1991

M.I. Kanovich

Fast theorem proving in intuitionistic propositional logic

Computer Science/Department of Software Technology Report CS-R9157 December

Fast Theorem Proving in Intuitionistic Propositional Logic

Max I. KANOVICH
Tveŕ State University, Tveŕ 170000, USSR and
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

The decision problem for Intuitionistic Propositional Logic **Int** is considered:

(i) A computational semantics is introduced for relational knowledge bases. Our semantics naturally arises from practical experience of databases and knowledge bases.
It is stated that the corresponding logic coincides exactly with the intuitionistic one.

(ii) Our methods of proof of the general theorems turn out to be very useful for designing new efficient algorithms.
In particular, on the basis of a **specific Calculus of Tasks** related to this computational interpretation, an efficient prove-or-disprove algorithm is designed with the following properties:

- For an arbitrary intuitionistic propositional formula, the algorithm runs in linear deterministic space,
- For every 'reasonable' formula, the algorithm runs in 'reasonable' time, despite of the fact that in theory it has an 'exponential' uniform lower bound.

Note that in view of the PSPACE-completeness of Propositional Intuitionistic Logic an exponential execution time can be expected in the worst case. But such cases only arise for very unnatural formulas, i.e., for formulas that even in their best solutions need maximal cross-linking of all their possible subtasks.
The theorem prover has been implemented in PASCAL.

Mathematics Subject Classification: Primary 68P15; Secondary 68N05.
CR Categories: F.2.2, F.3.1, I.2.3, I.2.4.
Keywords and Phrases: Theorem Proving, Complexity, Intuitionism.

Note: The author was sponsored by project NF 102/62-356 ('Structural and Semantic Parallels in Natural Languages and Programming Languages'), funded by the Netherlands Organization for the Advancement of Research (N.W.O.).

Report CS-R9157
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Contents

1 What we used to have before

Let us observe some problems that have arisen in relation with Intuitionistic Propositional Logic **Int**.

1.1 Intuitionistic Logic as a Logic of Tasks

Starting with **A.Kolmogorov** and **A.Heyting** there were many attempts to interpret **Int** as a logic of tasks. **S.Kleene** and **Ju.Medvedev** were probably the first who proposed an explicit formalization for **Int**. Unfortunately, all known natural formalizations have led to logics that are essentially stronger than **Int**. Moreover, for the propositional logic of Kleene's realizability, nobody knows whether this logic is decidable or not.

When I started this study in relation with the problem of so-called program synthesis, I (like the most mathematical logicians) was convinced that the corresponding logic would be stronger than **Int** and, probably, undecidable. [1]

It should be noted that most troubles are originating from **implication and disjunction.**

1.2 The decision problem for Int

Exponential time. It is well-known that **Int** is PSPACE-complete [Statman 79]. Moreover, we have a uniform 'exponential' lower bound, i.e., almost all formulas may require exponential time to be recognized.

Space of the 4th degree. It follows from results of [Ladner 77] that **Int** can be solved in space

$$O(L^4),$$

where L is the size of a formula.

As for the decision complexity, most troubles are originating also from implication and, especially, disjunction.

2 What we have got now

Let us present our results related to **Int**.

2.1 The precise computational interpretation

In contrast to what I had expected, I have failed to construct a 'computational task' that requires super-intuitionistic rules.

[1] *See comments of G.Mints [Mints 83] with respect to the system of automatical program synthesis PRIZ.*

Moreover, to my surprise, I have been able to prove that **Int** is complete under a computational interpretation related to relational databases.

Justifying this statement, a specific **Calculus of Tasks** has been invented along with the corresponding Completeness Theorem.

2.2 Efficient decision algorithms

As a matter of fact, just the same Completeness Theorem yields very practical consequences for running time and space:

Linear deterministic space. We can recognize **Int** in linear deterministic space

$$O(L).$$

It should be pointed out that such space is sufficient for **the full set** of intuitionistic connectives.

Quasipolynomial time. We have established **a non-uniform upper bound** for running time:
For a given propositional formula f, we can recognize whether it belongs to **Int** or not, in running time, approximately,

$$L^{r+1}$$

where r is the measure of 'unnaturalness' of formula f:
even in its best solutions f needs cross-linking of, at least, r 'subtasks'.

