GRIGORI MINTS AND COMPUTER SCIENCE

Enn Tyugu
Institute of Cybernetics, Tallinn University of Technology

Abstract. A survey is presented of works of Grigori Mints from the eighties of the last century where logic was applied to program synthesis and semantics of specification languages. It demonstrates examples of fruitful application of logic in computing. He has left a visible footprint in the Estonian computer science.

1. Coming to Tallinn

We in Tallinn established contacts with Grigori Mints in 1980. He started participating in the summer- and winter-schools organized with the best Soviet computer scientists and mathematicians as lecturers. Years ago before that, Grigori had participated in a successful large project of construction of a natural deduction theorem-prover together with N. Shanin, S. Maslov et al. [9] which required considerable amount of programming, but his interests and experiences in computer science and computer applications were low. It was a pleasant surprise that he became interested in the program synthesizer that we were developing with a strong practical orientation. The synthesizer called PRIZ [9] had been developed as a result of a long experimentation and worked on mainframes similar to IBM370. We were trying to explain the synthesis algorithm in terms of logic, but found little understanding from logicians. Grigori spent many hours questioning us about the synthesis algorithm and trying to fit it into logic.

Grigori lost his researcher position in St. Petersburg, because he was a quiet dissident, not actively outspoken, but enough to be persona non grata for the official Soviet science. For a shorter period he worked even in a software house in St. Petersburg as a programmer, writing code in the IBM assembler language. Hillar Aben -- Director of the Institute of Cybernetics of Estonian Academy of Sciences listened to my pleading and agreed to give Grigori a researcher position at the Institute of Cybernetics in Tallinn in the end of 1980. This was a really good luck for the computer science and logic in Estonia. Grigori spent ten years in Tallinn -- until 1991, when he took a position of professor at the Stanford University. He started educating us in logic, published an easily readable preprint on logical foundation of program synthesis [9], and gave lectures. He was teaching logic, doing research, organizing scientific meetings (including a large meeting on relations between computing and logic COLOG-88 [9]) and communicating with scientists of different countries as much as the circumstances permitted it. Institute of Cybernetics hosted S. McLane, H. Barendregt, S. Maslov, P. Martin-Löf, J. McCarthy, P. Suppes, J.-Y. Girard, V. Lifshits, A. Slisenko, E. Griffor and many other mathematicians and logicians as visitors invited by Grigori Mints.

2. Structural synthesis in the beginning

In the beginning of the eighties we started calling the program synthesis that was used in PRIZ "structural program synthesis" [9], because it relied on structural properties of computations, taking into the account only which variables were inputs and outputs of building blocks of a synthesized program -- like it is generally accepted now in service composition. Actual relations between the values of inputs and outputs were ignored. However, functional inputs were permitted, and the values of these inputs had to be synthesized as well.

A goal of the synthesis (a synthesis problem) was presented by defining only input and output variables of the required program. Complete specification of a synthesis problem was presented as a set of all objects that are candidates for being computed (i.e. that may be involved in solving the problem) and a set of functions that can be used for computing these objects. The elements of the first set could be called variables, because they could get values computed by

the elements of the second set, but these were variables with single assignment, hence not conventional program variables. Besides these two sets, a set of functional variables that are inputs of the functions was given. Such a variable had to be evaluated during the program synthesis process, if a function having it as an input was used in computations.

A specification for structural synthesis of a program could be easily represented as a graph with nodes as elements of the sets from above, and incidence relation of nodes representing data dependencies (i.e. inputs and outputs) of functions. In order to explain the synthesis algorithm, a symbolic notation was introduced for the specification as well. Each function was represented by a formula $x_1,..x_m \to y_1,...y_n\{f\}$ called a computability statement, where $x_1,..x_m$ were the inputs and $y_1,...y_n$ the outputs of the function f. There was a problem that some of the inputs were functional variables that had to be evaluated during the program synthesis, and not during the execution of the program. Inputs and outputs of a functional variable were given in a specification as well. Therefore such a variable itself could be described by a formula that gives its inputs and outputs: $u_1,... u_k \to v_1,... v_l$. The order of inputs and outputs of a function could be ignored at the derivation of an algorithm, hence a formula for a function took the form $a \to b$ with a and b sets of its inputs and outputs. Different versions of the synthesis algorithm were described by respective derivation rules for stepwise construction of an algorithm. As an example here is a rule for a case when no functional variables were present

$$\frac{A \to B \cup C \quad C \cup G \to F}{A \cup G \to F},$$

where A, B, C, G, F are metavariables denoting sets of variables. Handling functional inputs of the building blocks required higher order notations, and this was not exactly defined for the structural synthesis of programs. This was how we saw the structural synthesis of programs (*SSP*) before Grigori became involved.

