JUSTIFICATION OF THE STRUCTURAL SYNTHESIS OF PROGRAMS

G. MINTS and E. TYUGU

Institute of Cybernetics, 21 Akadeemia tee, Tallinn, 200026 U.S.S.R.

Communicated by M. Sintzoff
Received October 1982
Revised February 1983

Abstract. We present a constructive description of the automatic program synthesis method used in the PRIZ programming system. We give a justification of the method by proving the completeness of its inference rules for the class of constructive theories, and we present rules for transforming any intuitionistic propositional formula into a form suitable for these inference rules.

1. Introduction

There are several programming systems using automatic theorem proving for program synthesis, which employ the schema

\[
\text{SPECIFICATION} \longrightarrow \text{PROOF} \longrightarrow \text{PROGRAM}. \quad (\ast)
\]

The quite familiar system PROLOG works with a specification written as a finite set of Horn clauses and constructs a proof essentially in the classical predicate calculus.

The 'structural synthesis' approach considered in this paper and used in the system PRIZ [25] (a program product installed on more than 200 Ryad computer mainframes), deals with a specification written as a propositional formula, and constructs proofs in a version of the intuitionistic propositional calculus. The range of applications of the method turned out to be much more extensive than one might suspect [20, 21, 26]: an application example is presented in Section 8. This can be explained by the completeness of the structural synthesis for the intuitionistic propositional calculus, recently discovered [18] and presented here. The second stage in (\ast), that is the extraction of the program from a proof, in the PRIZ system uses the same basic ideas as the standard interpretation for the intuitionistic system. The latter was outlined by Heyting and Kolmogorov [9, 12], made precise by Kleene [10] and Gödel [8] in the form of realizability interpretations, and adapted to the programming context by numerous authors [3, 7, 23, 17, 16].

The idea of the structural synthesis is that a proof can be built and that the overall structure of a program can be derived from the proof, knowing very little
about the actual properties of the functions used in the program. We assume that
our problem specification contains information only about the applicability of
functions for computing values of variables which occur in the problem specification.
Some of this information is encoded in implicative formulas of the form

\[ A \land \cdots \land B \rightarrow C \]

(1)

which have a logical meaning "\( A, \ldots, B \) implies \( C \)" as well as a computational
meaning "\( C \) is computable from \( A, \ldots, B \)". In the first case \( A, \ldots, B, C \) are
considered as atomic formulas, in the second case as computable objects according
to the familiar constructive interpretation of an implication \( A \land \cdots \land B \rightarrow C \), viz.
a problem constructing procedure (e.g. a computation of \( C \) from \( A, \ldots, B \)) transforming any realization of \( A, \ldots, B \) into some realization of \( C \). Sometimes we
shall explicitly indicate this transformation by writing 'f' under the arrow:

\[ A \land \cdots \land B \rightarrow C. \]

Such formulas constitute a more restricted language than PROLOG because no
free variables are allowed in the atomic formulas \( A, \ldots, C \). But we also allow
nested implications like

\[ (A \rightarrow B) \land \cdots \land (C \rightarrow D) \rightarrow (E \rightarrow F) \]

(2)

which add generality to the language. They introduce procedures of higher types,
since the interpretation of formula (2) is a computation of \( F \) from the realizations
of \( A \rightarrow B, \ldots, C \rightarrow D \) and of \( E \). In this way formulas (2) allow implicitly to use
objects of all finite types by introducing new propositional variables through
transformations described in Section 5. These reduce any propositional formula to
a form the structural synthesis deals with. Actually, the transformations considered
in Section 5 enabled us to prove very efficiently all theorems of the intuitionistic
propositional logic (about 100 formulas) contained in [10]. In this case the existing
program synthesizer was used as a theorem prover [27], which is just the way
opposite to the usual suggestions to use a theorem prover for program synthesis.

Let us note that the PRIZ system was originally developed as a practical programming system, entirely from data flow considerations; only much later its authors became aware of its logical ground.

Some projects described in the literature [19, 17] deal with specifications given
by formulas in the language of the first-order, or even higher-order, arithmetic and
use some form of realizability interpretation to extract programs from constructive
proofs. The theoretical and practical difficulties inherent in attempts to find such
proofs automatically are analyzed in [14]. We know two systems of this kind [1, 3]
which were actually implemented. Since their applicability range formally includes
the range of PRIZ, it would be interesting to compare the performance of these
systems on the latter range.

In the present paper we introduce the structural synthesis step by step. In Section
3 we describe a language in which programs are derived, and then a logical language
together with inference rules for the automatic construction of proofs. In Section 4 we use a combination of these languages to explain the extraction of programs from proofs. In Section 5 we give a justification of the structural synthesis rules firstly by proving their completeness, and secondly by presenting the reduction of arbitrary intuitionistic propositional formulas to our form.

In the PRIZ system the proofs are constructed completely automatically. Therefore the efficiency of theorem proving is the crucial point of the system. A data structure for automatic theorem proving for structural synthesis is considered in Section 6. Data flow considerations, described in Section 6, appeared extremely helpful for improving the performance of the theorem prover.

An extension of the structural synthesis technique for recursive programs is considered in Section 7, and an example of recursive synthesis is presented in Section 8.

2. General schema of the structural synthesis and an example

Our general philosophy for developing a program to solve a given problem is the following. The problem is specified in a problem-oriented language (Fig. 1) and has the form:

\[ \text{compute } y \text{ from } x \text{ when } Q. \]

This specification is then translated into a set of axioms \( T \) in a logical language \( (LL) \) in which the theorem \( T \vdash \exists f(x \rightarrow y) \) is constructively provable for each solvable problem; having the proof, a program is derived from it, which satisfies the specification.

![Diagram of the way from a problem to a program](Image)

Fig. 1. The way from a problem to a program.
We distinguish between two levels of problem specifications, as shown in Fig. 1. The first one is a source specification $Q$ given in a problem-oriented specification language. The second level is an extended specification in the form of a formal theory $T$ in which the solution existence theorem is constructively provable for the problem, if this one is solvable. The transformation of a source specification $Q$ into an extended specification $T$ is an essential feature of the structural synthesis. It includes the derivation of axioms in the form (1), (2) from relations given in $Q$. In particular, solvable equations $P(x, y) = 0$ give axioms $x \overset{\tau_1}{\rightarrow} y$ and $y \overset{\tau_2}{\rightarrow} x$, where $f_1, f_2$ are the solution functions for $P(x, y) = 0$.

Let us look at a simple problem, shown in Fig. 2. We must compute the $x$-coordinate of the point $C$ from the given angle of the bar $AB$. We present this problem using the concepts 'point' and 'bar' which we describe using a common notation for specifying abstract data types. Actually this example is written in the source language of the PRIZ system.

**point:** $(x, y : \text{real})$;

**bar:** $(P, Q : \text{point})$;

\[
\text{length, angle : real;}
\]
\[
\text{rel length} \uparrow 2 = (O . x - P . x)^2 + (O . y - P . y)^2;
\]
\[
\text{rel cos angle} = (Q . x . - P . x)/\text{length}.
\]

Here $Q . x, P . y, \text{etc. mean } x \text{ of } Q, y \text{ of } P, \text{etc.}$ Having specified these two concepts we can present the whole problem:

\[
u, v : \text{real;}
\]

**$AB$: bar** $\text{length} = 1.1, \text{angle} = u, P = (0, 0)$;

**$BC$: bar** $P = AB . Q, \text{length} = 1.8, Q . y = -0.5, Q . x = v$;

*compute* $v$ from $u$.  

