THE USE OF METASYSTEM TRANSITION IN THEOREM PROVING AND PROGRAM OPTIMIZATION

Valentin F. Turchin, The City College, The City University of New York

Compare proving a theorem in an axiomatic system with the computation process when we are dealing with recursive function definitions. The former is nondeterministic and requires either an exhaustive search or an heuristic technique to set subgoals which are likely to lead to the desired end. The latter is deterministic and straightforward. Obviously, we should try to substitute the proof by computation for the proof by constructing a demonstration if our aim is to facilitate computerization. The purpose of the present paper is to introduce a technique which, as we believe, crucially increases the power of the proof by computation.

1. Metasystem transition in formal arithmetic

We shall illustrate our idea by examples from formal arithmetic. In the axiomatic arithmetic, 0 is a constant, x, y, etc. are variables, and x' denotes the number which immediately follows x . The axioms are those of the predicate calculus with equality, the axiom of induction, and a number of specific axioms, which may be, e.g., as follows:

(1) $x=y \Rightarrow x'=y'$

(2) $x'=y' \Rightarrow x=y$

(3) $\neg\, x'=0$

(4) $x+0 = 0$

(5) $x+y' = (x+y)'$

(see [1]; universal quantifiers are implied; we will not use multiplication.)

In the recursive arithmetic the numbers are: 0, 01, 011, etc. The predicate of equality and the function of addition are recursive functions defined in some algorithmic language. Our formalism is based on Refal (see [2-4]). The definition of equality and addition in Refal is:

#1.1 k=(0)(0) ⇒ T
#1.2 k=(e_x1)(e_y1) ⇒ k=(e_x)(e_y) ⊥
#1.3 k= e_x ⇒ F

#2.1 k+(e_x)(0) ⇒ e_x
#2.2 k+(e_x)(e_y1) ⇒ k+(e_x)(e_y) ⊥ 1

Essentials of Refal can be seen here even without reading the description of the language. Functions are defined on expressions, meaning by an expression any string of symbols and parentheses having correct structure with respect to parentheses. A function definition is a sequence of sentences, which are replacement rules, with ⇒ separating the right side from the left. A function call is represented by k $\mathcal{F}$ $\mathcal{E}$ ⊥, where $\mathcal{F}$ is a function symbol and $\mathcal{E}$ is the argument. Concretization brackets k and ⊥ obey the bracket syntax, the sign ⇒ serves as ⊥ for the initial k in the left side of a sentence. e_x, e_y, etc. are free e-variables, which can take any expressions as values. Free s-variables: s_x, s_y, etc. take only symbols as their values. In concretizing (evaluating) a function call, the first applicable sentence is used in each replacement step. The applicability of a sentence and the values of the free variables are determined in matching the function call with the left side. The Refal machine transforms workable expressions in its view-field, step by step, using sentences until there are no k-signs in the view-field.

Consider the theorem

(6) 0'' + 0''' = 0'''''

of the axiomatic arithmetic. To prove it, we start from the axiom (5) with x=0'', and y=0 :

(7) 0'' + 0' = (0''+ 0)'

Using axiom (4) with x=0'', we have:

(8) 0'' + 0 = 0''

From (7), (8), and the axioms for equality we derive:

(9) 0'' + 0' = 0'''

Proceding in this manner, we obtain a demonstration of (6) in two more steps.

The analogue of theorem (6) in the recursive arithmetic is the statement that the result of the evaluation of the function call k+(0l1)(0l11)⊥ is 0lllll . To prove this statement we only have to

put the former into the view-field of the Refal machine, to start the machine, and to check that when it stops, the contents of the view-field is the latter.

Consider now a statement with (implied) universal quantifiers:

(10) $\neg\, x' = 0$

which is an axiom in the axiomatic arithmetic. In the recursive arithmetic it is equivalent to the statement that the concretization of

(11) $k=(e_x 1)(0)_\perp$

is F with any e_x.

To formalize this statement, we introduce the function (predicate)

#3 $kP^1 e_x \Rightarrow k=(e_x 1)(0)_\perp$

Our statement now is: the definition #3 is <u>functionally equivalent</u> to the following definition:

#4 $kP^1 e_x \Rightarrow F$

(to be referred as an <u>F-identity</u>).

