View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Theoretical Computer Science 90 (1991) 253-270
Elsevier

A bridge between constructive logic and computer programming

N.N. Nepejvoda

Applied Logics and Systemotechnic Department, Kirova Str. 132, Izhevsk, SU-426001, USSR

Abstract

Nepejvoda, N.N., A bridge between constructive logic and computer programming, Theoretical Computer Science 90 (1991) 253-270.

Some logic notions have their analogies among programming concepts and vice versa. But people often try to understand these analogies in too straightforward a manner. A collection of analogies arising between constructive logics and programming is summarized and illustrated here. Some examples of complexities usually not taken into account are shown.

This paper is deliberately written informally. There are many works on applications of constructive logics, but there is a lack of understanding of new views and possibilities opened by these applications. It has been shown by more than 10 years experience that *informal* understanding of these peculiarities is maybe more valuable for many people than a rigorous *formal technique* (which is only necessary for researchers in this domain). Here we do not try to be rigorous; precise constructions have been presented in many papers (at least 70 from more than 10 authors) and will be summarized and explained in our forthcoming book on applied constructive logics.

1. Role of A.P. Ershov in bridging together theory and programming

Many people have said that there is a wall between practice and theory (even inside the single mind of very experienced persons) (see, e.g., [8]). To destroy this obstacle demands much effort. But for most programmers computer scientists there is no such problem. Theory and practice peacefully coexistent like the two banks of a deep river. Usually there is a mutual non-interference in the internal affairs. Sometimes a challenge from one side is accepted by the other and a bridge arises. Good examples are complexity theory, grammars and the theory of determinate games.

This state of affairs is sufficiently more difficult in the area of logic and programming. Although from the very beginning many basic ideas have been exported from

© 1991—IFIP

logic (especially from computability theory) to programming and recently their has been feedback (see e.g. dynamic logic and logic programming theory), many possible interconnections have been frozen for more than forty years. One reason is that *both* sides are here to adapt one for another, although many prejudicies of logic and programming interfere here.

The role of A.P. Ershov in the building of new bridges was the first. He is one of the founders of partial and mixed computations, which has resulted in many surprising connections between recursive functions and programs. Some results provocated and supported by him are outlined here.

Our research in this area was influenced by two men. First, the speech of A.A. Markov mentioned below gave a firm philosophical basis for the investigations. Second, A.P. Ershov was the man who understood how a "high theory" can be applied and constantly encouraged us to develop our work. A.P. Ershov stressed that programming needs its own theory, but most of Computer Science is a theory of *completed* programs. A.P. Ershov steered our investigations in the at first less promising direction. He understood that logic program synthesis was not realizable at that time, but he stressed that in this way we could see new, often surprising and striking, correlations and contradictions in programming and logic concepts, and to teach some powerful practitioners to use new programming methods. Only this direction had proved its viability at the present time.

In 1975–1978 I was Associate Professor of Udmurt University and there was a small group of promising young students interested mainly in applications. I was not introduced to A.P. Ershov but from Professor Ceytin he had heard that somewhere at Izhevsk there was a young logician interested in some advanced problems of programming, e.g. in more co-ordinated design of programming languages. He invited me to join the Soviet Algol-68 Commission and there were some discussions on the possibilities and on the need to apply logic to programming. But the "internal wall" mentioned above hindered me from working in this direction. Then A.P. Ershov published (in 1977) in the main Soviet Computer Science journal "Programming", a paper where he mentioned that there was a strong group at Izhevsk working on programming theory and practice. The first thesis was at that time wrong, the second one at least strongly exaggerating. It remained to us to save Ershov's reputation and to do something new immediately! So, in 1978 it came to light the first five works on constructive logic applications.

The second important factor in the development of our research was that Ershov in contrast to almost all logicians and mathematicians, supported my thesis that we can exploit one usually forgotten peculiarity of mathematical proofs; moreover, this is the informal definition of proof for our purposes.

Proofs are objects such that their syntactic correctness implies their semantic correctness.

Usually many partial properties of known proof classes are stressed, e.g. proof rules, three structure and so on. For *each* such property a counterexample is known today; namely, strange but useful structure classes are considered which can naturally

be regarded as proofs but do not have any likeness to the usual rule tree structure. For example, it has been shown so far back as the 19th century by J. Venn that some figures can be regarded as proofs of set theoretical judgements (so called Venn diagrams [18]) and there are nothing like rules in these diagrams.

The independent and deep insight of A.P. Ershov can be illustrated by the following example from 1982. During one discussion I had remarked that the loop

$$\{\mathfrak{A}\}\ \textbf{while}\ \mathfrak{A}\ \textbf{do}\ S\ \textbf{od}\ \{\mathfrak{B}\} \tag{1}$$

should be expressed logically as $\mathfrak{A} \Rightarrow \mathfrak{A} \vee \mathfrak{B}$, but this is out of sense. Ershov said that because it is natural there may be an interpretation such that it can be expressed in this way. Because to accept such an expressive form means to reject many of the best logical traditions, this idea was not realized until 3 years later as *program schemes logics*. These logics have *loop formation rules* such as

$$\frac{\mathfrak{A} \vee \mathfrak{B} \Rightarrow \mathfrak{B} \vee \mathfrak{C}}{\mathfrak{A} \Rightarrow \mathfrak{C}} \tag{2}$$

where $\mathfrak{A}$ can be interpreted as a precondition, $\mathfrak{B}$ as a loop invariant and $\mathfrak{C}$ as a postcondition. This rule in some sense implies the "mad" *infinite loop rule*

$$\frac{\mathfrak{A} \Rightarrow \mathfrak{A}}{\neg \mathfrak{A}} \tag{3}$$

1. How a constructive approach to programming came to light?

It had been mentioned as early as at the beginning of our century that in modern mathematics two notions: "*to exist*" and "*to be constructed*" are different. Many theorems have been proved which state the existence of some objects without giving any way to construct them. The most striking example is the Lebesgue unmeasurable subset of [0,1]. It had been proven that *each set of reals which can be defined in the set theory ZF cannot be proved to be unmeasurable.* Moreover, it had been shown by L.E.J. Brouwer that the roots of this divergence are logical. Namely, the usual logical principle *tertium non datur* $\mathfrak{A} \vee \neg\mathfrak{A}$ implies (in each sufficiently strong theory) such a formula $\exists x \in \mathbb{N}\ A(x)$ that $A(n)$ cannot be proved for each concrete $n \in \mathbb{N}$. Roughly speaking, it follows from the Gödel Incompleteness Theorem and from the fact that *tertium non datur* claims decidability of each problem.