---

Fig. 2. A computational problem.
Here \( u \) is an input and \( v \) an output variable of the problem. The second and third lines specify two bars \( AB \) and \( BC \). The last line presents the goal.

The specifications in the problem-oriented language are translated into the logical language using a generalized macro technique. The translation is rather trivial, by unfolding of specifications.

\[
\text{point}: (x, y: \text{real})
\]
is unfolded to the following three axioms:

\[
\begin{align*}
\text{point} & \rightarrow \text{point} . x, \\
\text{point} & \rightarrow \text{point} . y, \\
\text{point} . x, \text{point} . y & \rightarrow \text{point},
\end{align*}
\]

where \( a \rightarrow b \) means "\( b \) is computable from \( a \)". The unfolding of the specification of \( \text{bar} \) gives us 20 axioms which are not reproduced here. The sample problem specification is unfolded to

\[
\begin{align*}
\rightarrow AB : \text{length} \\
u & \rightarrow AB . \text{angle} \\
AB . \text{angle} & \rightarrow u \\
& \rightarrow AB . P
\end{align*}
\]

plus 20 axioms for \( AB \), inherited from \( \text{bar} \), and

\[
\begin{align*}
AB . Q & \rightarrow BC . P \\
BC . P & \rightarrow AB . Q \\
& \rightarrow BC . \text{length} \\
& \rightarrow BC . Q . y \\
v & \rightarrow BC . Q . x \\
BC . Q . x & \rightarrow v
\end{align*}
\]

plus 20 axioms for \( BC \), inherited from \( \text{bar} \).

In our notation, the theorem we must prove is

\[
u \rightarrow v.
\]

For this kind of problems, the form of axioms is so simple that an automatic deduction algorithm exists which has linear (!) time complexity. It gives the following sequence of computations as a result:

1. \( u \rightarrow AB . \text{angle} \)
2. \( \rightarrow AB . \text{length} \)
3. \( \rightarrow AB . P \)
4. \( AB . P \rightarrow AB . P . x \)
5. \( AB . P \rightarrow AB . P . y \)
6. \( AB . \text{angle}, AB . P . x, AB . \text{length} \rightarrow AB . Q . x \)
7. \( AB . \text{length}, AB . P . x, AB . P . y, AB . Q . x \rightarrow AB . Q . y \)
8. \( AB . Q . x, AB . Q . y \rightarrow AB . Q \)
9. \( AB \cdot Q \rightarrow BC \cdot P \)
10. \( BC \cdot P \rightarrow BC \cdot P \cdot x \)
11. \( BC \cdot P \rightarrow BC \cdot P \cdot y \)
12. \( BC \cdot \text{length}, BC \cdot P \cdot x, BC \cdot P \cdot y, BC \cdot Q \cdot y \rightarrow BC \cdot Q \cdot x \)
13. \( BC \cdot Q \cdot x \rightarrow v. \)

This sequence can be looked upon as a proof of computability of \( v \) from \( u \) as well as an algorithm for computing \( v \) from \( u \).

The example above is designed to illustrate that (1) automatic theorem proving for program synthesis purposes need not be time consuming, and (2) the source language for an automatic program synthesizer can be a user-oriented specification language.

The language we used in the example is almost the same as "If you See What I Mean" (ISWIM) presented by P. Landin more than 15 years ago [15]. But that is also our idea about a simple and user-friendly specification language.

We do not think that this trivial example by itself can convince the reader of the usefulness of the program synthesis. We intend to do this by showing the generality and by proving the completeness of the structural synthesis theories. From now on we dismiss the problem-oriented specification language and concentrate on theorem proving, on program derivation, and on languages needed for these purposes.

3. Structural synthesis theories

3.1. Programming language (PL)

We shall use a typed functional language as a programming language. We assume a finite number of primitive types: \( \pi_1, \pi_2, \ldots, \pi_k \).

We assume that there exist an infinite number of variables \( x^\sigma, y^\sigma, \phi^\sigma, \psi^\sigma, x_1^\sigma, \ldots \) for each type \( \sigma \). Sometimes we omit the type index \( \sigma \) and write \( x, y, \ldots \) for variables of primitive types and \( \phi, \psi, \ldots \) for higher types.

We assume the existence of constants of primitive and nonprimitive types: \( a, b, f, g, F, G, F_1, \ldots \).

Terms are built from variables and constants using parentheses and the symbol \( \lambda \):

1. Constants and variables are terms;
2. If \( t \) is a term of type \( (\sigma_1, \ldots, \sigma_n : \tau) \) and \( \overline{s} = s_1, \ldots, s_n \) are of types \( \sigma_1, \ldots, \sigma_n \), then \( t(\overline{s}) \) is a term of type \( \tau \). (The value of the term \( t(\overline{s}) \) is intended to be the value derived from the value of \( \overline{s} \) by the function which is the value of \( t \));
3. If \( t \) is a term of type \( \tau \) and \( x^\sigma \) is a variable of type \( \sigma \), then \( \lambda x^\sigma t \) is a term of type \( (\sigma : \tau) \); \( x^\sigma \) is a bound variable in this term. In general, \( t \) may contain occurrences of \( x^\sigma \) and this is expressed by the notation \( t[x^\sigma] \); \( \lambda x^\sigma t[x^\sigma] \) is a function yielding \( t(\alpha^\sigma) \) for the given value \( \alpha^\sigma \).
Let us use a common metanotation $t_x[r]$ to express the substitution of $r$ for all occurrences of $x$ in $t$ (with renaming of bound variables to avoid collisions). Then the usual semantics of $\lambda$-terms is given by

$$(\lambda x t)(r) = t_x[r].$$

The language PL is obviously weaker than Gödel's language T [8] due to the absence of recursion constants.

The constants of PL represent programs which are available to the synthesizer. In this sense the set of constants is potentially infinite. New programs are obtained and new constants are realized for instance when an equation given in a problem specification is translated into the internal language.

A program may be a term with free variables of primitive types only. At first glance this programming language prevents a programmer from using usual constructs of programs (if-then-else and while-do). In fact these constructs can be introduced using constants with a predefined realization (which is not part of PL).

The conditional expression

$$\text{if } p \text{ then } f \text{ else } g \text{ fi}$$

can be presented by a term

$$C^{(\sigma_1, \text{bool}, \sigma_2, \sigma_2)}(p^{(\sigma_1, \text{bool})}, f^{\sigma_2}, g^{\sigma_2}),$$

where $C$ is a constant and $\text{bool}$ is a primitive type. The realization of $C$ must be supplied by a programmer and must have the properties:

$$t_1 \to C(t_1, t_2, t_3) = t_2, \quad \neg t_1 \to C(t_1, t_2, t_3) = t_3.$$

Recursion can be expressed by means of constants $R$ of type $(\sigma, (\text{nat}, \sigma), \text{nat}:\sigma)$ with a realization given by a programmer and satisfying

$$R(r, s, 0) = r, \quad R(r, s, \text{next } (u)) = s(u, R(r, s, u)),$$

where $0$ is the zero constant and $\text{next}$ is a constant realized by the successor function.