3. Completeness of structural synthesis

We were quite sure that the synthesis algorithm of PRIZ was complete in some sense (which we had not defined yet). But, I remember very well one morning, when Grigori just arrived by train from St. Petersburg, where he still was working part time as a programmer in a company called "Lengipromjasomolprom", and gave an example of a specification and a goal, asking whether the goal is solvable by the PRIZ system. It became soon clear that this goal was unsolvable by the synthesis algorithm implemented in PRIZ. This event started a discussion of exact representation of the *SSP* in logic and, naturally, attempts to improve the synthesis algorithm. As a result, the synthesis algorithm was changed, and the first paper on logic of *SSP* appeared [9].

The following is a brief summary of the results on completeness of *SSP*. Let us assume for simplicity that the output of a function is always only one (maybe structured) value. A precise logical description of a building block f, represented earlier by a computability statement

$x_1,x_2,...x_k \to y\{f\}$

is as follows:

$$\forall u_1 \forall u_2 ... \forall u_k (X_1(u_1) \wedge X_2(u_2) \wedge ... X_k(u_k) \to Y(f(u_1,u_2,...u_k))), \qquad (*)$$

where $X_1, X_2,...X_k$ are unary predicate symbols denoting computability of a proper value of x_i, so that the formula $X_i(s_i)$ can be read as "the value s_i is a suitable value for the input (or output) x_i".

In the case of m functional variables $g_1,g_2...g_m$ as additional inputs of a building block F, the logical formula describing the building block takes the form

$$\bigwedge_{1\le i\le m}(\forall s_{i,1}\forall s_{i,2}...\forall s_{i,ki}(U_{i,1}(s_{i,1})\wedge U_{i,2}(s_{i,2})\wedge...U_{i,ki}(s_{i,ki})\to V(g_i(s_{i,1},s_{i,2},...s_{i,ki})))\ )\to$$

$$\to\forall u_1\forall u_2...\forall u_k(X_1(u_1)\wedge X_2(u_2)\wedge...X_k(u_k)\to Y(F(u_1,u_2,...u_k,g_1,g_2,...g_m))).\quad (**)$$

This is the general form of formulas appearing in the logical language of structural synthesis. The nested implications on the left side and functional variables may be missing in many specifications of building blocks, see (*). To present the logical rules of structural synthesis of programs we will use the abbreviation (***) instead of the form (**)

$$\bigwedge_{1\le i\le m}(\boldsymbol{U}_i\to V_i\{\phi_i\})\to(\boldsymbol{X}\to Y\{F\}),\qquad (***)$$

where the structure of a formula is preserved, but all bound variables as well as inessential indices are omitted, $\boldsymbol{U}_i$ and $\boldsymbol{X}$ denote conjunctions of unary predicates. Let us note that this formula includes functional variables $\phi_1,\ \phi_2...\phi_m$ and a higher-order functional constant F. From now on it will be assumed that the formulas are closed – universally quantified with respect to the variables $\phi_1,\ \phi_2...\phi_m$ as well.

The structural synthesis rules *SSR* in a sequent notation were defined as follows:

$$\frac{\Rightarrow\bigwedge_{1\le i\le m}(\boldsymbol{U}_i\to V_i\{\phi_i\})\to(\boldsymbol{X}\to Y\{F\});\ \ \Gamma_i\Rightarrow\boldsymbol{U}_i\to V_i\{g_i\},\ i=1,2,...m}{\Gamma_1,\ \Gamma_2,...\ \Gamma_m\Rightarrow\boldsymbol{X}\to Y\{F^*\}}\quad (--)$$

$$\frac{\Gamma\Rightarrow\boldsymbol{X}\to Y\{F\};\ \Sigma_i\Rightarrow X_i(t_i),\ \ i=1,2,...n}{\Gamma,\Sigma_1,\Sigma_2,...\Sigma_n\Rightarrow Y(F(t_1,t_2,...t_n))}\quad (-)$$

$$\frac{\Gamma,\boldsymbol{X}\Rightarrow Y(t)}{\Gamma\Rightarrow\boldsymbol{X}\to Y\{\lambda\boldsymbol{x}.t\}}\quad (+)$$

Axioms $\Gamma,H\Rightarrow H$ for any formula H are also assumed. $\boldsymbol{X}$ on the left side of the sequent in the rule (+) denotes a list of unary predicates. Σ,Γ denote lists of formulas. F^* denotes the result of substitution of g_i for ϕ_i, $i=1,2,...m$ in F. We distinguish between terms for unary predicates and functional terms for implications by using different brackets: () and {} respectively.