3 The means that were used

Now, I am going to present the main ideas of our computational interpretation:

- A formula is considered as **an entity, or quantity.** The domain of such an entity may be infinite.
- An entity that represents an implication may be considered as an entity of 'functional' type, a possible value of such a 'functional entity' **should be a program.**
- Possible values of all the entities are collected into **a database,** relations between the entities are perceived as **constraints, or dependencies,** for it.
- Justifying a formula means that the corresponding dependency is satisfied on all 'admissible' databases.

The following notational conventions are followed throughout this paper:

(I) By

$$A\text{s},\ B\text{s and } C\text{s}$$

we denote positive literals, or names of 'entities'.

(II) By

$$Xs,\ Ys,\ Ws \text{ and } Zs$$

we will denote conjunctions of a number of positive literals.

(III) The **'empty' conjunction** is denoted by $\emptyset$.

3.1 From Formulas to Tasks

Each propositional formula can easily be rewritten in the following form:

$$\Gamma \vdash Z \ ,$$

where Γ is a multiset consisting of formulas of one of the following three **basic forms:**

(a) $(X \to Y)$,

(b) $((X_1 \to Y_1) \to Y)$,

(c) $(X \to (Y_1 \ or \ Y_2))$.

Such **a Task Sequent** $\Gamma \vdash Z$ is perceived as a computational task:

Task: Given all laws and dependecies from Γ, compute Z.

Example 3.1 Introducing new names F and G, the propositional formula

$$((A\&B \to C) \to (A \to (B \to C)))$$

can be represented by the following task sequent

$$((B \to C) \to F), (F\&B \to C), ((A \to F) \to G), (G\&A \to F), (A\&B \to C) \vdash G \ .$$

3.2 A formula as a Knowledge base

The left side Γ of a Task Sequent $\Gamma \vdash Z$ contains all informations that can be used for computing Z and, hence, represents, so to say, **a knowledge base,** i.e. the collection of all laws and dependencies related to our problem.

3.3 The formal definition of a Knowledge base

Now, we give a formal definition of a **relational knowledge base:**

A relational knowledge base is a tuple

$$KB = (Names, Functs, Doms, Deps)$$

where

(1)
$$Names = (A_1, A_2, \ldots, A_n, \ldots)$$
is a recursively enumerable sequence of literals or, in other words, names of "entities" (or "attributes").

(2) The set *Names* is divided into two parts: some entities are declared as "functional entities".

Functs is a recursively enumerable set of names of all "functional entities" together with their finite types:
the type of a functional entity F is an expression of the form
$$(X_1 \rightarrow Y_1),$$
this X_1 is called "the argument list".

(3) *Doms* is a recursively enumerable sequence of domains of entities from Names: the domain of entity V is denoted by $Dom(V)$.
(An **empty entity** B such that $Dom(B)$ is empty may be considered).

For X, $Dom(X)$ is the Cartesian product of domains of all entities from X.

The crucial point of the definition is that, for functional entities, we require that in order to specify some concrete value of a functional entity it is not sufficient to give its set-theoretical description; instead, it is necessary to present a program calculating this function. Therefore, the domain of a functional entity F of the type $(X_1 \rightarrow Y_1)$ is defined **to be the set of all programs** mapping $Dom(X_1)$ into $Dom(Y_1)$.

(4) *Deps* is a recursively enumerable set of laws, constraints, dependencies etc. connected with our problem area.

Example 3.2 *(continues Example 3.1)* We may say that there are

(I) two functional entities:
F of the type $(B \rightarrow C)$ and
G of the type $(A \rightarrow F)$, and

(II) one dependency:
$$(A\&B \rightarrow C).$$

3.4 A basic formula as a constraint, or a dependency

Each basic formula is considered as a representation of some constraint:

(a) An implication $(X \rightarrow Y)$ should be perceived as the assertion:

> "There is a COMPUTABLE function (a functional of higher type) from $Dom(X)$ into $Dom(Y)$". [2]

[2]It should be noted that names of functional entities may be contained in X and Y.