We say that a program (algorithm) $\mathcal{A}'$ in Refal is functionally equivalent to an algorithm $\mathcal{A}$ with respect to function $\mathcal{F}$, iff for every expression $\mathcal{E}$ from the domain of $\mathcal{F}$ according to $\mathcal{A}$, the concretization of $k\mathcal{F}\mathcal{E}\perp$ according to $\mathcal{A}'$ produces the same result as according to $\mathcal{A}$. The transformation of $\mathcal{A}$ into $\mathcal{A}'$ will be called an <u>equivalence transformation</u>. We note that the relation between $\mathcal{A}'$ and $\mathcal{A}$ is not symmetric, thus it is not a "relation of equivalency" in the usual mathematical sense. The domain of a function may be extended as a result of an equivalence transformation.

A system of rules for equivalence transformations in Refal has been formulated (see [3]). We shall not reproduce these rules in the present paper, but will use them in an informal manner. In addition to the rules of equivalence transformations, a strategy of applying these rules has been also formulated in [3], which results in an algorithm of equivalence transformation. We denote this algorithm Q without its formal definition. Instead we shall show in examples what it can, and what it cannot do.

It is easy to transform #3 into #4. We have only to "drive" through the Refal machine a set of workable expressions represented by a general Refal expression, which may include, unlike a workable expression, free variables. We call this procedure <u>driving</u>. To drive expression (11), we notice that neither #1.1, nor #1.2 will be found

applicable for concretization, whatever the value of e_x is. Hence #1.3 will be used, which gives F as the result.

This was one of the simplest cases of driving. Generally, the Refal sentence used in concretization step will depend on the values of the free variables, which leads to branching. The branches will correspond to certain subsets of the set of all possible values of each variable involved. We call this subsets contractions of the original full set, and represent them as substitutions for variables. For example, the contraction $(e_x \to Ae_x)$ defines:(1)the subset of the set of all expressions comprising all expressions which start with A; (2)the corresponding branching condition depending on the value of the variable e_x;(3)the new value of the variable e_x, which is the old value less the initial A . The set of all possible values of the new variable e_x will again be the full set of all expressions.

Consider the theorem:

$$0 + x = 0$$

which is proved by induction in the axiomatic arithmetic. In the recursive arithmetic, this is equivalent to the transformation of

#5 $\quad kP^2e_x \Rightarrow k=(k+(0)(e_x)_\perp)(e_x)_\perp$

into the T-identity: $kP^2e_x \Rightarrow T$.

Let us put

(12) $\quad kP^2e_{x\perp}$

into the view-field of the Refal machine. In one step we will have:

(13) $\quad k=(k+(0)(e_x)_\perp)(e_x)_\perp$

Apply driving to (13). The concretization of the function + call leads to the branching:

$$\begin{cases} (e_x \to 0)\text{: sentence \#2.1 will be used} \\ (e_x \to e_x)\text{: sentence \#2.2 will be used} \end{cases}$$

If the first branch is taken, we easily come to T as the final result of concretization. Taking the second branch, we have

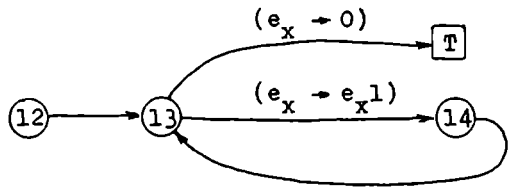
(14) $\quad k=(k+(0)(e_x)_\perp\ 1)(e_x\ 1)_\perp$

in the view-field as the result of the substitution for e_x and the subsequent step of the Refal machine using #2.2 .

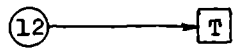
Now we use #1.2 to concretize (14), which gives:

$$k=(k+(0)(e_x)_\perp)(e_x)_\perp$$

We have come to exactly the same configuration in the view-field as it was at an earlier stage (13). The graph of states of the Refal machine which operates according to #5 is:



It is not difficult to recognize the structures of this kind, and transform them into:



which corresponds to a T-identity.

The equivalence transformation Q does it. In fact, it does only a little more: it knows how to produce simple generalizations representable in the form of pattern expressions. For example, two expressions: (ABC) and (AXYZ), may be generalized as (Ae_x). Using this generalization technique, Q performs a transformation which is equivalent to a one-time application of the induction principle to a hypothesis produced by generalization.

