 0.0000 to 0.0100. The precision of the presentation can vary and is set during the compilation, setting aside a memory of 8 byte, which gives an accuracy of 63 digits after the point.

In order to facilitate the calculation of the intuitionistic fuzzy values of each rule, a special marking technique is used on the tree, along which a goal proof is "let down". Each unification of part of the goal with a clause from the knowledge base engenders new leaves (potential branches) in the tree and they get a number greater than the parent one and represent distance to the root. Reducing the leaves to their parent knot necessitates calculation of the parent value. It can be obtained by applying AND function (not depending on whether the variant is fuzzy or not) to all its children, identified by their unique number.

Deep Thought building Technologies

The practical part of the present work, namely the Deep Thought application, is done in programming language C++ in Microsoft Visual Studio 6.0 environment. The application can be conditionally separated in three parts: interface to the user, syntactic analyzer and interpreter. Information will be given about the building of each of its parts in a section discovering the programming technologies and techniques. The show supposes acquaintance with the application block diagram given below:



Interface

The main characteristics and appearance of the application Deep Thought are given by the Wizard used for automatic generation of MDI application. The Wizard mentioned takes the task to structure the application, including: Document-View architecture – proposing independence of the data from their appearance on the screen, automatically generated menu with the main functions included in it; tool-bar with doubling for faster and easier operation, functions from the menu, automatically generated help for all standard functions. The rest non-specific part for an ordinary application, is manually programmed or supported by the visual programming means included in the Visual Studio 6.0. This includes: its appearance and functionality, for project; its appearance and servicing, for messages; all additional visual components for setting and starting a DT application.

Syntactic analyzer

The syntactic analyzer could be divided into: Preprocessor, Lexical analyzer, Syntactic analyzer, Semantic analyzer and Data base operation. The Preprocessor is included in the lexical analyzer and it is its semantic analyzer. The Lexical analyzer is automatically generated with the help of Flex – an instrument developed in the Barkley University, allowing automatic generation of a lexical analyzer for any language according to description of a specific one. This language allows mixing of C and C++ languages, representing the semantic analysis of the recognized constructions. The semantic analysis within the lexical analyzer is manually generated. The result of Flex is C or C++ source code, subject to additional compilation. The syntactic and semantic ones are analogous to the Preprocessor and the lexical analyzer. For the programming of the syntactic analyzer is used BISON (the inheritor of the well known compiler generator YACC), using the Flex generated "lexer" and analogically generates C or C++ program recognizer upon given specific language grammar. The semantic analyzer is manually made and is included in the syntactic one. The Data base operation is extracting from the DT application text the necessary information for its interpretation. The data are handled mainly by the inheriting the class collections provided by the MFC library.

The whole interpreter from the goal satisfaction algorithm to the operation with the compiler extracted structure is entirely a proper development.

The Project is an hierarchically organized structure for keeping the information about the current task, that we'll call DT application. For one DT application during operation of the DTProlog the following application is kept. Type of interpretation – which can be classic prolog, fuzzy prolog or intuitionistic fuzzy prolog. Files that compose the project –these are just the files that are hierarchically arranged. The main level is that of the project, which has got three sub-levels – one for source files, one for additional files and one for documentation files. Each of the listed levels can contain (as a third level in the tree) an unlimited number of files of the corresponding type.

The messages view allows the user to interactively follow the operation of the program. It displays messages for the expected operations from the part of the user, explanations for rejected operations, eventual errors, program status, results from the operation of the compiler and interpreter.

The program provides a panel with icons for fast and convenient work. The potentialities of the panel double those of the menus, namely icons for project editing, as creating, adding files, determining the type; icons for editing the program text of the DT application, as printing, copying, inserting, cutting text; icon for compiling; icon for interpreting, icon for help.

Deep Thought potentialities

Deep Thought is a completely efficient application with a standard programming environment functionality according the present day views. It provides an intuitive Windows style user interface. Its good structuring and program components hierarchy in combination with a non-structured programming language give a different approach to the task assigned.

The modeling potentialities of Deep Thought are considerably greater than those of the Prologs known until now. This is due to the more powerful theory based in the deduction rules. The fact that DProlog works with fuzzy statements including uncertainness, makes it a generalization of the existing fuzzy Prologs, and from there to the classical one as well. What is more, the settings of DTProlog makes it universal for solving tasks with any fuzziness. The user, without having to additionally rewrite or correct the program, is able to examine its behaviour in the case of any of the three fuzziness (without fuzziness, pure fuzziness and intuitionistic fuzziness).

The proposed characteristics accompanying the result, as space and time dimension of the task being solved, make the extracted information still more exhaustive and the analysis – deeper. Using the execution time and/or the space of memory used a theoretical estimation can be made whether the greater time for solving a fuzzy task is compensated by the precision of the received information.

The theory on which is based the DTProlog gives possibilities for solving such kind of tasks that are otherwise difficult to solve or unsolvable. Paradoxes of the classical logic are solvable even in the terms of fuzzy Prolog, and consequentially by the DTProlog. The main advantage, as compared to the other means for automatic theorem proving or new knowledge extraction, is the possibility to handle facts and rules with incomplete information. This makes it particularly useful in almost all artificial intelligence spheres, as: object classification, decision support systems, expert information processing and next generation expert systems.

Conclusion

In the first part of the present paper it has been described Intuitionistic Fuzzy Prolog containing the first order logic operation and the main modal operators. In a subsequent authors' studies there shall be discussed the extensions of the modal operators, of level operators and others.

References

1. Atanassov K., Intuitionistic Fuzzy Sets, Fuzzy Sets and Systems Vol. 20, 1986.
2. Atanassov K., Two Variants of Intuitionistic Fuzzy Propositional Calculus. Preprint IM-MFAIS-5-88, Sofia, 1988.
3. Atanassov K., Intuitionistic Fuzzy Prolog. Preprint IM-MFAIS-5-89, Sofia, 1989.
4. Atanassov K., Two Variants of Intuitionistic Fuzzy Modal Logic. Preprint IM-MFAIS-3-89, Sofia, 1989.
5. Atanassov K., Intuitionistic Fuzzy Sets. Springer Physica-Verlag, Berlin, 1999.
6. Atanassov K., Gargov G., Intuituinistic Fuzzy Logic. Comptes Rendus de l' Academie bulgare des Sciences, Tome 43, 1990, No. 3, 9-12.
7. Atanassov K., Gargov G., Elements of Intuitionistic Fuzzy Logic. I. Fuzzy sets and Systems Vol. 95, 1998, No. 1, 39-52.
8. Atanassov K., Gargov G., Two Results in Intuitionistic Fuzzy Logic. Comptes Rendus de l' Academie bulgare des Sciences, Tome 45, 1992, No. 12, 29-31.

9. Atanassov K.,
10. Borland Inc., TURBO PROLOG User's Guide version 2.0, 1988
11. Chang Ch., Lee R., Symbolic Logic and Machanical Theorem Proving. Academic Press New York San Francisco London, 1973
12. Davis M., Putnam H., A Computing Procedure for Quantification Theory. J. Assoc. Comput. Mach., 1960
13. Gilmore P. C., A Proof Method for Quantification Theory: Its Justification and Realization. IBM J.Res. Develop., 1960
14. Robinson J.A., The generalized resolution principle. Machine Intelligence, 1968