(b) An embedded implication $((X_1 \to Y_1) \to Y)$ is treated as the 'simple' implication $(F \to Y)$
where F is a functional entity of type $(X_1 \to Y_1)$.

(c) 'Variant dependency' $(X \to (Y_1 \textit{ or } Y_2))$ should be taken as the assertion:

"For some values of X, the values of Y_1 can be calculated, and, for the rest of the values of X, the values of Y_2 can be computed".

Y_1 and Y_2 are called "alternative lists".

4 What does 'BE SOLVABLE' mean ?

4.1 Relational databases

As models for a knowledge base we consider relational databases:

First, define the notion of a "possible state". We assume that there is a symbol UNDEF that represents undefinedness.

Definition 4.1 A sequence
$$s = (a_1, a_2, \ldots, a_n, \ldots)$$
where for every n, a_n is from $Dom(A_n)$ or equals UNDEF , is called the state of an object.

Every a_n is denoted also by $A_n(s)$ and is treated as
"the value of the entity A_n in the state s".
For $X = B_1, B_2, \ldots$, we will write

$$X(s) = (B_1(s), B_2(s), \ldots),$$

if some $B_j(s)$ is equal to UNDEF we shall take $X(s)$ to be undefined:

$$X(s) = \text{UNDEF} .$$

Definition 4.2 An arbitrary set of states is called (an instance of) a database.

Example 4.1 *(continues Example 3.2)* We may consider the following database T:

A	B	C	F	G
1	1	2	p_2	q_2
1	2	3	p_3	q_3
2	1	3	p_3	q_3

where p_2 and p_3 are programs transforming any input into 2 and 3, respectively,
q_2 and q_3 are programs transforming any input into p_2 and p_3, respectively.

4.2 Four possible versions for 'BE SOLVABLE'

We are interested in instances of databases that are consistent with all the laws from a given relational knowledge base:

Definition 4.3 For basic dependencies:

(a) We say that a 'functional dependency' $(X \to Y)$ is satisfied on an instance T if there exists a program g mapping $Dom(X)$ into $Dom(Y)$ such that, for every state s from T, if $X(s)$ is defined then $Y(s)$ is defined and $Y(s) = g(X(s))$.

(b) An 'operator dependency' $D : ((X_1 \to Y_1) \to Y)$ is satisfied on an instance T if, for every W, if the functional dependency $(W\&X_1 \to Y_1)$ is satisfied on T then the functional dependency $(W \to Y)$ is satisfied on T.

This item of the definition is based on the following

Principle of conservation: WHEN THE VALUES OF ENTITIES ARE KEPT FIXED, ALL THE LAWS MUST BE PRESERVED.

That is to say:

For a given W, [3] let g be a program computing Y_1 from the conjunction $W\&X_1$ on the whole database T.

When we fix values for the entities from W, say $W = w$, we thus truncate the database. On the new database the dependency between X_1 and Y_1 becomes functional, namely, one can extract a program p from g such that $Y_1 = p(X_1)$ on the new database. According to the operator dependency D, Y can be computed from p and, finally, Y is computed from w. [4]

(c) A variant dependency $(X \to (Y_1 \text{ } or \text{ } Y_2))$ is said to be satisfied on an instance T if there exists a program q mapping $Dom(X)$ into the set $\{1,2\}$, a program g_1 mapping $Dom(X)$ into $Dom(Y_1)$, and a program g_2 mapping $Dom(X)$ into $Dom(Y_2)$ such that, for every state s from T, if $X(s)$ is defined then

1. if $q(X(s)) = 1$ then $Y_1(s)$ is defined and $Y_1(s) = g_1(X(s))$,
2. if $q(X(s)) = 2$ then $Y_2(s)$ is defined and $Y_2(s) = g_2(X(s))$.

(d) We say that an instance T is **in full accordance** with a functional entity F of the type $(X_1 \to Y_1)$ if

(a) for every state s from T, if both $X_1(s)$ and $F(s)$ are defined then $Y_1(s)$ is defined and $Y_1(s) = F(s)(X_1(s))$. [5]

(b) the operator dependency $((X_1 \to Y_1) \to F)$ is satisfied on T.