The theory is *constructive* if each proof contains an (implicit or explicit) construction of objects which are proved to exist and all functions which can be defined by formulas of the form $\forall x \in X\ \exists 1 y \in Y\ \mathfrak{A}(x, y)$ are computable.

There are three ways to overcome the above obstacle and to reach constructive theories.

First of all, we can restrict ourselves to *weak* theories and/or sublanguages. It has been proved, for example, that existence implies constructability in the elementary theory of real numbers. Well known Horn theories are also in some sense

constructive. Namely, if the disjunction of elementary formulas is proved in a Horn theory, then one of these sentences is proved.

Secondly, we can restrict our proofs to those that maintain constructivity, but here we have the disadvantages of non-classical logics without their advantages.

And finally, we can use non-classical logics which grant constructivity and we hope that they have some advantages by using non-classical expressive means and new proof methods.

The first known constructive logic was invented by Brouwer and formalized by Heyting in 1930. It is called *intuitionistic logic.* It uses the same languages as classical logic (propositional connectives &, $\vee$, $\Rightarrow$, $\neg$, quantifiers $\forall$, $\exists$) and all the classical axioms but two, *tertium non datur* and *double negation principle*: $\neg\neg\mathfrak{A}\Rightarrow\mathfrak{A}$. Thus, it can be viewed at first sight as the result of the realization of the second method above: to omit some proofs which do not give constructions. Each intuitionistically valid formula is also classic tautology. But some consistent intuitionistic theories can contradict classical logic; for example, in intuitionistic logic it is possible to express the judgement "$\mathfrak{A}(x)$ is undecidable" in the following form:

$$\neg\forall x \quad (\mathfrak{A}(x) \vee \neg\mathfrak{A}(x)). \tag{4}$$

Kolmogoroff and Heyting developed a new kind of interpretation of logic formulas. Intuitionistic formulas are understood as *problems.* Each problem demands the construction of some objects or some effective transformations of objects or some effective functionals transforming computable functions and so on. . . .

Kolmogoroff's interpretation was not a precise mathematical semantic because it remains indefinite which functionals can be regarded as *computable.* However its ideas are in the origins of constructive interpretation of programming activity.

The most important constructive connective is a *constructive implication* $\Rightarrow$. $\mathfrak{A}\Rightarrow\mathfrak{B}$ means that each solution of $\mathfrak{A}$ can be transformed in some uniform way into a solution of $\mathfrak{B}$. The method of this transformation is regarded as a solution of $\mathfrak{A}\Rightarrow\mathfrak{B}$, so, $(\mathfrak{A}\Rightarrow\mathfrak{B})\Rightarrow\mathfrak{C}$ demands *computable functionals* from functions transforming solutions of $\mathfrak{A}$ to solutions of $\mathfrak{B}$ into solutions of $\mathfrak{C}$. So, intuitionistic logic and many other constructive systems *implicitly* contain high order notions which can be expressed by a first order predicate (or even propositional) language. See undecidability of $\mathfrak{A}$ above.

Other constructive connectives correspond to the other usual transformations of solutions; conjunction (say) can be viewed in intuitionistic logic as a pair of solutions of both its conjunctive members. But the meaning of this connective can be changed in some constructive logics.

Furthermore, Kleene [7] had constructed the notion of *recursive realizability* for formulas of constructive arithmetic. Here effective transformations are treated as recursive functions. Because a program for recursive functions can be encoded by natural numbers (*Gödel numbers*), high order functionals can be also treated as recursive functions from Gödel numbers to Gödel numbers. It has been proved that we can *extract* from each constructive proof of an arithmetic sentence, a recursive

function realizing the proved theorem. Though Kleene's construction of realizations for an arithmetic theorem was rather ineffective, it provided a way forward to logic program synthesis.

Thus we can see that the very first review of constructive logic concepts can show us that it seems to be promising to state the mutual analogies of programming concepts and of constructive logics.

This has been attempted many times. The very first was the forgotten work of Curry [4] who showed that some kinds of proofs can be treated as *implicit* constructions of effective programs. Curry is famous due to his numerous original conceptions in mathematical logic. When Computer Science was emerging he proposed the original logic system not like the usual logic but in some sense constructive. Only 30 years later this approach has come into consideration again.

From 1968 there were numerous attempts to apply constructive logics to programming.

At first they were considered as a promising tool of program synthesis. Bishop [1] proposed the scheme

$$\textit{specifications} \rightarrow \textit{proof} \rightarrow \textit{program} \tag{5}$$

based on the idea that it suffices to extract a program from a constructive proof and after that there is no need to prove its correctness or to debug it.

In 1971, Constable [2] proposed to use Bishop's idea to develop an automatic program synthesizer based on constructive arithmetic. This was a naive but valuable experiment. It was shown that there are many obstacles and complexities in this straightforward approach. It resulted in the program system CL [3] which illustrates many peculiarities of applied constructive systems.

Constable's claims led to a more serious theoretical analysis. Kreisel [10] investigated many possible connections of constructive systems and programming and some possible constructive uses of classical systems. For example, his ideas opened the way to testing automatically whether the classical existential theorem $\forall x\, \exists y\, \mathfrak{A}(x, y)$ gives us an effective algorithm for computing y. He claimed that only a small part of the constructive proof really contains the desired construction. He pointed out many complexity and formalization technique problems arising when we try to use constructive systems. His work was the second after well forgotten Curry's where *real* analogies between constructive logic and programming were studied.