Finally we remind that PL is not a language for writing programs. It is an internal language in which programs are synthesized automatically. In this case the convenience of reading and writing in it is not as important as the possibility of transforming programs into efficient code. This transformation is trivial, as long as we only have non-recursive function definitions, which is the case here. The transformation consists in the elimination of repetitive computations of identical sub-expressions by introducing assignments, and in the elimination of computations of unused variables. The latter has been described in [25].

3.2. The logical language (LL)

This is an internal language in which theorems are proved.

It has only three kinds of formulas:
(1) propositional variables: $A$, $B$, etc.;

(2) formulas called unconditional computability statements:

$$A_1 \land \cdots \land A_k \to B$$

or in a shorter way: $A_k \to B$;

(3) formulas called conditional computability statements:

$$\bigwedge_{1 \leq i \leq n} (A_k^i \to B^i) \to (C_m \to D).$$

Besides the formulas, LL also includes expressions which are called sequents: $\Gamma \vdash X$, where $\Gamma$ is an unordered list of formulas and $X$ is a formula. The turnstile $\vdash$ plays the role of a delimiter between the goal $X$ and assumptions $X_1, \ldots, X_k$. It is treated like $\to$ in $X_1 \land \cdots \land X_k \to X$.

Let us informally explain the meaning of formulas. A propositional variable is used in one sense only. It corresponds to some variable from the source problem, and it expresses the fact that a value of this variable can be computed.

An unconditional computability statement $A_1 \land \cdots \land A_k \to B$ expresses the computability of (a value of the variable) $b$ corresponding to $B$ from (values of) $a_1, \ldots, a_k$, which correspond to $A_1, \ldots, A_k$.

A conditional computability statement, for instance,

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

expresses the computability of $d$ from $c$ depending on the computation of $b$ from $a$. (As above, the lower-case letters $a$, $b$, $c$, $d$ are the variables corresponding to $A$, $B$, $C$, $D$.)

We have already used unconditional computability statements in our sample problem in Section 2, using the names of variables from the source problem as propositional variables.

Let us encode in our logical language the problem of computing the double integral

$$s = \int_0^a \int_0^b z \, dx \, dy, \quad z = g(x, y).$$

First of all, propositional variables $S$, $A$, $B$, $X$, $Y$, $Z$ and $W$ must be introduced, $W$ standing for the computability of $w = \int_0^b z \, dx$.

The computability statements are

$$X \land Y \to Z$$ \hspace{1cm} (for $z = g(x, y)$),

$$(X \to Z) \to (B \to W)$$ \hspace{1cm} (for $w = \int_0^b z \, dx$),

$$(Y \to W) \to (A \to S)$$ \hspace{1cm} (for $s = \int_0^a w \, dy$).

The goal is to prove $A \land B \to S$. 

3.3. Derivation of formulas

To complete the description of the class of theories we are dealing with, we define a derivation of a formula and give the inference rules for building the derivations. We use a sequent notation \([6]\) for expressing the derivability of formulas. A sequent \(\Gamma \vdash X\) means that the formula \(X\) is derivable from the formulas appearing in \(\Gamma\). \(\Sigma, \Gamma\) means concatenation of the lists \(\Sigma\) and \(\Gamma\), with contraction of repetitions of formulas in the resulting list. For any formula \(X\), any list \(\Gamma\), and any theory considered here, the sequent \(\Gamma, X \vdash X\) is an axiom. Traditionally we call it a logical axiom. Beyond this we have axioms in the form of \(\vdash X\), where \(X\) is a computability statement. These axioms are called specific or problem-oriented axioms, because they express the specific knowledge about particular problems. Such are, for instance, the axioms of double integration:

\[
\vdash X \land Y \rightarrow Z; \quad \vdash (X \rightarrow Z) \rightarrow (B \rightarrow W); \quad \vdash (Y \rightarrow W) \rightarrow (A \rightarrow S).
\]  (5)

The derivation of a formula \(X\) is a tree of sequents with the sequent \(X\) in its root. Each sequent in the tree is either an axiom, being a leaf of the tree, or a consequence of the nodes immediately above it according to one of the following three rules which we call structural synthesis rules (SSR):

\[
\frac{\bigwedge_{1 \leq i \leq n} (\mathcal{A}_i \rightarrow B^i) \rightarrow (C_m \rightarrow V); \Gamma_i, \mathcal{A}_i \vdash B^i (i = 1, 2, \ldots, n); \Sigma_i \vdash C_j (j = 1, 2, \ldots, m)}{\Gamma_1, \ldots, \Gamma_n, \Sigma_1, \ldots, \Sigma_m \vdash V}.
\]  \((\rightarrow - - )\)

\[
\frac{\vdash \mathcal{A}_k \rightarrow V; \Sigma_i \vdash A_i (i = 1, 2, \ldots, k)}{\Sigma_1, \ldots, \Sigma_k \vdash V}.
\]  \((\rightarrow - )\)

\[
\frac{A_1, \ldots, A_k \vdash B}{\mathcal{A}_k \rightarrow B}.
\]  \((\rightarrow + )\)

In the rule \((\rightarrow - - )\) \(\mathcal{A}_i\) is a sequence of propositional variables from \(\mathcal{A}_i\). Not necessarily all variables from \(\mathcal{A}_i\) must be present in \(\mathcal{A}_i\). That means that \(B^i\) may be computable from less variables than it is assumed in the conditional computability statement.

In Fig. 3 the derivation of the goal \(A \land B \rightarrow S\) is shown, given the double-integration axioms (5) and four logical axioms: \(A \vdash A\), \(B \vdash B\), \(X \vdash X\), and \(Y \vdash Y\).

The three rules presented above are the only ones used for proof search in the PRIZ system. In fact the derivations are constructed not in the tree form but as formula sequences with block structure (Fitch-style derivations \([5]\)). This allows to avoid duplication of subderivations.

One can immediately see that the rules \((\rightarrow - - ), (\rightarrow - )\) and \((\rightarrow + )\) may be obtained from the familiar natural deduction rules \([9]\) for \(\rightarrow\) and \(\land\). So the SSR constitute a subsystem of the intuitionistic propositional calculus. Later we shall see that the SSR are equivalent to it.
4. Program extraction

Now we show how to extract a program in PL from the SSR derivation of a sequent in LL. We shall assign terms of the PL language to the sequents of the SSR derivation, beginning from the axioms and proceeding along the applications of the rules. This assignment uses a known device traceable to the Heyting-Kolmogorov interpretation of intuitionistic connectives, or more precisely the Kleene realizability [10].

We extend our logical language LL with terms of PL which represent programs. Instead of propositional variables of LL, we shall use monadic predicates which are in one-to-one correspondence with the propositional variables of LL. So we obtain a new logical language LL1.

(1) A formula $A(t)$, where $t$ is a term, corresponds in LL1 to a propositional variable $A$ of the LL. $A(t)$ expresses the fact that $t$ is the right value of the variable $a$ in the problem at hand.

(2) The formula

$$\forall x_1 \cdots \forall x_k (A_1(x_1) \land \cdots \land A_k(x_k) \rightarrow B(f(x_1, \ldots, x_k)))$$

abbreviated to

$$A_k \rightarrow_B B$$
corresponds in LL1 to the unconditional computability statement $A_k \rightarrow B$ in LL. It expresses not only the computability of $b$, but also the means for computing it, i.e. the function $f$.