Definition 4.4 An instance T is called consistent with a set of basic formulas Γ if

(1) all the dependencies from Γ are satisfied on T,

[3] For the weakest case, when W is empty, we have got a characterization related to modal logics S4 and S5.
[4] It should be noted that this item of the definition is also correlated with Kleene's s-m-n theorem.
[5] Let us recall that $F(s)$ is a program of the type $(X_1 \to Y_1)$.

(2) T is in full accordance with every functional entity F from Γ.

Definition 4.5 (Solvability) For a given class of instances K, we say that a task $\Gamma \vdash Z$ is K-solvable if, for every instance T from K, if T is consistent with Γ, then the functional dependency $(\emptyset \to Z)$ is satisfied on T.

Example 4.2 *(continues Example 4.1)* It seems that the database T from Example 4.1 **rejects** our **valid formula** from Example 3.1 because

(1) the dependency $(A\&B \to C)$ is satisfied on T,

(2) T is in accordance with both functional entities F and G, but

(3) the dependency $(\emptyset \to G)$ is not satisfied on T.

The point is that **there is no full accordance** between T and F.

If we truncated T by setting:

$$A = 1,$$

then, on the truncated database T_1:

A	B	C	F	G
1	1	2	p_2	q_2
1	2	3	p_3	q_3

the relation between B and C became functional and, hence, F were not in accordance with T_1.

Theorem 4.1 (Robustness and Completeness) . *For every V , let $Dom(V)$ be infinite or empty.* [6]
Let K be

(a) either the class of all databases, or

(b) the class of all finite databases.

Then, for a given task sequent $\vdash Z$ the following sentences are equivalent pairwise:

(i) $\Gamma \vdash Z$ is K-solvable,

(ii) on replacing all computable functions with the corresponding set-theoretical functions in the definitions related to the concept of consistent databases, the task $\Gamma \vdash Z$ is K-solvable in this new sense,

(iii) one can construct a program for the task $\Gamma \vdash Z$.

(iv) $\Gamma \vdash Z$ can be derived in the **Calculus of Tasks** *(see below).*

[6] This hypothesis is essential!

Theorem 4.1 shows that all reasonable definitions are equivalent and demonstrates that our definition of solvable tasks is very robust and does not depend on the particular choice of a level of constructivity. [7]

Corollary 4.1 *Under the hypotheses of Theorem 4.1, a sequent* $\Delta,\ (\emptyset \to X) \ \vdash\ Z$ *is derivable in the* **Calculus of Tasks** *if and only if, for every database* T *(from* K*) that is consistent with* Δ **(saying nothing about** $(\emptyset \to X)$ *), the functional dependency* $(X \to Z)$ *is satisfied on* T*.*

5 The Calculus of Tasks

All theorems are based on the **Calculus of Tasks** that operates with task sequents.

5.1 The language of the Calculus of Tasks

Let us recall that a **task sequent** is of the form

$$\Gamma \ \vdash\ Z \ ,$$

where Γ is a multiset consisting of formulas of one of the following three **basic forms:**

(a) $(X \to Y)$,

(b) $((X_1 \to Y_1) \to Y)$,

(c) $(X \to (Y_1 \ or\ Y_2))$.

5.2 Axioms for the Calculus of Tasks

Definition 5.1 For a sequent $\Gamma \ \vdash\ Z$, by $Out(\Gamma)$ we denote the set of all B such that a formula of the form $(\emptyset \to W_1\&B\&W_2)$ is contained in Γ.

Definition 5.2 A sequent $\Gamma \ \vdash\ Z$, is called **an axiom** if either

(a) Z is contained in $Out(\Gamma)$, or

(b) for some B such that $Dom(B)$ is empty, B is contained in $Out(\Gamma)$.