In 1973, Markov gave an informal but very deep analysis of real and misleading analogies between constructive mathematics and computer programming. He stressed that although the intuitionistic arithmetic proof of $\forall x\, \exists y\, \mathfrak{A}(x, y)$ *formally* contains the algorithm to transform x into y, *really* it is a mere theoretical result. Kleene's extraction algorithm appeals to a universal function; so, often the extracted algorithm is practically incomputable and almost always too complex. Moreover, proofs in the original system of constructive arithmetics are more complex than in the corresponding classical system, but it is known that the problem of proof search

in classical arithmetic is theoretically and practically undecidable. Thus Bishop's scheme in its original form cannot be realized.

There was one more important note in this lecture. It had been shown by the Soviet constructive school in the 1950s (see, e.g., [10]) that *many formulas of constructive arithmetic can be treated classically*; *so, many parts of a constructive proof cannot contain implicit construction for realization of our theorem.* Thus, a constructive proof can be divided into two parts: *active* and *passive.* The passive part is merely a correctness proof of constructions made in the active part. Some formulas, so called *normal* formulas, can occur in the active part only as premises of some rules transforming active statements. So, each formula which is used only to prove a normal formula (even not normal itself) can be omitted during extraction. This is close to the ideas of Kreisel. But in most of the following works these observations are not taken into account. Only the recent book [6] tries to utilise them partially.

In the second half of the 1970s constructive systems were used in some research projects. A very sophisticated and powerful system had been developed by Martin-Löf and his school [11]. It uses transfinite types and Kreisel's brilliant idea of *formulas as types.*

We must mention here the interesting works of Tyugu and Minc [17]. They proved that there is a natural constructive logic of the program synthesis system PRIZ and this logic is varying for different strategies of subproblem solving. For the most common case this logic is the *intuitionistic propositional calculus.*

A new step in constructive logic applications is Girard's idea of *Linear Logic* [5]. This logic is one more argument that different classes of programs and problems demand different constructive logics. This idea was stated in [13] which first systematically described some methods of using constructive logics as tools to develop new programming methods and to investigate some theoretical properties of programs and programming languages.

In the addition to the above we can point out that some of the background of program verification seems to be unsound. The constructivist paradigm has an implicit consequence that *each* mathematical construction is made by a (hidden) constructive proof. Because correct program development can be viewed as a mathematical activity, it is natural to accept the following hypothesis: *Each correct program for a precisely stated problem is the result of some (not expressed explicitly) mathematical demonstration.*

Therefore, the problem of program verification seems a bit idiotic: having a mathematical demonstration we at first make our best efforts to ignore it except for those parts which can be expressed in our programming language. And then we try to reconstruct the original proof. The main problem is not how to prove programs but *how to write correct programs and how to specify them to simplify their reconstruction.*

There are numerous ways to solve this problem. The constructive way cannot be viewed as the best but it has its own interesting peculiarities and possibilities.

Finally, the language of algorithmic logics consists of two poorly co-ordinated parts (logical conditions and program construction). Investigations of program and

algorithmic logic revealed many striking properties and many hidden inconsistencies in programming languages. They are also often useful to develop a constructive approach, but there is another tendency in constructive logics: to make a single language with two well co-ordinated interpretations, a logic one and a programming one. This sometimes allows us to see deeper, but often (when the corresponding programming concepts are not perfectly designed) it simply fails to work. Here (if our programming language is ill designed) algorithmic logics are out of comparison.

3. Some basic considerations and analogies

There is a frequent problem involved in attempting to bridge two domains. The easiest analogies are almost always misleading when we are interested in real, complex problems. On the other hand, deeper analogies and possible warnings usually cannot be explained (at least for a long time) by very simple model examples. Here we try to state a basic system of analogies and to explain some shortcomings of obviously more simple decisions.

What is a logical analogy of a program and a programmer's analogy of a proof?

The simplest way is to state that *proof is program.* So, we can pose a claim such as "*program = proof + control*" [9], i.e. that we can execute the process of a proof search as a program (as in pure Prolog). This decision fails completely when we have not totally defined the function, so, to apply a function we must first state that it can be applied to given data. It is completely misleading when our constructive implications $\mathfrak{A} \Rightarrow \mathfrak{B}$ are understood as *actions* transforming states $\mathfrak{A}$ into states $\mathfrak{B}$. These actions can be noninvertible and an attempt to compute an unfinished proof can lead to the same conclusions as attempting to cure an ill person without stating the diagnosis. So, a Prolog-based aberration that we can always accept Kowalsky's thesis is wrong. This analogy can be applied only in some exclusive cases.

The second attempt to save this simple analogy is to claim that a *completed* constructive proof is a program. This has its origins in the worth of Martin-Lof and Constable works, and the majority of linear logic application works. The scope of this analogy is wider, but there are some shortcomings.

It is known that intuitionistic formulas of the form $\neg\mathfrak{A}$ cannot contain any nontrivial part of a solution of the desired problem. In other classes of constructive logics we can point out analogous classes of formulas (classical formulas in program scheme logics, negative ones in linear and so on). Moreover, we can point out that even for formulas such as

$$\forall xyz \quad (\mathfrak{A}(x, y) \,\&\, \mathfrak{B}(y, z) \Rightarrow \mathfrak{C}(x, z)), \tag{6}$$

all constructions to find a concrete value of y can become useless for the completed program: y is used here and nowhere else, including the conclusion of the proved theorem. So, *passive* formulas which do not affect the desired program but are necessary to prove its correctness, can be a major part of the constructive proof. It is not so dangerous as in the previous case if they are not taken into account.

Some danger can arise if our functions can fail. If we try to compute *all* parts of our proof, we can try to compute functions used (say) in the *reductio ad absurdum*. Their applicability is proved based on wrong suppositions, and our computations can fail due to parts which could be omitted. As a striking example we can see the *infinite loop rule* (3) which can prove only passive formulas and annihilates the conditions of each implication used in the proof of $\mathfrak{A} \Rightarrow \mathfrak{A}$ (as the result $\neg\mathfrak{A}$ is stated).