(3) The formula

$$\bigwedge_{1 \leq i \leq n} (A^i_k \rightarrow B^i) \rightarrow (C_m \rightarrow D)$$

(8)

corresponds in LL1 to the conditional computability statement

$$\bigwedge_{1 \leq i \leq n} (A^i_k \rightarrow B^i) \rightarrow (C_m \rightarrow D).$$

It expresses the computability of $d$ from $c_1, \ldots, c_m$ on the condition of the computability of $b^i$ from $a^i_1, \ldots, a^i_k$ ($i = 1, 2, \ldots, n$) by means of a new function built from $\varphi^1, \ldots, \varphi^n$ by $F$.

Having a function $\text{imps}(f, u) = \int_0^1 f \, dn$ for integration, we can express in LL1 the problem-oriented axioms for double integration:

$$\vdash X \land Y \rightarrow Z,$$

$$\vdash (X \rightarrow Z) \rightarrow (B \rightarrow W),$$

$$\vdash (Y \rightarrow W) \rightarrow (A \rightarrow S).$$

Due to the one-to-one correspondence between the formulas of LL and LL1, we immediately can conclude that the structural synthesis rules are applicable in LL1 after minor syntactic amendments. These amendments concern the construction of new terms included into derived sequents. The terms appear as follows:

(1) Terms in axioms are given. This is obvious for specific axioms, and it is true for logical axioms because they have the form $A(t) \vdash A(t)$, where $A$ is a monadic predicate and $t$ is either a variable or a constant of primitive type.

(2) The application of rule ($\rightarrow -$) gives a sequent

$$\Sigma_1, \Sigma_2, \ldots, \Sigma_k \vdash V(f(a_1, a_2, \ldots, a_k)),$$

where $f$ is taken from the premise $\Lambda_k \vdash V$ and each $a_i$ is taken from $\Sigma_i \vdash A_i(a_i)$, $i = 1, 2, \ldots, k$.

(3) The application of rule ($\rightarrow -$) gives a sequent

$$\Gamma_1, \Gamma_2, \ldots, \Gamma_n, \Sigma_1, \Sigma_2, \ldots, \Sigma_m \vdash V(\lambda a^1_1 \ldots a^1_k \ldots a^n_{i_1} \ldots a^n_{i_l} b^1_1 \ldots (a^1_{i_1} \ldots a^1_k \ldots a^n_{i_1} \ldots a^n_{i_l} b^1_1) \ldots (c_1, \ldots, c_m)),$$

where $a^i_s$ are individual variables from $A^i_s(a^i_s)$, each $b^i$ is taken from $A^i_s(B^i(b^i))$, and each $c_j$ is taken from $\Sigma_j \vdash C_j(c_j)$.

(4) The rule ($\rightarrow +$) yields a sequent

$$\vdash A_k \rightarrow B$$

where $b$ comes from $B(b)$ and each $a_i$ comes from $A_i(a_i)$.
We finish the example of double integration by extracting a program from the derivation of \( \vdash A \land B \rightarrow S \). We assign terms (free variables \( x, y, a, b \)) to the logical axioms: \( X(x) \vdash X(x) \), \( Y(y) \vdash Y(y) \), \( A(a) \vdash A(a) \), \( B(b) \vdash B(b) \). The terms of the problem-oriented axioms of this example have been constructed already in the beginning of this section.

The derivation of \( \vdash A \land B \rightarrow S \) together with the extraction of a program in LL1 is as follows:

\[
\begin{align*}
\vdash X & \land Y \rightarrow Z ; X(x) \vdash X(x) ; Y(y) \vdash Y(y) \\
X, Y & \vdash Z(g(x, y)) \\
\end{align*}
\]

\[
\begin{align*}
\vdash (X & \rightarrow Z) \rightarrow (B \xrightarrow{\text{simps}(\phi)} W) ; \\
Y, B & \vdash W(\text{simps}(\lambda x g(x, y)(b))) \\
\end{align*}
\]

\[
\begin{align*}
\vdash (Y & \rightarrow W) \rightarrow (A \xrightarrow{\text{simps}(\phi)} S) ; \\
A, B & \vdash S(\text{simps}(\lambda y \text{simps}(\lambda x g(x, y)(b)(a)))) \\
\end{align*}
\]

\[
\begin{align*}
\vdash A \land B & \xrightarrow{\lambda a b \text{simps}(\lambda y \text{simps}(\lambda x g(x, y)(b)(a)))} S
\end{align*}
\]

**Remark 1.** Using the constants with a preprogrammed meaning (see Section 3):

- \textit{recu} for primitive recursion,
- \textit{min} for unbounded minimization,
- \textit{0} for zero function,
- \textit{next} for successor function

we can encode in LL1 any recursive function as soon as we know its partial recursive description, i.e. its representation through primitive-recursion and minimization operators. This shows a trivial way for the automatic construction of any recursive function from its axiomatic representation in LL1, provided only four operations \textit{recu}, \textit{min}, \textit{0} and \textit{next} are preprogrammed.

5. **Completeness of the structural synthesis rules**

It may be surprising that the language LL with quite simple implicative formulas allows to express computability and is equivalent to the whole propositional calculus. Besides that, the formulas of LL are simple enough to allow an efficient search of proofs needed for the automatic program synthesis.
The completeness of the structural synthesis rules will be proved by
(1) reducing arbitrary propositional formulas to LL-form,
(2) showing that the SSR rules are sufficient for intuitionistic proofs of the latter.

Intuitionistic logic does not in general allow the transformation of formulas into
disjunctive or conjunctive normal form, but it still allows simplification by introduction
of new variables. We use this to reduce arbitrary propositional formulas to
the equivalent LL-form. We denote by \( F_x[G] \) (or \( F[G] \)) the result of substituting
\( G \) for \( X \) in \( F \). For any sequent \( S \) and \( T \), the notation \( S \equiv_{\text{ded}} T \) means that \( S \) and
\( T \) are deductively equivalent, i.e., the derivability of \( S \) implies the derivability of
\( T \) and vice versa.

**Lemma 1.**

(a) \( \Gamma \vdash F[E] \equiv_{\text{ded}} (X \leftrightarrow E), \Gamma \vdash F[X] \).

(\&) \( \Gamma \vdash F[G \land H] \equiv_{\text{ded}} (X \rightarrow G), (X \rightarrow H), (G \rightarrow (H \rightarrow X)), \Gamma \vdash F[X] \).

(\rightarrow) \( \Gamma \vdash F[G \rightarrow H] \equiv_{\text{ded}} (X \rightarrow (G \rightarrow H)), ((G \rightarrow H) \rightarrow X), \Gamma \vdash F[X] \).

(\lor) \( F[G \lor H] \equiv_{\text{ded}} (X \rightarrow (G \lor H)), (G \rightarrow X), (H \rightarrow X), \Gamma \vdash F[X] \).

(\leftrightarrow) \( F[G \leftrightarrow H] \equiv_{\text{ded}} (X \rightarrow (G \rightarrow H)), (X \rightarrow (H \rightarrow G)),
((G \rightarrow H) \rightarrow ((H \rightarrow G) \rightarrow X)), \Gamma \vdash F[X] \),

where \( E \) is any formula in \( F \) and \( X \) is a new variable occurring only to the right of \( \equiv_{\text{ded}} \).