5.3 Inference Rules for the Calculus of Tasks

Let us give the inference rules for the **Calculus of Tasks** :

[7]Theorem 4.1 is valid for all finite knowledge bases. As for infinite knowledge bases cf. section 7

Composition: $$\frac{\Gamma,\ (X \to Y)\ \vdash\ Z \qquad B \in Out(\Gamma)}{\Gamma,\ (X\&B \to Y)\ \vdash\ Z}$$

Composition': $$\frac{\Gamma,\ (X \to (Y_1\ or\ Y_2))\ \vdash\ Z \qquad B \in Out(\Gamma)}{\Gamma,\ (X\&B \to (Y_1\ or\ Y_2))\ \vdash\ Z}$$

Subprogram: $$\frac{\Gamma,\ (\emptyset \to X_1)\ \vdash\ Y_1 \qquad \Gamma,\ (\emptyset \to Y)\ \vdash\ Z}{\Gamma,\ ((X_1 \to Y_1) \to Y)\ \vdash\ Z}$$

Branching: $$\frac{\Gamma,\ (\emptyset \to Y_1)\ \vdash\ Z \qquad \Gamma,\ (\emptyset \to Y_2)\ \vdash\ Z}{\Gamma,\ (\emptyset \to (Y_1\ or\ Y_2))\ \vdash\ Z}$$

5.4 The programmer's interpretation of the Calculus of Tasks

Each inference rule can be perceived as a **formalization of a constructive step in a natural reasoning** related with a process of solving tasks:

(Composition) If a knowledge base contains a dependency $(\emptyset \to B)$ (that means B is computed) and a dependency $(B \to Y)$ (that means Y is computed from B) then we can compute Y with the help of a composition and, therefore, the true law $(\emptyset \to Y)$ can be added to the knowledge base.

(Subprogram) If, for a functional entity F of the type $(X_1 \to Y_1)$, we want to add the dependency $(\emptyset \to F)$ ("a procedure F is implemented") to the set of laws Γ, then , in advance, we must synthesize a subprogram for this F, in other words solve the **'subtask'** $\Gamma,\ (\emptyset \to X_1)\ \vdash\ Y_1$.

(Branching) If, for a variant dependency $(\emptyset \to (Y_1\ or\ Y_2))$ that means either Y_1 or Y_2 is computed, we are able to solve both 'subtasks': $\Gamma,\ (\emptyset \to Y_1)\ \vdash\ Z$ and $\Gamma,\ (\emptyset \to Y_2)\ \vdash\ Z$, then we can solve the main task.

Corollary 5.1 *Taking into account what has been sa a program can be extracted* **directly** *from a derivation in the* **Calculus of Tasks** .

Corollary 5.2 *This minimal se rules of tasks reasoning turns out to cover* **all possible rules of tasks reasoning** *on th opositional level.*

5.5 The Calculus of Tasks is an information preserving calculus

We can used the liberty that is provided by the following **strange corollary.**

Corollary 5.3 *None of the rules of the* **Calculus of Tasks** *does loose any information. More specifically,*

For the composition rule: *its conclusion is derivable if and only if its premise is derivable.*

For the subprogram rule: *if its left premise is derivable then its conclusion is derivable if and only if its right premise is derivable.*

For the branching rule: *its conclusion is derivable if and only if both its premises are derivable.*

5.6 Kripke models vs Databases

Our refutable database can be easily transformed into a Kripke model K that refutes the corresponding propositional formula f.

But there is **no straightforward inverse transformation** because such an inverse **should ensure** the full accordance with all the functional entities, which is a very strong and tough condition. In particular, considering "self- and cross-referential" functional entities and variant dependencies, we need **the recursion theorem and unbounded domains.**

6 Complexity of the Decision Problem for Int

6.1 The Algorithm of Analysis and Synthesis

Let us consider the following algorithm based on the **Calculus of Tasks** .

Input. A task sequent $S: \ \Gamma \vdash Z$.

Output. A scheme program for S if S is solvable, or a refutable database , otherwise.

Method. A derivation of the sequent $\Gamma \vdash Z$ is being searched for.

If this search is successful then the derivation is transformed into the scheme program for S.

Otherwise, the answer is: "S is unsolvable" and the corresponding refutable database is constructed.

Theorem 6.1 *For each entity V, let $Dom(V)$ be either infinite or empty. Then this algorithm runs correctly on all finite task sequents.*

6.2 Space Complexity of the Decision Problem for Int

Theorem 6.2 *For a given sequent $S: \ \Gamma \vdash Z$, one can*

(a) recognize whether S is solvable or not,

(b) construct a minimal scheme program for S,

in deterministic **linear** *space*

$$O(L)$$

where L is the number of all occurrences of literals in Γ.