Thus, we can formulate the following analogy.

(1) *Formulas and objects used in a proof are divided into active and passive ones. Each active item and no passive ones have their image in a program.*

The program is the image of the active part of a proof. The proof corresponds to the program together with its correctness proof.

In program verification, some people independently discovered that to prove a program we often have to introduce new values and expressions (*ghosts*), but this was perhaps first pointed out by Tseytin [16] in 1971. Our passive objects corresponds to ghosts.

After this weakening of the proof and program connections, we can strengthen them in another direction. Because during a complex problem solution we cannot grant that our problem formalization will remain the same and because all formalizations of real domains can be viewed only as incomplete and partially correct, we should grant maintenance of as large a part of our logic proof as possible when the problem, program and formalization are changed. Moreover, the same demands are useful in theoretical analysis. Here we may keep the corresponding logical and program notions as close one to another as possible. It demands many mutual co-ordinations of logic and programming notions which are tiresome and seem to be small and unpromising. This is necessary for the success of the following informal and formal investigations and analysis. But it is more prominent to prove strong theorems without revision of the basic notions which proved their valuability in theoretic logic investigations. We choose the more difficult way.

It was pointed out during the preparation of our very first work [12] in this area that the level of corresponding program and proof construction is different. Programming was a bit forward in the expressiveness and design of constructions, logic was incomparable in the severe choice of only basic primitives. The first step (see [13]) was to develop a well co-ordinated programming language and logic calculus. None of the achievements of either area can be sacrificed during this process. It becomes possible to grant the following strong condition.

(2) *Each construction of a program is the image of a single construction of the proof. Each active construction of a proof in turn generates only one construction of a program. Sorts of* (*active*) *proof and program construction are in one-to-one correspondence.*

This co-ordination level of proofs and programs is a good basis for deep theoretical and practical invasions.

The second analogy gives a collection of more special realisations which are nevertheless very important and enlightening (maybe even more valuable or practical). Now we list and illustrate them.

(3) (a) *Proof rules correspond to program statements.*

(b) *Subproofs correspond to blocks and procedures.*

(c) *Terms correspond to basic type values.*

(d) *Formulas correspond to compound type values.*

(e) *Only elementary formulas which are parts of disjunctions can transfer into a program (as conditions of loops and guards of guarded commands).*

For example, the loop generation rule (2) corresponds to the loop statment

$$\{\mathfrak{A}\}\ \mathbf{while}\ \mathfrak{B}\ \mathbf{do}\ f\ \mathbf{od}\ \{\mathfrak{C}\}. \tag{7}$$

Here $\mathfrak{A}$ and $\mathfrak{C}$ become comments to our program whilst $\mathfrak{B}$ is used in the **while** part of the statement and is explicitly used in our program.

Let us now consider another case. *Functional formulas*

$$\forall x_1 \ldots x_n \quad (\mathfrak{A}_1 \,\&\, \cdots \,\&\, \mathfrak{A}_m \Rightarrow \exists y_1 \ldots y_k \quad (\mathfrak{B}_1 \,\&\, \cdots \,\&\, \mathfrak{B}_p)) \tag{8}$$

correspond to procedures. x_i can be interpreted as input values, y_j can be understood as output values. What is the role of $\mathfrak{A}_l$ and $\mathfrak{B}_q$? They cannot always be interpreted simply as pre- and postconditions. This was the mistake made by Constable in his first work [2]. For example, if $\mathfrak{A}_i$ is in turn a functional formula, it is interpreted as function parameter and realization of the whole formula becomes a functional of second order. This consideration illustrates our analogies on variables and formulas.

Now subproofs will be considered. First, a subproof is used while proving an implication (constructive or not), say, to prove a functional formula. Here a subproof becomes a procedure body, and the whole rule does a procedure declaration.

Assume $\mathfrak{A}$	**function** $f(\alpha\colon a)\colon b$;	
$\cdots$	**begin**	
Prove $\mathfrak{B}$	$\cdots$	(9)
——	$f := b$;	
$\mathfrak{A} \Rightarrow \mathfrak{B}$	**end**;	

When induction is used, the subproof of the induction step

Assume $\mathfrak{A}(n)$
$\cdots$ (10)
Prove $\mathfrak{A}(n+1)$

becomes the loop body. Analogously for case analysis,

$\mathfrak{A}_1 \vee \cdots \vee \mathfrak{A}_n$

Assume $\mathfrak{A}_1$ ...	**Assume** $\mathfrak{A}_n$	(11)
$\cdots$	$\cdots$	
Prove $\mathfrak{B}$	**Prove** $\mathfrak{B}$	

the corresponding subproofs become part of the **case of** statement. **If-then-else** conditional statement is generated by case analysis using $\mathfrak{A} \vee \neg\mathfrak{A}$.

Summing up the previous considerations, we can claim:

(4) *Natural deduction proofs are closer to programs than Gilbert or sequential ones.*

Natural deduction proofs are characterized by the possibility of introducing assumptions, to deduce consequences from these assumptions and to forbid further use of assumptions and their consequences after our subgoal is reached. To reflect this, the proof (like programs in most languages) has a *block structure*; its blocks are subproofs; each subproof has its own assumptions which can be used only inside it and maybe some other local objects; subproofs can be embedded and form a *subproof tree*; the main proof is its root, and all blocks and procedures are subordinated to the main program.

The last analogy is less common but very fruitful for logical analysis.

In this section we consider intuitionistic theories without functions. Thus, terms are only variables and constants.