**Proof.** (a) The left-to-right implication follows from the equivalence replacement
theorem \( (X \leftrightarrow E) \rightarrow (F[X] \leftrightarrow F[E]) \).

The inverse implication follows from the substitution of \( E \) for \( X \) on the right-hand
side.

The remaining clauses of the lemma are obtained from clause (a) after the
replacement of \( E \) by \( G \land H, G \rightarrow H, \) etc., using the fact that the formula \( X \leftrightarrow E \)
becomes equivalent to the conjunction of formulas to the right of \( \equiv_{\text{ded}} \). \( \square \)

The following lemma is suggested by the second-order equivalences
\[
(p \lor q) \leftrightarrow \forall x ((p \rightarrow x) \rightarrow ((q \rightarrow x) \rightarrow x))
\]
and
\[
\bot \leftrightarrow \forall x x,
\]
where the constant \( \bot \) denotes falsity.

**Lemma 2.**

(a) \( F[\bot] \equiv_{\text{ded}} L \leftrightarrow (K \land Z) \vdash F[L] \)
where \( K \) is the conjunction of all propositional variables on the left-hand side, and \( L, Z \) are new variables.

(b) \[ \Gamma, (X \rightarrow B \lor C) \vdash F \quad \text{ded} \quad \Gamma, (X \rightarrow K) \vdash F, \]

where \( K \) is the conjunction
\[ \bigwedge_Z (B \rightarrow Z) \rightarrow ((C \rightarrow Z) \rightarrow Z) \]
over all variables \( Z \) on the left-hand side, provided \( L \) does not occur there and \( \lor \) occurs only in the formulas of the form \( W \rightarrow U \lor V, U, V, W \) being variables, to the left of \( \vdash \).

The proof of Lemma 2 is presented in the Appendix, where the normal form theorem is considered.

**Remark 2.** The transformations described in Lemmas 1 and 2 expand the size of sequents not more than in square. If the transformations 2(b) are not applied, then the increase is no more than linear and the general structure of the derivation is preserved.

**Theorem 1.** Any propositional formula \( A \) is deductively equivalent to a sequent \( A_1, A_2, \ldots, A_k \vdash V \), where \( V \) is a variable and each \( A_i \) is a variable or a formula of the LL.

**Proof.** Using Lemmas 1 and 2. \( \square \)

**Theorem 2.** Any intuitionistically derivable formula of LL is derivable by the structural synthesis rules.

A proof of the theorem, using a long normal form theorem, is presented in the Appendix.

Now we have established the completeness of the structural synthesis theories. The LL language enables us to encode any intuitionistic propositional formula, and the set of SSR rules is sufficient for deriving any intuitionistically derivable LL formula.

6. **Structural synthesis and data flow**

The fact that any propositional formula can be encoded as a set of formulas of LL, together with the intuitionistic completeness of SSR, shows that the structural synthesis of programs is P-SPACE complete, because provability in the intuitionistic propositional calculus is P-SPACE complete [24]. Nevertheless, a synthesizer exists
in which the structural synthesis of programs is used rather efficiently for solving many practical problems. But, indeed, a small number of theorems from those we proved in [27] were significantly harder to prove than we had expected from our experience with PRIZ applications. These theorems contain some combination of negations and disjunctions.

Here we give some hints for the efficient implementation of the structural synthesis. Let us consider a set of computability statements $\Gamma$ describing problem conditions, and a formula which must be proved for solving the problem “compute $v$ from $u$ when $\Gamma$”. A proper data structure for representing $\Gamma$ is a network. Every propositional variable and every computability statement is represented as a node in the network. The node of any computability statement is connected with the nodes of the propositional variables which occur in this formula. The computability statements are connected with each other in the network through the common propositional variables. A position of the propositional variable in the formula is represented by a labelling $(in, out, arg1, res1, arg2, res2, \ldots)$ on the edges. Figure 4 shows the network for the double integration problem.

![Fig. 4. Data flow schema.](image)

Having the network representation of the problem conditions, it is possible to transform this network into a data-flow schema for any program which can be synthesized from these conditions.

For this purpose we determine a direction for every edge in the network by the following rule: the arrows lead from negative occurrences of propositional variables to positive occurrences of propositional variables. As usually in logic, we say that an occurrence of a subformula is negative when it is on the left side of an odd number of implications in the formula. Otherwise the occurrence is positive. The directions have already been determined in Fig. 4, and the network can be considered as a data-flow schema for the integration program derived in Section 4.
The rule for data flow directions is suggested by the SSR rules. A negative occurrence of a subformula $A \rightarrow B$ in an axiom determines a description of a function, and its positive occurrence introduces a call of the function in the final program. ‘$A$’ corresponds to the input and ‘$B$’ to the output of the function of the implication $A \rightarrow B$. This gives the directions for the edges connecting $A$ and $B$.

The data-flow schema is very useful for building a proof of the solvability theorem. In the case where there are only unconditional computability statements in $\Gamma$, the search becomes a simple flow analysis on a graph. (It has recently been shown by Dikovski that this search can be done in linear time.) If $\Gamma$ contains conditional computability statements, the search is done on an and–or-tree of subproblems. Subproblems are generated for negative occurrences of subformulas $A \rightarrow B$ in the computability statements. No pattern matching is needed, because the data-flow schema explicitly represents all possible connections between the formulas. No unification is applied. Nevertheless, the concretization of variables is not excluded from the structural synthesis of programs. Let us recall that a problem specification is initially written in a problem-oriented language. Concretization is done when a problem specification given in the problem-oriented language is unfolded into a problem specification in LL. In our first example the axioms for a bar are transformed into the axioms for the bars $AB$ and $BC$. This concretization is always done with linear time complexity because one pass along the text suffices.

7. Recursion

The structural synthesis rules described above allow to synthesize applicative programs. A variant of the PRIZ system permits to synthesize recursive programs from axioms given in the language LL. In order to formulate the corresponding extension of the derivation rules, we extend the notion of a sequent by allowing expressions on the form $[A]$ for formulas $A$ to occur to the left of the turnstile. We add a rule for recursion:

\[ \Gamma; [A_k \rightarrow X], A_1, \ldots, A_k \vdash X \]
\[ \Gamma; A_k \vdash X \] (Rec)

This rule can be obtained from the usual transfinite induction rule by suppressing individual variables. (Recall the suppressing of variables when passing from LL1 to LL.)

The rule $(\rightarrow - -)$ is extended as follows:

\[ \bigwedge_{1 \leq i \leq n} (A_{k_i} \rightarrow B_i) \rightarrow (C_m \rightarrow V); \Gamma_0, \bar{A}^i \vdash B^i (i = 1, \ldots, l); \Sigma_j \vdash C_j (j = 1, \ldots, m) \]
\[ [A_{k_{i+1}} \rightarrow B^{i+1}], \ldots, [A_{k_n} \rightarrow B^n], \Gamma_1, \ldots, \Gamma_n, \Sigma_1, \ldots, \Sigma_m \vdash V \] (\rightarrow - -)
More precisely, when applying this rule, one should partition the set \( \{1, \ldots, n\} \) into two sets \( U, V \), and write \( I, \bar{A} \vdash B^i \) \( (i \in U) \) above the line and \( [A_k \rightarrow B^j] \) \( (j \in V) \), \( I_i \) \( (i \in U) \), \( \Sigma_1, \ldots, \Sigma_m \vdash V \) under the line.