Proof. We can search for a derivation of the sequent $\Gamma \vdash Z$ with the help of a depth-first search.

Taking into account Corollary 5.3, for a given sequent $\Gamma_1 \vdash Z_1$ that is assigned to a vertex of this search tree, the search stack needs to contain no more than

(1) a number of names of 'variant' and 'operator' formulas (without repetitions),

(2) the set $Out(\Gamma_1)$ (without repetitions).

Thus we get a linear bound on the stack size. □

6.3 Subtasks and Measure of Unnaturalness

Taking into account the PSPACE-completeness, all known algorithms of analysis and synthesis are forced to perform an exponential search for "almost all" computational tasks. In spite of this, all examples of "bad" tasks are unnatural.

We introduce some level (rank) r to task sequents, so that natural (realistic) tasks have a small level r.

Now let us explain what subtasks are and how they interact.

According to what has been said, in performing the task $\Gamma \vdash Z$ there may appear 'subtasks', e.g. in such cases as follows:

(a) For a functional entity F of the type $(X_1 \to Y_1)$, we must solve a subtask $\Gamma,\ (\emptyset \to X_1) \vdash Y_1$, the input of it is the "argument list" X_1.

(b) If we use a variant dependency $(\emptyset \to (Y_1\ or\ Y_2))$ for computing some Z_1, we have to solve two subtasks
$\Gamma,\ (\emptyset \to Y_1) \vdash Z_1$ and $\Gamma,\ (\emptyset \to Y_2) \vdash Z_1$,
where the inputs of these subtasks are the "alternative lists" Y_1 and Y_2.

In performing the main task, subtasks can interact, namely, we can solve a subtask provided that values of inputs of some other subtasks are given in addition. Embedding of subprograms is related to this phenomenon.

Definition 6.1 (The degree of subtask interaction) For a given task sequent S, we say that **the degree of subtask interaction in** S is not greater than r if there is a solution for S such that the maximal number of subtasks with different inputs which can interact in the process of this solution does not exceed the integer r.

6.4 Non-uniform upper bound for Time Complexity

Theorem 6.3 *For a given sequent* $S:\ \Gamma \vdash Z$ *, one can*

(a) recognize whether S *is solvable or not,*

(b) *construct a program for S,*

in **quasipolynomial** *running time*

$$O(\frac{L \cdot n^2 \cdot m^{3r}}{(r!)^3})$$

where L is the number of all occurrences of literals in Γ,
n is the number of different literals from Γ,
m is the total number of different "argument lists" and "alternative lists" from Γ,
r is the minimal degree of subtasks interaction with which S can be solved.

In fact, our algorithm runs faster than in Theorem 6.3.

Let us consider small r.

Corollary 6.1 *For all sequents with $r = 0$ (that corresponds to Horn clauses), our algorithm runs in* **linear time.**

If the minimal degree of subtasks interaction with which a task can be solved is equal to 1, we say that this task is solvable **with separable subtasks.**

Corollary 6.2 *Our algorithm runs in* **quadratic time** *for all task sequents that can be solved with separable subtasks.*

Corollary 6.3 *We can solve task sequents with separable subtasks in parallel near-linear time.*

6.5 Subtask Interaction vs Embedding of Subtasks

Finally, let $Task_{Embedding=r}$ be a set of all tasks for which there exist programs such that embedding of their subprograms and conditional statements is not greater than r.

Theorem 6.4 *For every r, this class is a* **proper** *subclass of the class of all tasks that are solvable with degree r of subtasks interaction.*

On the other hand, Theorem 4.1 implies

Corollary 6.4 *For each entity V, let $Dom(V)$ be either infinite or empty. Then a task $\Gamma \vdash Z$ is solvable if and only if it is contained in the class $Task_{Embedding=m}$,*
where m is the total number of different "argument lists" and "alternative lists" from Γ.