Grigori proved the following two theorems in [9] that demonstrate the completeness of SSR.

Theorem 1.
Let $C_1,...,C_k$ be formulas of the form (***) and let C be a goal $\boldsymbol{U}\to V$.
Then a sequent $C'_1,...,C'_k\Rightarrow C'$ is derivable in the calculus of natural deduction if and only if $\Rightarrow C$ is derivable from $\Rightarrow C_1,...,\ \Rightarrow C_k$ by the rules SSR, where $C'_1,...,C'_k$, $C'$are respective formulas of the form (**).

Theorem 2.

Let $C_1,\ldots,C_k$ be formulas of the form (***), $\boldsymbol{W}$ a list of unary predicates denoting computability, and let $\boldsymbol{K}$ be a conjunction of the goals $\boldsymbol{U}_i \to V_i$, $1 \le i \le m$.
Then the sequent

$C'_1,\ldots,C'_k, \boldsymbol{W} \Rightarrow \boldsymbol{K}$

is derivable by natural deduction if and only if all the sequents

$\boldsymbol{W} \Rightarrow \boldsymbol{U}_i \to V_i$, $1 \le i \le m$ *have normal deductions from* $\Rightarrow C_1,\ldots,\ \Rightarrow C_k$ *according to the rules of SSR. Here* $C'_1,\ldots,C'_k$ *denote respective formulas of the form (**).*

Proof of the first theorem is rather obvious – *SSR* are admissible rules of intuitionistic logic for the abbreviated form (***) of formulas. Proof of the second theorem is based on the observation that any natural deduction can be transformed into deduction in the long normal form, and deductions in the long normal form can be easily transformed in deductions with *SSR* [9].

4. It can be a propositional logic

The formula (**) includes only unary predicates, and the quantifiers are used so that we can move quantifiers close to the predicates with respective bound variables and transform the quantified subformulas into the form $\exists u U(u)$. Indeed, instead of a conventional specification $\forall x(P(x) \to \exists y R(x,y))$ of a program (or its building block) where input x and output y are bound by a relation R, the *SSP* uses a specification $\forall x(P(x) \to \exists y R(y))$. It can be presented in the equivalent form $\exists x P(x) \to \exists y R(y)$. Now the closed subformulas $\exists x P(x)$ and $\exists y R(y)$ can be considered as propositions, i.e. the specification becomes $P \to R$, where P denotes computability of the input and R denotes computability of the output of a program. Bearing in mind that the proofs in SSP are constructed in intuitionistic logic, one should not worry about the description of computations – the realizations of formulas would be programs, more precisely – lambda terms that can be easily converted to programs. This is how the authors reasoned when writing the paper [9].

The logical language of *SSP* becomes now an implicative fragment of the propositional language with restricted nestedness of implications. The general form of the formulas of a specification is (***), assuming that a special case without nested implications is also accepted. But now the formulas obtain precise meaning instead of being just abbreviations of quantified formulas. The symbols g_i and F denote now lambda terms that are realizations of implications $\boldsymbol{U}_i \to V_i$ and $(\boldsymbol{X} \to Y)$ respectively. They are constants for preprogrammed building blocks of programs, and more complex lambda-terms built for derived formulas. The inference rules for this language remain in essence the same as shown above. Changing slightly the rule (--) gives us derivations where each step exactly corresponds to the application of a preprogrammed function – a computation step. The changed rule requires that all inputs of a function are computed, and the rule is as follows:

$$\Rightarrow \bigwedge_{1 \le i \le m} (\boldsymbol{U}_i \to V_i\{\phi_i\}) \to (\boldsymbol{X} \to Y\{F\});\ \ \Gamma_i,\ \boldsymbol{U}_i \Rightarrow V_i(g_i),\ i=1,2,\ldots m;\ \Sigma_j \Rightarrow X_j(t_j),\ j=1,\ldots n$$

$$\text{---} \quad (--)$$

$$\Gamma_1,\ \Gamma_2,\ldots\ \Gamma_m,\ \Sigma_1,\Sigma_2,\ldots\Sigma_n \Rightarrow Y(F(g_1,\ldots,g_m,t_1,\ldots,t_n))$$

Now the rule (-) becomes a special case of the rule (--), hence it is not needed any more.