The most important classes of intuitionistic formulas according to their role in program extraction are the following:

(1) Constructions:

$$\forall x_1 \ldots x_n \quad (\mathfrak{A} \Rightarrow \exists y_1 \ldots y_l \mathfrak{B}(x, \tilde{y})).$$

(2) Higher order constructions:

$$\forall x_1 \ldots x_n \quad (\mathfrak{A} \,\&\, \mathfrak{B} \Rightarrow \exists y_1 \ldots y_l \mathfrak{B}(x, \tilde{y})).$$

(3) Classifications:

$$\forall x_1 \ldots x_n \quad (\mathfrak{A} \Rightarrow \mathscr{B}_1(\bar{\bar{x}}) \vee \cdots \vee \mathscr{B}_k(\bar{\bar{x}})).$$

(4) Criteria I:

$$\forall x_1 \ldots x_n y_1 \ldots y_l \quad (\mathfrak{A}(x, y) \Rightarrow \mathscr{B}(\bar{\bar{x}})).$$

(5) Criteria II:

$$\forall x_1 \ldots x_n \quad (\forall y_1 \ldots y_l \quad \mathfrak{A}(x, y) \Rightarrow \mathfrak{B}(\bar{\bar{x}})).$$

(6) Criteria III:

$$\forall x_1 \ldots x_n \quad ((\mathfrak{A}_1(x) \Rightarrow \mathfrak{A}_2(x)) \Rightarrow \mathfrak{B}(\bar{\bar{x}})).$$

(7) Rejections:

$$\forall x_1 \ldots x_n \quad (\mathfrak{A}(x) \Rightarrow \neg \mathfrak{B}(\bar{\bar{x}})).$$

(8) Connections:

$$\forall x_1 \ldots x_n \quad (\mathfrak{A}(x) \Rightarrow \mathfrak{B}(\bar{\bar{x}})).$$

(9) Facts:

$$P(t_1, \ldots, t_n) \text{ or } \neg P(t_1, \ldots, t_n).$$

Here $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{B}_i$ are conjunctions of atomic formulas, $\mathfrak{B}$ is a conjunction of constructions (maybe, of higher order also), $\mathfrak{B}_i(\bar{\bar{x}})$ means that *all* variables of x occur in *all* atoms of $\mathfrak{B}_i$, $\mathfrak{B}_i(\tilde{y})$ means that at least one variable of y occurs in each atom.

Formulas of the nine listed classes are called *standard.*

Example 1. $\neg(0=1)$ is a fact, $\forall xy\ (x=y \Rightarrow x \leqslant y)$ is a connection, $\forall xy\ \forall z\ (x<z\ \&\ z<y \Rightarrow x<y)$ is a criterion I, $\forall xy\ (\forall z\ (x \in z \Rightarrow y \in z) \Rightarrow x \subset y)$ is a combination of criterion II and criterion III, $\forall xyz\ (z>0 \Rightarrow y>x-z \vee y<x+z)$ is a classification, $\forall xy\ (y>0 \Rightarrow \exists z\ (y \cdot z=x))$ is a construction.

Each intuitionistic theory can be reduced to a standard one, preserving extracted programs.

(5) *During program extraction, constructions give us functions, higher order constructions give functionals, classifications give conditional statements. Other classes of formulas cannot generate any construction in the resulting program.*

Moreover, criteria and rejections allow us to omit some constructions which are necessary to prove the correctness of the resulting program but are useless during its execution. So, they can turn some values or even pieces of proof into *passive* ones, which have no images in the resulting program. Often the necessity to remember these constructions leads programmers to the decision to write down something like used criteria and rejections as comments to the program. They may write:

"*Case when* $x>0$ *cannot appear and is omitted*".

Connections cannot generate any program constructions and, in turn, cannot make some potentially active constructions passive. They usually disappear completely from the program, and from its comments. But they form that "knowledge base" which is necessary to understand the program and comments.

To end this section, we mention one comparatively simple result from 1978 (one year after the beginning of our work) which at first proved the power of constructive logical analysis. It had been proven that having a proof of a program we can reconstruct a *constructive* proof which cannot be more than twice as long and which allows us to extract a program for the same goal. So, if our specifications are strong enough, the unique way to prove a program is to remember its construction.

4. Incomplete proof structures and programming

Practical constructive reasoning is almost always incomplete; and the *whole* proof rarely can be transformed into a program. Thus, it is very interesting to develop such proof fragments that are sufficient to reach our principal goal (e.g. to extract programs from). These fragments help us (say) to avoid the problem of a *full* automatic proof search. It is known that this last is theoretically undecidable and very difficult in practice. We try to replace it by more realistic ones, for example by enlargement of fragments created during man-machine dialogue up to proof.

Our considerations are based on the notion of a *proof as a graph* [14]. Nevertheless, no prior knowledge is presumed because our explanation is rather informal.

Graph structures can represent not only the result of a proof search but also its various stages. This was stated in [14]. We start by considering a simple example.

Example 1. Let we consider a very simple calculus Ω_{00}^{0}. Its formulas are classical propositional formulas and constructive implications $\mathfrak{A} \Rightarrow \mathfrak{B}$ (read: "Each state $\mathfrak{A}$ can be transformed into state $\mathfrak{B}$"). The rules for constructive implications are the following:

$$\frac{\neg\mathfrak{A}}{\mathfrak{A}\Rightarrow\mathfrak{B}} \qquad \frac{\mathfrak{A}\Rightarrow\mathfrak{B} \quad \neg\mathfrak{B}}{\neg\mathfrak{A}} \qquad \frac{\mathfrak{A}\Rightarrow\mathfrak{B} \quad \mathfrak{B}\Rightarrow\mathfrak{C}}{\mathfrak{A}\Rightarrow\mathfrak{C}} \qquad \frac{\mathfrak{A}\supset\mathfrak{B} \quad \mathfrak{B}\Rightarrow\mathfrak{C} \quad \mathfrak{C}\supset\mathfrak{D}}{\mathfrak{A}\Rightarrow\mathfrak{D}} \tag{12}$$

Let us add a slightly unusual rule: a *gap rule*:

$$\begin{array}{l} \cdot\, \mathfrak{A}\Rightarrow\mathfrak{B} \\ \downarrow \\ \circ\, \mathrm{Gap} \\ \downarrow \\ :\, \mathfrak{C}\Rightarrow\mathfrak{D} \end{array} \tag{13}$$

$\mathfrak{A}\Rightarrow\mathfrak{B}$ and $\mathfrak{C}\Rightarrow\mathfrak{D}$ are not connected here syntactically. We can conclude every constructive formula from every other. It is obvious that each theory is Curry inconsistent if $\Omega_{00}^{0} = \Omega_{00} + \mathrm{Gap}$ (i.e. each formula is derivable). Ω_{00}^{0} will be called *a calculus of incomplete proofs for* Ω_{00}.