**Example.** Derivation of \( N \rightarrow F \) from an axiom \( (N \rightarrow F) \vdash (N \rightarrow F) \) according to the extended set of rules:

\[
\begin{align*}
\frac{\vdash (N \rightarrow F) \rightarrow (N \rightarrow F); \quad N \vdash N}{[N \rightarrow F], N \vdash F} & \quad (\rightarrow - -) \\
& \quad \text{(Rec)} \\
N \vdash F & \quad (9) \\
\vdash N \rightarrow F
\end{align*}
\]

The language PL is extended by introducing a new rule for constructing terms. It uses a recursion functional \( \rho \) which binds variables syntactically, much in the same way as the \( \lambda \)-symbol does.

If \( t \) is a term of type \( \tau \), \( \bar{a} \) is a sequence of variables of types \( \bar{\sigma} \) (i.e. \( \bar{a} = a_1^{\sigma_1}, \ldots, a_n^{\sigma_n} \); \( \bar{\sigma} = \sigma_1, \ldots, \sigma_n \)), and \( f \) is a variable of type \( (\bar{\sigma} : \tau) \), then \( \rho \bar{a} t \) is again of type \( (\bar{\sigma} : \tau) \).

The operational semantics for the recursion functional \( \rho \) is given by the equation:

\[
g(\bar{a}) = t(g, \bar{a}),
\]

where \( g \) stands for \( \rho \bar{a} t \) and \( t \) is presented as \( t(f, \bar{a}) \).

The program-extraction algorithm given in Section 4 is extended as follows. If \( t(f, \bar{a}) \) is a term assigned to the premise of \( \text{(Rec)} \), then \( \rho \bar{a} t \) is assigned to the conclusion. Here \( f \) is the variable assigned to \( [A_k \rightarrow X] \), and \( \bar{a} \) is the sequence of variables assigned to \( A_k \) to the left of the turnstile. Considering the new form of the \( (\rightarrow - -) \) rule, we remind the reader that \( F(\varphi^1, \ldots, \varphi^n) \) is assigned in (8) to the axiom which is the left premise of this rule; \( \varphi^i \) are variables assigned to conjunctive members \( A_k \rightarrow B^i \). Now we assign the term \( F(\lambda \bar{a}^1 b^1, \ldots, \lambda \bar{a}^l b^l, \varphi^{l+1}, \ldots, \varphi^n)(c_1, \ldots, c_m) \) to the conclusion of the new \( (\rightarrow - -) \) rule, where \( \bar{a}^i, b^i, c_j \) have the same meaning as in Section 4.

**Example continued.** Let the LL1-axiom corresponding to the LL-axiom

\[
(N \rightarrow F) \rightarrow (N \rightarrow F)
\]

be

\[
(N \rightarrow F) \rightarrow (N \rightarrow F)
\]

where \( G \) is a preprogrammed constant with the properties

\[
n = 0 \rightarrow G(\varphi)(n) = 1, \quad n \neq 0 \rightarrow G(\varphi)(n) = n \times G(\varphi)(n - 1).
\]

Then the program extracted from the derivation (9) is the familiar recursion for the factorial function.
In fact the PRIZ program synthesizer can handle recursion synthesis, but then it uses a restriction on the SSR rules. The restriction is to avoid the use of Fitch's reiteration rule [5].

Let us consider the question of program correctness. The applicative procedures involved in program extraction according to SSR rules obviously preserve the property of being total, i.e. they terminate for all values of arguments. The recursion scheme (10) does not have this property: take, for example, \( t(g, a) = 1 + g(a + 1) \).

The most common way to ensure termination is to fix a well-ordering relation \(<\) (i.e. a binary relation without infinite descending chains \(x_0 > x_1 > x_2 > \cdots\)) with the least element 0, and to require

\[
t(g, a) = S((\lambda x < a)g, a)
\]

where

\[
((\lambda x < a)g)(x) = \begin{cases} g(x) & \text{if } x < a, \\ 0 & \text{otherwise} \end{cases}
\]

Then the termination of \( pf\) can be proved by transfinite induction on \(<\). The term \( t \) for the factorial has the properties required by this scheme; but, to fit another favourite example such as the Ackermann function

\[
f(x, y) = \text{if } x \cdot y = 0 \text{ then } f_0(x, y) \text{ else } f(x - 1, f(x, y - 1)) \text{ fi,}
\]

one has either to use a complicated ordering of the order-type \( \omega^\omega \) or to increase type levels by first defining \( F(x) = \lambda yF(x, y) \) using primitive recursion on \( x \).

These examples show us that in the case of the structural synthesis of recursive programs the proof of total correctness must be done by means not available in the logical language used for automatic deduction. We believe that for meaningful problem specifications the termination can be recognized immediately since the terms assigned to the axioms can be transformed into the form required by (11).

8. Example

M.C. Gaudel has considered the problem of compiler construction as a problem of implementation of abstract data types. In a related approach pursued by J. Penjam [20, 21], the PRIZ system has been successfully used for the automatic synthesis of semantic processors of some languages. We shall outline this approach as an example of application of structural synthesis.

Let \( G \) be an attribute grammar [11] with a set \( P \) of production rules and a set \( X \) of attributes. We attach a set of computability statements to every production \( p \) from \( P \), and call this set an attribute model of \( p \). There is a unique way for deriving attribute models from the usual representation of an attribute grammar
$G$ [20], so that for any correct $G$ the attribute models correctly represent the language given by $G$, in the following sense.

The attribute models can be transformed into a set of axioms which allow to prove the computability of the attributes of the initial symbol $s_0$ of the grammar, assuming that a parsing tree $t$ of a text is given. In this way we can construct a semantic processor for the language. It takes a parsing tree of a text as an input, and outputs the values of the attributes of $s_0$ which represent the meaning of the text.

For instance, let us consider the following grammar:

Terminal symbols: $res, par, ser, 1, 0$.
Nonterminal symbols: schema, number.

Productions:
- $pl : schema \rightarrow res(number)$
- $p2 : schema \rightarrow par(schema, schema)$
- $p3 : schema \rightarrow ser(schema, schema)$
- $p4 : number \rightarrow 1$
- $p5 : number \rightarrow number 1$
- $p6 : number \rightarrow number 0$

The initial symbol is schema.

Let us use this language for describing electrical circuits, so that $par(s_1, s_2)$ stands for the parallel connection of $s_1$ and $s_2$, $ser(s_1, s_2)$ stands for the series connection of $s_1$ and $s_2$, and $res(N)$ is a resistor with the resistance given as a binary number $N$. Let the meaning of any expression of the language be the resistance of a circuit described by the expression. For instance, the meaning of

$par(ser(res(1), res(1)), res(10))$

is the resistance of the circuit in Fig. 5., and is equal to 1.

Fig. 5. A circuit.