Grigori pointed out that the language of *SSP* is expressive and *SSR* rules are complete in the following sense. Given a list L of intuitionistic propositional formulas and a proposition B, the list L can be transformed into a list L' of formulas of the form (***) such that $L' \Rightarrow B$ is derivable in *SSR* if and only if $L \Rightarrow B$ is derivable in intiutionistic logic [9]. This transformation is performed by introducing new propositional variables for every subformula $A*B$ with a connective * that spoils the form (***) and applying equivalent substitution theorem – substituting a new variable W instead of a subformula $A*B$, and adding the implications $W \rightarrow A*B$ and $A*B \rightarrow W$ to the list L so that the derivability is preserved. Elimination of $\perp$ as well as elimination of $\vee$ on the right side of an implication requires more effort and expands the list L' polynomially, because new implications have to be introduced for every existing propositional variable, see [9].) These transformations are suggested by the following two second order equivalences:

$(p \vee q) \leftrightarrow \forall x((p \rightarrow x) \rightarrow ((q \rightarrow x) \rightarrow x))$ and $\perp \leftrightarrow \forall x x$.

We made a theorem-proving experiment – proved all intuitionistic propositional theorems (more than one hundred theorems) from S. Kleene's "Introduction to metamathematics" [9], first, encoding the theorems in the input language of the PRIZ system and then using the *SSP* program synthesizer as a theorem prover [9]. This happened to be an interesting experiment, because the prover that had been initially designed for program synthesis gave interesting proofs. An example is a derivation of the intuitionistic analog $((((A \rightarrow B) \rightarrow A) \rightarrow A) \rightarrow B) \rightarrow B$ of the valid classical formula $((A \rightarrow B) \rightarrow A) \rightarrow A$. Because of the deeper nestedness of implications, this formula had to be rewritten by introducing new propositional variables X, Y denoting respectively $A \rightarrow B$ and $X \rightarrow A$. This gave a new formula $((Y \rightarrow A) \rightarrow B) \rightarrow B$. Taking B as a goal, one had to derive the sequent $\Rightarrow B$ from the sequent

$\Rightarrow (Y \rightarrow A) \rightarrow B$,

using possibly also the axioms that were added according to the equivalent replacement condition:

$\Rightarrow X \rightarrow (A \rightarrow B)$
$\Rightarrow (A \rightarrow B) \rightarrow X$
$\Rightarrow Y \rightarrow (X \rightarrow A)$
$\Rightarrow (X \rightarrow A) \rightarrow Y$.

The proof was as follows:

$$\dfrac{\dfrac{\dfrac{\Rightarrow (Y \rightarrow A) \rightarrow B \qquad Y \Rightarrow Y \qquad \Rightarrow X}{\Rightarrow Y \rightarrow (X \rightarrow A) \qquad Y \Rightarrow A}(--)}{\Rightarrow (Y \rightarrow A) \rightarrow B \qquad A \Rightarrow B}(--)}{\Rightarrow (A \rightarrow B) \rightarrow X \qquad Y, A \Rightarrow A}(--)$$

$$\Rightarrow B$$

5. Specifications as types

In practical applications, specifications for program synthesis are written in a language different from the logical language used for synthesis. Looking from a practical side – program synthesis has become today a part of a compilation technique of declarative problem-oriented languages, and most of the logic is hidden in a compiler. Although this approach was not common in the eighties of the last century, the *SSP* was used already in combination with a user-friendly declarative language. A question of the precise semantics of this language arouse. In the paper [9], an attempt was made by G. Mints, J. Smith and E. Tyugu to define the semantics of a small specification language in three different ways: by translating it into logic and applying SSP (*logical semantics*); by considering a specification as a presentation of a type, and proving that the type of a goal is inhabited (*types semantics*); by interpreting the specification in a set theory (*sets semantics*). The equivalence of these three semantics was shown. Considering the idea of formulas as types, the relatedness of these semantics should not be a surprise.