Nevertheless Ω_{00}^{0} is restrictive when it is considered as description of some stages of proof search. We can conclude only constructive implications from constructive ones. This means that *all* classical formulas used by incomplete proofs are completely proved. Each application of the gap rule can be considered as a subproblem: **derive** $\mathfrak{C}\Rightarrow\mathfrak{D}$ **from** $\mathfrak{A}\Rightarrow\mathfrak{B}$. Ω_{00}^{0} can be interpreted as a calculus representing one of the most general strategies of proof search: *divide the problem into simple subproblems.*

Similarly, we can construct incomplete proof calculi for $\Omega_{00}^{\rightarrow}$ and $\Omega_{00}^{\leftarrow}$ by adding the Gap rule. But here we must give new global conditions: there cannot be more than one Gap in an incomplete proof; for $\Omega_{00}^{\rightarrow}$ its conclusion must be the theorem; for $\Omega_{00}^{\leftarrow}$ its premise must be an axiom.

Example 2. Let we have the theory

$$A\Rightarrow B, \quad B\Rightarrow C, \quad C\Rightarrow D, \quad D\Rightarrow E; \qquad H\Rightarrow E, \quad J\Rightarrow H, \quad I\Rightarrow J;$$
$$A\Rightarrow K, \quad K\Rightarrow L. \tag{14}$$

The following are incomplete proofs of $A\Rightarrow E$ in different calculi:

Ω_{00}^{0}:

$$\frac{\dfrac{A\Rightarrow B}{A\Rightarrow C}\,\mathrm{Gap} \qquad \dfrac{\dfrac{I\Rightarrow J \quad J\Rightarrow H}{I\Rightarrow H}}{C\Rightarrow E}\,\mathrm{Gap}}{A\Rightarrow E}$$

$\Omega_{00}^{\rightarrow}$:

$$\cfrac{\cfrac{A\Rightarrow B \quad B\Rightarrow C}{A\Rightarrow C}}{A\Rightarrow E}\ \text{Gap}$$

$\Omega_{00}^{\leftarrow}$:

$$\cfrac{\cfrac{A\Rightarrow B}{A\Rightarrow D}\ \text{Gap} \quad D\Rightarrow E}{A\Rightarrow E} \tag{15}$$

It is intuitively clear that although $C\Rightarrow E$ and $A\Rightarrow E$ are derivable, the first incomplete proof is "bad"; it contains a gap between $I\Rightarrow H$ and $C\Rightarrow E$ which cannot be repaired. Second and third proofs can, of course, be enlarged up to a complete proof of our goal.

So, we can define *piece of a proof* Π in a calculus $\mathcal{I}$ as its full subgraph Θ such that it contains a rule iff all its premises and conclusions belong to Θ, and it contains a structure vertex (subproof and so on) iff all its subordinated formulas belong to Θ. If formula vertex $\mathfrak{A}$ is a conclusion of the rule not belonging to Θ, $\mathfrak{A}$ is called an *entry* of Θ. If $\mathfrak{A}$ is a premise of the rule not belonging to Θ, $\mathfrak{A}$ is called a *result* of Θ.

Now we can define the notion "$\mathcal{J}$ is a *calculus of incomplete proofs* for $\mathcal{I}$". Its main properties can be outlined as follows.

Formulas of $\mathcal{J}$ can be considered as metaexpressions for ones of $\mathcal{I}$. Rules of $\mathcal{J}$ are divided into two classes: rules of $\mathcal{I}$ and gaps. Each correct application of $\mathcal{I}$-rule π in the calculus $\mathcal{J}$ is transformed into its correct application in $\mathcal{I}$ by substitution for all metaexpressions.

A relation "proof Π of $\mathcal{I}$" is an *enlargement* of an $\mathcal{J}$-proof Σ. $\mathfrak{E}(\Pi, \Sigma)$ is defined such that there is an injective map ψ of $\mathcal{I}$-rule applications of Σ into the same rule applications of Π and there is a piece $\Theta(\delta)$ of Σ for each gap rule δ of Π such that its entries can be mapped one-to-one onto premises of δ. This results in conclusions of δ, and there is a substitution for all metaexpressions occurring into premises and conclusions of δ, resulting in corresponding information into entry and result vertices of $\Theta(\delta)$.

Usually proofs of $\mathcal{J}$, viewed as an incomplete proof calculus for $\mathcal{I}$, are divided into two classes: *complete* and *incomplete* w.r.t. $\mathcal{I}$. In most cases, the proof is complete iff it does not contain gap rules. An incomplete proof is *enlargable* if it has an enlargement.

Example 3. Our calculi Ω_{00}^{0}, $\Omega_{00}^{\rightarrow}$, $\Omega_{00}^{\leftarrow}$ are incomplete proof calculi for Ω_{00}. The first proof of Example 3 is not enlargable for our theory *Th*. The second and third can be enlarged obviously.

A calculus of incomplete proofs $\mathcal{J}$ is *adequate to* a proof search strategy $\mathcal{S}$. If each stage of the proof search according to $\mathcal{S}$ corresponds to an incomplete proof

of $\mathcal{J}$ and if Π is an incomplete proof of $\mathcal{J}$, corresponding to some stage of the proof search resulting accordingly to $\mathcal{S}$ in a proof Σ, and there occurs no backtrackings on the way from Π to Σ according to $\mathcal{S}$, then $\mathfrak{C}(\Pi, \Sigma)$. $\mathcal{J}$ is *fully adequate* to $\mathcal{S}$, if each incomplete proof of $\mathcal{J}$ corresponds to some proof search stage of $\mathcal{S}$.