We express the attribute models of the productions in the specification language which has been used in the example of Section 2. Let us introduce an attribute $r$ for expressing the resistance of the parallel or series connection of two circuits which have the resistances represented by attributes $r1$ and $r2$ respectively. The
attribute models of the productions \(p_2, p_3\) (which express parallel and series connection) are as follows:

\[
\begin{align*}
\text{modp2: } & (r, r_1, r_2: \text{real}; \\
& \quad \text{rel } r = r_1 \times r_2 / (r_1 + r_2)); \\
\text{modp3: } & (r, r_1, r_2: \text{real}; \\
& \quad \text{rel } r = r_1 + r_2).
\end{align*}
\]

The attribute models for building binary numbers contain the attributes \(\text{newnr}\) and \(\text{oldnr}\), respectively associated with the symbol number on the left- and right-hand side of productions \(p_4, p_5, p_6\).

\[
\begin{align*}
\text{modp4: } & (\text{newnr: real}; \\
& \quad \text{rel } \text{newnr} = 1); \\
\text{modp5: } & (\text{newnr, oldnr: real}; \\
& \quad \text{rel } \text{newnr} = 2 \times \text{oldnr} + 1); \\
\text{modp6: } & (\text{newnr, oldnr: real}; \\
& \quad \text{rel } \text{newnr} = 2 \times \text{oldnr}).
\end{align*}
\]

In the axioms for the grammar we use propositional variables \(NR, PR, SR\) and \(SCR\) which express the computability of the resistance of a resistor, a parallel connection, a series connection and a schema. The propositional variables \(NR_1, NR_2\) and \(NR_3\) represent the computability of numbers, and \(T\) represents the computability of a parsing tree of a text.

The axioms representing the semantics are as follows:

\[
\begin{align*}
A1 & \quad (T \xrightarrow{\varphi_1} \text{NR}) \land (T \xrightarrow{\varphi_2} \text{PR}) \land (T \xrightarrow{\varphi_3} \text{SR}) \rightarrow (T \xrightarrow{\varphi_{F(\varphi_1,\varphi_2,\varphi_3)}} \text{SCR}), \\
A2 & \quad (T \xrightarrow{\varphi} \text{SCR}) \rightarrow (T \xrightarrow{\varphi_{F_1(\varphi)}} \text{PR}), \\
A3 & \quad (T \xrightarrow{\varphi} \text{SCR}) \rightarrow (T \xrightarrow{\varphi_{F_2(\varphi)}} \text{SR}), \\
A4 & \quad (T \xrightarrow{\varphi_1} \text{NR}_1) \land (T \xrightarrow{\varphi_2} \text{NR}_2) \land (T \xrightarrow{\varphi_3} \text{NR}_3) \rightarrow (T \xrightarrow{\varphi_{G(\varphi_1,\varphi_2,\varphi_3)}} \text{NR}), \\
A5 & \quad T \rightarrow \text{NR}_1, \\
A6 & \quad (T \xrightarrow{\varphi} \text{NR}) \rightarrow (T \xrightarrow{\varphi_{F_1(\varphi)}} \text{NR}_2), \\
A7 & \quad (T \xrightarrow{\varphi} \text{NR}) \rightarrow (T \xrightarrow{\varphi_{F_2(\varphi)}} \text{NR}_3).
\end{align*}
\]

\(F, G, F_1, F_2, F_3, F_4\) and \(f\) are preprogrammed constants with the following properties:

\[
F(t) = \begin{cases} 
p_1 & \text{if } \text{prod}(t) = p_1 \text{ then } \varphi_1(t) \\
& \text{elif } \text{prod}(t) = p_2 \text{ then } \varphi_2(t) \\
& \text{else } \varphi_3(t) \end{cases}
\]

$G(t) = \begin{cases} 
\text{if } \text{prod}(t) = p_4 & \varphi_1(t) \\
\text{elif } \text{prod}(t) = p_5 & \varphi_2(t) \\
\text{else } \varphi_3(t) \end{cases}$

$F_1(t) = \varphi(\text{left}(t)) * \varphi(\text{right}(t))/(\varphi(\text{left}(t)) + \varphi(\text{right}(t)))$,

$F_2(t) = \varphi(\text{left}(t)) + \varphi(\text{right}(t))$,

$F_3(t) = 2 * \varphi(\text{left}(t)) + 1$,

$F_4(t) = 2 * \varphi(\text{left}(t))$,

$f(t) = 1$.

where \(\text{right}(t)\) is the right subtree of \(t\), \(\text{left}(t)\) is the left subtree of \(t\), and \(\text{prod}(t)\) is the production used in the root of \(t\).

Our goal is to compute the resistance of the schema, having its parsing tree; therefore we have to prove \(T \rightarrow \text{SCR}\).

The proof is represented in Fig. 6.

Finally, this proof gives us the following recursive definitions of functions \(F\) and \(G\) such that \(T \rightarrow \text{SCR}\) and \(T \rightarrow \text{NR}\): 

\[
F(t) = \begin{cases} 
\text{if } \text{prod}(t) = p_1 & G(\text{left}(t)) \\
\text{elif } \text{prod}(t) = p_2 & 
F(\text{right}(t)) * F(\text{left}(t))/(F(\text{right}(t)) + F(\text{left}(t))) \\
\text{else } F(\text{right}(t)) + F(\text{left}(t)) \end{cases}
\]
\[ G(t) = \begin{cases} 1 & \text{if } \text{prod}(t) = p_4 \\ \text{elif } \text{prod}(t) = p_5 & 2 \times G(\text{left}(t)) + 1 \\ \text{else} & 2 \times G(\text{left}(t)) \end{cases} \]

9. Concluding remarks

We believe that we have depicted a mathematically sound and practically efficient way of going from specifications to programs. Its efficiency has been demonstrated in practice by use of the PRIZ system.

We can regard the structural synthesis of programs also as a technique for the automatic implementation of abstract data types which have axiomatic descriptions in LL1. The concepts point, bar, and the description of a mechanism in Section 2 are abstract data types. But abstract data types are being handled in our case in an essentially different way compared to most of theoretical papers. We use in some sense a minimal description of a data type. Unconditional and conditional computability statements are the only axioms used in our specifications of data types. They do not contain any information about the properties of predefined (primitive) functions except the information about the applicability of the functions for computing some objects.

We were in trouble when we tried to present an example in every detail here, because the unfolding of source specifications gave us a great number of axioms. Already for the problem in Section 2 the number of axioms after unfolding was 50; yet, the problem description in the source language took 4 lines, and a synthesizer which works with unconditional computability statements on an Apple-II microcomputer solves the problem in a few seconds. The number of axioms in typical applications of the PRIZ system is measured not only in hundreds but also in thousands. This makes searching proofs by hand completely impossible. In the example of Section 7 we could not consider any real-life language for the same reasons.

Acknowledgments

We would like to thank our colleagues and especially J. Penjam for discussing an example and testing it on the computer. We thank the referees for many useful remarks.