Let us look at the semantics of the kernel language of specifications for *SSP* presented in [9] and [9]. A specification written in this language is a sequence of statements of the following form:

a:(x:s;...;y:t),

where *a, x,... y* are new names, *a* will denote an object and *x,...,y* will denote its components that are also objects having types given by the type specifiers *s,...,t*. Names of the components *x,...,y* used outside of the object *a* are *a.x,...,a.y*, i.e. the prefix *a* is added to the names. Longer compound names like *a.x.u* may also appear, depending on the types specified by *s,...,t*. A type specifier can be

-- name of a primitive type
-- name of a object specified earlier
-- an expression of one of the following forms

$u_1,...,u_m \rightarrow u_{m+1}$

$u_1,...,u_m \rightarrow u_{m+1}\{f\}$,

where $u_1,...,u_m$, u_{m+1} are names of components, f is a name of a predefined function.

Logical semantics of the language was defined by giving a method of constructing a program for a given specification and a solvable goal. This method consisted, first, of rules for translating a specification in the language of *SSP,* and second, applying the *SSP*. The translation was rather straightforward: unfolding the specifications and copying the implications expressing computability for all introduced objects and their components. Besides that, one had to introduce extra implications to represent a structure of objects. For each compound object *a* with components *x,...,y* the following new implications had to be introduced:

$a \rightarrow x, ..., a \rightarrow y$ and $x,...,y \rightarrow a$.

These implications described the computability of components of *a* and the computability of *a* itself from its components.

It is easy to see that the kernel language describes objects of simple types, and it is easy today to apply type inference to find a proper term for a given goal. However, a more general approach with dependent types was used for types semantics in the work [9] done together with Jan Smith from the Gothenburg computer science group, where program synthesis in Martin-Löf's type theory was investigated. The types $(\Pi\ x{:}A)B$ and $(\sum\ x{:}A)B$ were used for representing $A \rightarrow B$ and $A \wedge B$ respectively. For a specification *S* with representation $\Theta(S)$ in the type language, and a solvable goal $a \rightarrow b$, the types semantics gave a term *f* such that $\Theta(S)$ |-- *f:*

*(*Π *x:A)B,* where |-- denotes the derivability in Martin Löf's type theory as described in [9]. Here again, first the representation *Θ(S)* of the specification was found, and then the term *f* was built.

6. Induction in a propositional language

The program synthesis used in the PRIZ system permitted to synthesize recursive programs from axioms given in the language of SSP. In order to formulate an extension of derivation rules for synthesis of recursive programs, the notion of a sequent was extended so that inductive proofs became possible. Expressions of the form *[A]* for formulas *A* were allowed to occur on the left side of a sequent for expressing the induction. The following rule for recursion was added:

$$\frac{\Gamma,\ [\boldsymbol{X} \to Y],\ \boldsymbol{X} \Rightarrow Y}{\Gamma,\ \boldsymbol{X} \Rightarrow Y}\ (Rec)$$

This rule was obtained from the usual transfinite induction rule by suppressing individual variables like we did in Section 3 for all bound vaiables. The rule (--) had to be extended as well:

$$\frac{\Rightarrow \bigwedge_{1 \le i \le m} (\boldsymbol{U}_i \to V_i) \to (\boldsymbol{X} \to Y);\ \ \Gamma_i \Rightarrow \boldsymbol{U}_i \to V_i,\ i=1,2,\ldots l}{\Gamma_1,\ \Gamma_2,\ldots\ \Gamma_l, [\boldsymbol{U}_i \to V_i]_{l+1 \le i \le m} \Rightarrow \boldsymbol{X} \to Y}\ (--R)$$

Also the programming language had to be extended with a recursion functional. This approach was quite innovative in the eighties. Introducing induction without explicitly requiring well-foundedness meant that one could not ensure termination of a synthesized program. The proof of termination had to be done by other means (remained a responsibility of the user). A simple example of a recursive synthesis is the following synthesis of factorial:

$$\cfrac{\cfrac{\cfrac{\Rightarrow (N \to F) \to (N \to F)}{[N \to F] \Rightarrow N \to F}\ (--R)}{N \Rightarrow F}\ (Rec)}{\Rightarrow N \to F}\ (+)$$