Theorem 6.5 *For a given task sequent S : $\Gamma \vdash Z$, we can*

(a) *recognize whether S is solvable or not,*

(b) *construct a program for S,*

in **quasipolynomial** *running time*

$$O(L \cdot n \cdot m^r)$$

in **linear** *space*

$$O(L)$$

where r is the minimal integer that
S is contained in the class $Task_{Embedding=r}$.

Theorem 6.6 *For every r, the class $Task_{Embedding=r}$ is running in parallel near-linear time.*

Summary: Treating task sequents on the basis of our calculus, an exponential execution time should be expected in the worst case, as is customary.

But such cases arise for very unnatural tasks that need maximum cross-linking of all possible subprograms, even in the best programs.

Our prover runs in polynomial time on all natural tasks; the degree of the polynomial is determined by the minimal depth of interacting of subtasks that can be achieved in some solution for the main task.

7 Finite vs. Infinite Knowledge Bases

It should be pointed out that Theorem 4.1 is valid for all infinite knowledge bases containing no variant dependencies, as well as for many infinite knowledge bases having such dependencies.

Nevertheless, we can show an infinite knowledge base Γ (containing only a single variant dependency) and a $S : \ \Gamma \vdash Z$ such that

(1) this S is solvable, but still

(2) there is no program for this S.

8 Conclusion

In conclusion it should be pointed out that

- In fact, our calculus works as **a rewriting system**, by means of **simplifying tasks and reducing them to equivalent 'normal' forms.**
- On the basis of similar **information preserving** calculi one can get algorithms that run in polynomial (and even linear or subquadratic) time also for

(1) the membership problem in the theory of relational databases with functional and multivalued dependencies,

(2) recognizing the validity of Horn formulas in monadic predicate logic,
(3) flow analysis of "and-or" graphs,
(4) recognizing derivability of formulas of some kind in the classical and intuitionistic propositional and modal calculi, etc.

Acknowledgements

I am gratefully indebted to Prof. Johan van Benthem and Prof. Jan van Eijck for providing supportive and stimulating environments to present and discuss the work reported here.

I would like to thank my Dutch colleagues: Johan van Benthem, Jan van Eijck, Dick de Jongh, Michel van Lambalgen, Fer-Jan de Vries, A.S.Troelstra and many others from the University of Amsterdam and CWI for very useful and stimulating discussions. In particular, Jan van Eijck gave valuable comments on the manuscript.

References

[AHU 76] A.Aho, J.Hopcroft and J.Ullman, The Design and Analysis of Computer Algorithms, (1976).

[DK 85] A.Ja.Dikovskii and M.I.Kanovich, Computational models with separable subtasks. Proceedings of Academy of Sci. of USSR, Technical Cybernetics, 5 (1985), 36-60. (Russian)

[GJ 79] M.R.Garey and D.S.Johnson, Computers and Intractability, (1979).

[Kanovich 87] M.I.Kanovich, Quasipolynomial algorithms for recognizing the satisfiability and derivability of propositional formulas. Soviet Mathematics Doklady, 34, N 2 (1987), 273-277.

[Kanovich 90] M.I.Kanovich, Efficient program synthesis in computational models. J. Logic Programming, 9, N 2-3 (1990), 159-177.

[Kanovich 91] M.I.Kanovich, Efficient program synthesis: Semantics, Logic, Complexity. Theoretical Aspects of Computer Software, TACS'91, Japan, Sendai, 1991, September.

[Ladner 77] R.Ladner, The computational complexity of provability in systems of modal propositional logic. SIAM J. Computing, 6 (1977), 467-480.

[Mints 83] G.E.Mints and E.Kh.Tyugu, Third All-Union Conference 'Application of the Methods of Mathematical Logic', Proceedings, Tallinn, (1983), 52-60. (Russian)

[Mints 90] G.E.Mints and E.Kh.Tyugu, Propositional logic programming and the PRIZ system. J. Logic Programming, 9, N 2-3 (1990), 179-193.

[Rogers 67] H.Rogers, Theory of Recursive Functions and Effective Computability, (1967).

[Statman 79] R.Statman, Intuitionistic propositional logic is Polynomial-Space complete. Theoret. Computer Sci., 9 (1979), 67-72.

[Ullman 80] J.D.Ullman, Principles of Database Systems, (1980).