Appendix. Normal form of derivations

Natural deduction. Propositional formulas are constructed from propositional variables and a constant \( \bot \) (falsity) by means of \( \rightarrow, \land, \lor \). Negation \( \neg \) and equivalence
are defined by
\[ (A \leftrightarrow B) \equiv ((A \rightarrow B) \land (B \rightarrow A)). \]
The derivable objects are sequents \( \Gamma \vdash A \). The axioms have the form \( X \vdash X \). There are 7 inference rules; one rule for \( \bot \) and two rules (introduction and elimination, marked + and −, respectively) for each connective:
\[
\begin{align*}
\frac{\Gamma \vdash \bot}{\Gamma \vdash A} & (\bot); & \frac{\Gamma \vdash A_1 \land A_2}{\Gamma \vdash A_i} (\land^{-}), \ i = 1, 2; & \frac{\Gamma \vdash A; \Sigma \vdash B}{\Gamma, \Sigma \vdash (A \land B)} (\land^{+}); \\
\frac{\Gamma \vdash A \rightarrow B; \Sigma \vdash A}{\Gamma, \Sigma \vdash B} (\rightarrow^{-}); & \frac{\Gamma, A \vdash B}{\Gamma \vdash (A \rightarrow B)} (\rightarrow^{+}); \\
\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \lor A_2} (\lor^{+}), \ i = 1, 2; & \frac{\Gamma \vdash A \lor B; A, \Sigma \vdash C; B, \Pi \vdash C}{\Gamma, \Sigma, \Pi \vdash C} (\lor^{-}).
\end{align*}
\]
The sequents over the horizontal line are premises, the sequent under the line is a conclusion.

A deduction is a tree of sequents constructed according to the rules and having axioms as leaves.

A deduction is normal if the leftmost premise of any elimination rule either is an axiom or is obtained by \((\rightarrow^{-})\) or \((\land^{-})\). The following proposition is known from proof theory [3].

**Normal form theorem.** (a) Any deduction can be transformed into the normal deduction of the same endsequent.

(b) A normal deduction contains only subformulas of its endsequent. The rules are applied only for the connectives (and constant \( \bot \)) occurring in the endsequent and they follow the signs of occurrences: introduction rules for positive ones, and elimination rules for negative ones.

A simple additional transformation yields the following extension.

**Long normal form theorem.** Any deduction can be transformed into a normal deduction of the same endsequent where the conclusion of any elimination rule having a form \((A \rightarrow B)\) or \((A \land B)\) is itself a left-most premise of an elimination rule.

**Proof.** Replace in the following way the topmost sequents violating the additional requirements:
\[
\begin{align*}
\frac{\Gamma \vdash A \rightarrow B; A \vdash A}{\Gamma, A \vdash B}, & \quad \frac{\Gamma \vdash A \land B \quad \Gamma \vdash A \land B}{\Gamma \vdash A \land B}. \quad \Box
\end{align*}
\]

**Proof of Lemma 2.** (a) If a derivation of \( F[\bot] \) is given, assume it contains only variables from \( F[\bot] \) (otherwise replace them by \( \bot \)). Now replace \( \bot \) by \( K \land Z \).
All applications of the rules, except the rule
\[ \frac{\Gamma \vdash \bot}{\Gamma \vdash A} \quad \text{for } \bot, \]
remain valid. Applications of the rule for \( \bot \) can be restored using the implication
\( K \land Z \rightarrow G \), derivable for any formula \( G \) which contain only variables from \( K \). This
gives us the derivation of the sequent \( F[K \land Z] \). From the Lemma 1 we obtain the
derivation of \( L \leftrightarrow (K \land Z) \vdash F[L] \).

\( F[\bot] \) can be obtained from \( L \leftrightarrow (K \land Z) \vdash F[L] \) by substituting \( \bot \) for \( Z, L \), and
using the derivability of the equivalence \( \bot \leftrightarrow (K \land \bot) \).

(b) Let us note that, from the normal form theorem, for any derivable sequent \( \Gamma, (X \rightarrow B \lor C) \vdash F \) satisfying the assumptions of the lemma being proved here, a
derivation exists where any formula derived by means of the rule (\( \lor^- \)) is a variable,
and where the rule (\( \lor^+ \)) is not applied at all. We can change all occurrences of
\( B \lor C \) in such a proof to \( K \) defined above. Only applications of the rule (\( \lor^- \)) with
the left assumption \( \Gamma \vdash B \lor C \) may become invalid in this case. Actually they take
the form
\[ \frac{\Gamma'' \vdash K, \Sigma', B \vdash Z; \Pi', C \vdash Z}{\Gamma'', \Sigma', \Pi' \vdash Z} \]
which can be easily completed into the requested derivation using the monotone
replacement theorem. The transition from \( \Gamma, (X \rightarrow K) \vdash F \) to \( \Gamma, (X \rightarrow B \lor C) \vdash F \)
is done using the same theorem. □

**Proof of Theorem 2.** Consider a deduction in the long normal form of a sequent
\[ C_1, \ldots, C_n \vdash (A_1, \ldots, A_k \rightarrow A) \quad (1) \]
where \( C_i \) are computability statements or variables, and \( A_n, A \) are variables. By
the normal form theorem the only rules applied are (\( \rightarrow^- \)), (\( \land^- \)) and (\( \rightarrow^+ \)).

It is sufficient to prove the theorem for \( k = 0 \) since we can replace (1) by
\( C_1, \ldots, C_n, A_1, \ldots, A_k \vdash A \). Note that the only antecedent members of the
sequent in the deduction are \( C_i \) and variables, since otherwise the subformula
property would be violated. Consider now some uppermost application of an
elimination rule in the deduction. Its left premise is an axiom \( X \vdash X \), and \( X \) should
be one of \( C_i \). If this is an unconditional computability statement \( B_1, \ldots, B_\gamma \rightarrow B \),
then (in view of long normality) the part of the deduction down the considered
rule is of the form
\[ C_i \vdash B_1 \rightarrow (\cdots \rightarrow (B_\gamma \rightarrow B) \cdots); \Gamma_1 \vdash B_1 \]
\[ C_i, \Gamma_1 \vdash B_2 \rightarrow (\cdots \rightarrow (B_\gamma \rightarrow B) \cdots) \]
\[ \vdots \]
\[ C_i, \Gamma_1, \ldots, \Gamma_{\gamma-1} \vdash B_\gamma \rightarrow B; \Gamma_\gamma \vdash B_\gamma \]
\[ C_i, \Gamma_1, \ldots, \Gamma_\gamma \vdash B \]

(2)
and, after deletion of $C_i$ from the antecedent, it is easily transformed into the SSR-form of $(\rightarrow \neg)$.

If $C_i$ is a conditional computability statement $\land_i (A^i \rightarrow B^i) \rightarrow (A \rightarrow B)$, then the part of the deduction we are interested in is of the form

$$C_i \vdash \land_i (A^i \rightarrow B^i) \rightarrow (A \rightarrow B); \Gamma \vdash \land_i (A^i \rightarrow B^i)$$

$$\frac{C_i, \Gamma \vdash A \rightarrow B; \Sigma \vdash A}{C_i, \Gamma, \Sigma \vdash B}$$  \hspace{1cm} (3)

where two lower levels have the same form as (2). Replacing $\land_i (A^i \rightarrow B^i)$ by $A^i \rightarrow B^i$ for each $i$ in turn, we obtain the derivations of $\Gamma \vdash A^i \rightarrow B^i$. Moving $A^i$ into the antecedent, we obtain the derivations of $\Gamma, A^i \vdash B^i$. So (3) can be turned into $(\rightarrow \neg \neg)$. Since the applications of introduction rules are possible only inside figures of the form (3), we have established our theorem. $\square$

References