7. Propositional logic programming and more synthesis

Although logic programming is defined broadly as "using logic for program construction, it is often thought of as predicate Horn clause programming, e.g. Prolog. But, more generally, logic programming means exploiting the basic truth that the structure of a program is similar to a constructive proof of the fact that the desired result of the program exists (can be computed). This enables one to use results obtained in logic for building correct program schemas or for determining schemas of computations and control the computations. We made an attempt to present program development in PRIZ, i.e. the structural synthesis of programs, as propositional logic programming, and to extend the scope of logic programming in this way [9,9]. The control of computations in Prolog is performed in runtime by unification, it may cause backtracking and unnecessary computations. Control in PRIZ is encoded in the preprogrammed higher order

functions that call synthesized parts of programs. The search with backtracking is done before the actual computations, and unnecessary computations are avoided. Comparing the control in Prolog and PRIZ in a naïve way, we can say that a clause

P(X,Y) :- Q(...),R(...)

means in Prolog "if *P(X,Y)* may be used, then try to use sequentially *Q(...),...,R(...),* if this is not successful, then try to find some other way." In PRIZ, the control of computations is preprogrammed in the implementations of formulas with nested implications. A formula

$(u \to v) \wedge \ldots (s \to t) \to (x \to y)\{F\}$

means in PRIZ "if *F* can be used, then try to find functions for solving $u \to v, \ldots, s \to t$ and apply them as prescribed by *F*."

Grigori collected a number of papers from the Soviet logicians and computer scientists, and we put them in a special issue of the Journal of Logic Programing [9]. The papers by I. Babaev [9], by M. Kanovich [9], and by G. Mints and E. Tyugu [9] in this journal are on propositional logic programming. An interesting paper of V. Mikhailov and N. Zamov [9] is about the synthesis of technological algorithms that is also a program synthesis in some sense.

Grigori explained another synthesis algorithm used in PRIZ for so called independent subtasks in modal logic [9]. He also wrote a comprehensive and elegant paper about the complexity of proof search in different fragments of the intuitionistic propositonal logic [9], and I think he got the idea that this kind of a paper may be useful for computer scientists, when discussing several modifications of the PRIZ synthesizer.

Independent subtasks are useful because of less complex search needed for synthesis. A subtask is called independent, if its solvability does not depend on the availability of inputs of other subtasks. In the case when all subtasks are independent, the order of testing the solvability of subtasks is inessential as long as no new variables are computed. Hence, for finding in the synthesis process a new applicable function, one has to make at most *n* solvability tests of subtasks where *n* is the number of subtasks. A subtask solvability test can be performed in linear time with respect of the number of occurrences of propositional variables in all formulas. As the result, the time complexity of search in this case is polynomial. (It is exponential in the general case of SSP as it is for the proof search in the intuitionistic propositional calculus.)

For the independent subtasks, one used another rule *(--)ind* instead of the SSR rule *(--)*:

$$\Rightarrow \bigwedge_{1 \le i \le m} (\boldsymbol{U}_i \to V_i\{\phi_i\}) \to (\boldsymbol{X} \to Y\{F\});\ \ \boldsymbol{U'}_i \Rightarrow V_i(g_i),\ i=1,2,\ldots m;\ \Sigma_j \Rightarrow X_j(t_j),\ j=1,\ldots n$$

$$\frac{}{\Rightarrow Y(F(g_1,\ldots,g_m,t_1,\ldots,t_n))} \quad (--)ind$$

where $\boldsymbol{U'}_i$ denoted a list of variables occurring in the conjunction $\boldsymbol{U}_i$. One can see that the second premise of the rule was simpler – it did not contain extra formulas Γ_i that made the complexity of the search exponential. The rule *(--)ind* together with the SSR rules *(-)* and *(+)* were called *SSRind*.

Unfortunately for the PRIZ developers, using the rule *(--)ind* did not fit in the logic as we understood it. To help us, Grigori translated the formulas and sequents of the synthesis with independent subtasks in the language of modal logic by rewriting implications $A \to B$ as strict

implications $(A \supset B)$. He proved that a sequent in this language is derivable in the modal logic S4 if and only if the original sequent in the language of *SSP* is derivable in *SSRind* [9].

8. Concluding remarks

Working together with computer scientists in Tallinn, Grigori became an advocate of using logic in computer science. It was his idea to organize together with P. Martin-Löf and P. Lorents an international conference where logicians and computer scientists could meet. The International Conference in Computer Logic COLOG-88 [9] was a big success and it indeed brought together researchers from computing and logic from many countries.