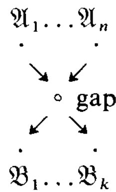
For example, $\Omega_{00}^{\leftarrow}$ and $\Omega_{00}^{\rightarrow}$ are fully adequate for inverse and direct proof search strategies, respectively.

For the first time incomplete proofs have been introduced for the natural deduction intuitionistic logic to perform logic analysis of so called "structured gotos", exit operators or "continuations". Here we outline this construction.

An exit statement serves as an escape from some innermost procedures during structured program execution if they become obsolete. For example, if we search for data in deeply hierarchical structures and the desired data are found, all introduced search trees, all search and access procedures are of no use; we may return to the point where these data were requested.

Intuitionistic logic provides a good basis for structured functional programming. It has helped us to discover many fine peculiarities of high order programming (programming by using and introducing functionals of higher types; do not confuse it with typeless functional programming). There are interesting logical analogies for functionals, data flow, loops, recursion... Why not for structured gotos?

Let IPC_0 be the intuitionistic predicate calculus without $\vee$ and $\neg$. We introduce an incomplete proof calculus $\mathcal{FS}_0^{\downarrow}$ by adding the *gap rule* of the form



where $\mathfrak{A}_i$ are IPC_0 formulas and $\mathfrak{B}_j$ can contain metavariables. Though premises and conclusions of this rule are completely independent as formulas, global conditions imply that they belong to the same subproof. A subproof containing a gap rule is called *non-completed.* Global conditions are extended by the demand that each subproof can contain no more than one gap.

A $\mathcal{FS}_0^{\downarrow}$ proof is *complete* if in each gap $n = k$, and there are substitution terms $\mathfrak{t}$ for metavariables such that after the substitution $\mathfrak{A}_i = \mathfrak{B}_i[\mathfrak{x}|\mathfrak{t}]$. If such substitution exists for a subproof and its subordinated subproofs then the subproof is *completed.* Complete proofs can be easily transformed into $\mathcal{FS}_0$ proofs.

$\mathcal{FS}_0^{\downarrow}$ corresponds to a global top-down proof search strategy but allows for each subgoal to combine forward and backward search strategies. To restrict ourselves by forward search only, it suffices to demand that each conclusion of the gap is a result of the subproof ($\mathcal{FS}_0^{\downarrow\rightarrow}$ calculus).

$\mathcal{FS}_0^{\downarrow\rightarrow}$ calculus allows us to find logic analogs of exit statements. Namely, a subproof is to be developed until our current results (premises of the gap) can be

unified with the desired ones (subproof results which are determined by the FD rule). But what happens if we deduce the desired results of a greater subproof?

This situation is very similar to exit, a more global goal having been reached than the current one. To finish the formalization of this feature, it is sufficient to introduce the *goal transfer rule* (GT) (it was called in [15], slightly ironically, the *good surprise rule*) which allows us to place the goals of the embedding proof into the embedded one, and to define *quasicomplete proofs*, if each gap has as conclusions the whole set of formulas subordinated to some goal of the current subproof or to a transferred goal and its premises and conclusions are unified.

Proposition. *$\mathfrak{A}$ is derivable from Th by a quasicomplete proof of IPL_0^+ iff $\mathfrak{A}$ is a classical theorem of Th.*

So, exit and high order functionals are in some sense incompatible; exit can destroy the structure of our arguments and lead us to inconstructivity when the extracted solution cannot be executed correctly. It has been stated that transformation semantics cannot show this failure. The results of our analysis clarified some roots of the errors in the ELBRUS system (its hardware supports high order functional programming) and were taken into account by its designers.

There is one more important notion induced by graph proofs, *proof fragments.* Vertices v of proofs are labelled by $\mathrm{Inf}(v)$. Let a partial ordering $\leqslant$ be given for information entities $\mathrm{Inf}(v)$. If we omit a formula preserving its free variables, it can be considered as an example of $\leqslant$. Let Π be a proof in a calculus $\mathscr{L}$. Graph Σ is a *fragment* of Π if there is an injective map φ of its vertex set V_Σ into the vertex set V_Π, and the type of v is the same that $\varphi(v)$ and $\mathrm{Inf}(\varphi(v)) \geqslant \mathrm{Inf}(v)$ and there is a path from $\varphi(source(\alpha))$ into $\varphi(destination\ (\alpha))$. If ξ is an algorithm totally defined on pairs of graphs, then Σ is called a *ξ-fragment* of Π, if Σ is a fragment of Π and $\xi(\Sigma, \Pi) = 0$.

This formal definition is inspired by considering a fragment as a graph with some vertices deleted and the information of the remainder weakened down according to some rules (these last can be tested by ξ). The resulting graph preserves in some sense the structure of the whole proof (this is expressed by the demand on the paths). It is natural to consider fragments as proofs in some generalized calculus $\mathscr{L}$, which can be constructed analogously to incomplete proof calculi. ξ itself is described, as a rule, by outlining informally the decidable condition which is to be tested.

Example 4. For each calculus $\mathscr{C}$ there is a natural fragment calculus; its global conditions are the same and all its information vertices are omitted. This calculus will be called *structure calculus* for $\mathscr{C}$, and its proofs *proof structures* of $\mathscr{C}$.

We can see here all the subproofs, rules and data flows.

Proposition. *If the relation $\leqslant$ is decidable, the problem whether Σ is a fragment of Π is decidable.*

Definition 1. The *problem of ξ-correct enlargement up to proof in a generalized calculus $\mathcal{N}$ is decidable for $\mathcal{L}$*, if there is an algorithm λ, which is total on derivations of $\mathcal{L}$ and a derivation Σ is processed into false; if it is not a ξ-fragment of any proof Π of $\mathcal{N}$, and into such proof Π otherwise.