Speaking about Grigori's influence on computer science one must not ignore his influence on young scientists whose educator he has been. A number of well-known professors found their way to logic and computer science due to Grigori, for example, Mati Pentus (now professor of Moscow University), Tarmo Uustalu and Tanel Tammet (professors of Tallinn Technical University). I am not speaking about Grigori's role in logic, but still wish to point out Sergei Tupailo as one example. Sergei was Grigori's doctoral student, he defended one Ph.D. thesis at Stanford University and another at Tartu University. He is dedicated to foundations of mathematics, and is known by his works on NF.

I take the complete responsibility for all mistakes and for the form of presentation of the results here, although most of ideas presented in this paper belong to Grigori Mints, who has had invaluable influence on the computer scientists who were lucky to work with him.

9. References

1. I. Babajev. Problem Specification and Program Synthesis in the System SPORA. J. of Logic Programming, v. 9, No. 2&3 (1990) 141 – 157.

2. Journal of Logic Programming, v. 9, No. 2&3 (1990)

3. M. Kanovich. Efficient Program Synthesis in Computational Models. J. Logic Programming, v. 9, No. 2&3 (1990) 159 – 177.

4. S. Kleene. Introduction to Metamatemathics. North-Holland (1952)

5. P. Martin-Löf, G. Mints: COLOG-88, International Conference on Computer Logic, Proceedings LNCS, Springer (1988)

6. V. Mikhailov, N. Zamov. Deductive Synthesis of Solutions for Technological Tasks. J. Logic Programming, v. 9, No. 2&3 (1990) 195 – 220.

7. G. Mints, E. Tyugu. The completeness of structural synthesis rules. Soviet Math. Doklady. V. 25 (1982) p. 2334 – 2336.

8. G. Mints. Logical foundation of program synthesis. Institute of Cybernetics (in Russian). Tallinn (1982).

9. G. Mints. Complexity of Subclasses of the Intuitionistic Propositional Calculus. Bit, v. 32, No. 1 (1992) 64 – 69.

10. G. Mints, E. Tyugu: Justifications of the Structural Synthesis of Programs. Sci. Comput. Program. 2(3): 215-240 (1982)

11. G. Mints, E. Tyugu. Semantics of a declarative language. OInformation Processing leters 23, Elsevier (1986) p. 147 – 151.

12. G. Mints, E. Tyugu: The Programming System PRIZ. J. Symb. Comput. 5(3): 359-375 (1988)

13. G. Mints, E. Tyugu: Propositional Logic Programming and the Priz System. J. Log. Program. 9 (2&3): 179-193 (1990)

14. G. Mints, E. Tyugu: The Programming System PRIZ. Baltic Computer Science LNCS, v. 502, Springer (1991) 1-17

15. G. Mints, J. M. Smith, E. Tyugu: Type-theoretical Semantics of Some Declarative Languages. Baltic Computer Science LNCS, v. 502, Springer (1991) p. 18-32

16. G. Mints, T. Tammet: Condensed Detachment is Complete for Relevance Logic: A Computer-Aided Proof. J. Autom. Reasoning 7(4): 587-596 (1991)

17. G. Mints. Propositional Logic Programming. In. J. Hayes, D. Michie, E. Tyugu (eds.) Machine Intelligence v. 12 (1991) 17 - 37

18. B. Nordström, K. Petersson, J. M. Smith. Programming in Martin-Löf's Type Theory. Chalmers (1989)

19. N. Shanin, G. Davidov, S. Maslov, G. Minc, V. Orevkov. A. Slisenko. An algorithm for a machine search of a natural logical deduction in a propositional calculus. (in Russian) Academy of Sciencies of the USSR, Steklov Mat. Inst., Leningrad department, "Nauka", Moscow (1965).

20. E. Tyugu: The structural synthesis of programs. Algorithms in Modern Mathematics and Computer Science LNCS, v. 122, Springer (1979) 290-303

21. E. Tyugu: Three New-Generation Software Environments. Commun. ACM 34(6): 46-59 (1991)

22. B. Volozh, M. Matskin, G. Mints, E. Tyugu. Theorem proving with the aid of a program synthesizer. Cybernetics, No. 6, (1982) 63 – 70.