Definition 2. $\mathcal{M}$ is called a *calculus of ξ-skeletons* for $\mathcal{N}$, if the problem of ξ-correct enlargement up to proofs in $\mathcal{N}$ is decidable for $\mathcal{M}$-derivations, and each ξ-fragment of each proof of $\mathcal{N}$ is a $\mathcal{M}$-derivation.

Let us consider the calculus obtained by weakening some rules of the intuitionistic calculus $\mathcal{FS}$ in the following manner. The functional formulas $\forall x\ (\mathfrak{A}(x)\Rightarrow\exists y\ \mathfrak{B}(x,y))$ are to be given explicitly; all arguments $\mathfrak{A}[x|t]$ can be omitted; all results $\mathfrak{B}[x|t]$ can be omitted also. The deduction rule FD (to prove functional formulas by a subproof of $\mathfrak{B}$ from $\mathfrak{A}$) can be weakened by omitting formulas $\mathfrak{A}$, $\mathfrak{B}$. Derivations in this calculus are called *typizable skeletons.*

The enlargement problem is decidable for typizable skeletons.

This kind of skeleton can be considered as a way to make the problem of finding data structures having the structure of function calls (the problem of *typization* for a functional program) more precise. Each rule applying a functional formula defines a call in an extracted program. All axioms are given; our goal is also given. So, the omitted formulas define data types of values. Because these types can be easily reconstructed, it is not reasonable to demand explicit data declarations in a functional program. If actual and formal parameter types do not match, our program contains a semantic error and this error can be detected automatically by our compiler. Thus one of the main demands of structured programming - to specify types of all variables - cannot be regarded as universal.

Let us now weaken the rules in another manner: we can omit the main premises (functional formulas) and arguments $\mathfrak{A}$ are to be preserved. Let the FD rule be weakened by omitting *all* formulas, preserving only the structure of the subderivations; let the axiom generation rule be weakened by omitting its conclusion, an applied axiom. The resulting calculus is called the calculus of *argument skeletons.*

The enlargement problem is also decidable for argument skeletons. Argument skeletons can be considered as problems of synthesis of a functional program having their data structures and the structure of data flow. It can be seen that these problems can also be solved automatically. So, it is sufficient to give explicitly only one side of the coin, whether functions or their arguments. The other side can be reconstructed automatically. But you may remember that our arguments are based on the assumption that all our functions (data) are completely specified in logical language.

What is the weakest fragment which can be regarded as a skeleton? We, obviously, cannot give a precise answer, but it suffices only to give the proof structure.

There are no known lower bounds for the complexity of the structure enlargement problem, but all known algorithms are more than exponentially hard (formula

unification is a difficult problem, and there are numerous backtrackings). This kind of skeleton can be considered as formalization of the program analysis problem.

Finally, we can see that the problem of enlargement of fragments including gaps is, of course, undecidable. But the problem of ζ-correct enlargement can be decidable. For example, let our *Th* consist of standard axioms. Let fragments contain gaps, and let ζ test whether the proof piece substituted for each gap consists only of criteria, rejections and connections. This is called a *constructive skeleton.*

Proposition. *The problem of ζ-correct enlargement is decidable for constructive skeletons.*

Constructive skeletons can be interpreted as the formulation of *program verification* problems. All constructions and classifications, giving us functions and conditional statements, are given explicitly. We may only test conditions on their input and output values.

5. Conclusion

These are only some insights into a strange new world, Applied Constructive Logic. It is interesting not only due its new possibilities but as a domain demanding unusual formalization techniques, hard logical considerations, and new insights into old ideas.

References

[1] E. Bishop, *Foundations of Constructive Analysis* (New York, 1967) 284 pp.

[2] R.L. Constable, Constructive mathematics and automatic program writers, in: *Information Processing 71*, vol. 1, Amsterdam (1972) 229-233.

[3] R.L. Constable et al., *Implementing Mathematics with the Nurpl Proof Development System* (Prentice Hall, Englewood Cliffs, NJ, 1986).

[4] H.B. Curry, The logic of program composition, *Collect Logique Math., Paris* **A5** (1954) 97-102.

[5] J.-Y. Girard, Linear logic, *Theoret. Comput. Sci.* **50** (1987) 1-102.

[6] S. Hayashi and H. Nakano, *PX: A Computational Logic* (Cambridge, MA, 1988) 200 pp.

[7] S.C. Kleene, On the interpretation of intuitionistic number theory, *J. Symbolic Logic* **10** (1945) 109-124.

[8] D.E. Knuth, *Algorithms in Modern Mathematics and Computer Science*, Lecture Notes in Computer Science **122** (Springer, Berlin, 1981) 82-99.

[9] R. Kowalsky, Algorithm = logic + control, *Comm. ACM* **22**(7) (1979) 424-436.

[10] G. Kreisel, Some uses of proof theory to improve computer programs, in: *Logic Colloquium*, Clermond-Ferrand (1975).

[11] P. Martin-Löf, Constructive mathematics and computer programming, in: L.J. Cohen et al., eds., *Logic, Methodology and Philosophy of Science*, VI (North-Holland, Amsterdam, 1982) 153-179.

[12] N.N. Nepejvoda, A relation between the natural deduction rules and operators of higher level algorithmic languages, *Soviet Math. Dokl.* **19**(2) (1978) 360-363.

[13] N.N. Nepejvoda, On correct program construction, *Voprosy Kibernet* (Moscow) **46** (1978).

[14] N.N. Nepejvoda, Proofs as graphs, *Semiotikai Informatika* (*Semiotics and Informatics*) **25** (1985) 52–82 (in Russian).
[15] N.N. Nepejvoda, The *good surprise* rule and structured gotos, *Semiotika i informatika* (*Semiotics and Informatics*) **23** (1984) (in Russian).
[16] G.S. Tseytin, Some features of a language for a proof-checking programming system, Lecture Notes in Computer Science **5** (Springer, Berlin, 1974) 394–407.
[17] E. Tyugu and G. Minc, Justification of the structural synthesis of programs.
[18] J. Venn, *Symbolic Logic* (London, 1894).