As our next example, consider the statement:

(15) $\quad x + y = y + x$

expressing the commutativity of addition. In recursive arithmetic it corresponds to the transformation of the predicate

#6 $\quad kP^c(e_x)(e_y) \Rightarrow k=(k+(e_x)(e_y)_\perp)(k+(e_y)(e_x)_\perp)_\perp$

into a T-identity.

Applying Q , we drive the function P^c call, and get

(16) $\quad k=(k+(e_x)(e_y)_\perp)(k+(e_y)(e_x)_\perp)_\perp$

in the view-field. The next step of driving produces the branching:

(17) $\quad (e_y \to 0)$: $\quad k=(e_x)(k+(0)(e_x)_\perp)_\perp$

(18) $\quad (e_y \to e_y 1)$: $\quad k=(k+(e_x)(e_y)_\perp\ 1)(k+(e_y 1)(e_x)_\perp)_\perp$

Configuration (17) is transformed into T by Q , as we saw in the preceding example. But configuration (18) causes trouble. The algorithm Q will drive the second + call at the next step, which will

lead to a branching on the value of e_x . The branch ($e_x \rightarrow 0$) will produce a configuration which is transformed by Q into T , like (17); the branch ($e_x \rightarrow e_x1$) will produce the configuration:

(19) $k=(k+(e_x1)(e_y)_\perp 1)(k+(e_y1)(e_x)_\perp 1)_\perp$

which after one more step of driving using #1.2 transforms to:

(20) $k=(k+(e_x1)(e_y)_\perp)(k+(e_y1)(e_x)_\perp)_\perp$

This configuration is not identical to (16). If we try to continue the transformation of (2) by Q, we only receive new configurations:

$k=(k+(e_x11)(e_y)_\perp)(k+(e_y11)(e_x)_\perp)_\perp$

etc., but never come back to the original configuration (16). But the only way for Q to transform a definition using induction is to loop to the same configuration in the course of generalized computation -- driving. Thus Q fails to prove the commutativity of addition.

Turning to the axiomatic arithmetic, we can see that the failure of Q to prove theorem (15) stems from the fact that a <u>double</u> induction loop is needed to prove it. Configuration (18) is:

(21) $(x + y)' = y' + x$

We firat prove by induction an auxilliary theorem:

(22) $y' + x = (y + x)'$

and then combine (22) and (21) by the transitivity of equality into

$(x + y)' = (y + x)'$

Using axiom (2), we come back to (15), which allows to close the second induction loop. Thus we have a loop nested in loop. The interaction of these two loops leads to the proliferation of new configurations in the straightforward computational approach,which dooms Q to failure.

The fact that statement (22) can be proved as a theorem, and that it will be useful, must be guessed somehow, and (22) must be set as a subgoal if we use the axiomatic approach. In our approach, we look for a different solution.

Suppose one of the free variables in the configuration (16), say e_y, is given a certain value. Then only one induction loop will be needed for transformation, and Q will be able to do the job. We have already seen it for $e_y \rightarrow 0$, configuration (17). Giving to e_y the values 01, 011,... etc., we reduce (16) to the configurations:

(23) $k=(e_x1)(k+(01)(e_x)_\perp)_\perp$
$k=(e_x11)(k+(011)(e_x)_\perp)_\perp$ ... etc.

each of which can be transformed by Q into T , as the reader can easily verify.

To prove the commutativity of addition for the case when both variables in P^c are arbitrary, we make a metasystem transition: we formalize Q as a recursive function in Refal (this will become a metasystem with respect to recursive arithmetic) and consider statements about Q . Our idea is to prove that the application of Q to any of the configurations (23) will produce T .Should we succeed in proving this, we have proved that configuration (16) can be replaced by T also. As the instrument of proof we choose the same algorithm Q which is applied to arithmetic statements. The big question is: will the new algorithm resulting from this self-application be more powerful than the original algorithm Q ? In particular, is it possible to prove the commutativity of addition in this way?

It is shown in [3] that the answer to this question is positive. In Sec. 2 we define the basic concepts which serve to formalize metasystem transition, and which lead, in particular,to the desired proof.

2. MST-formulas

The concept of metasystem transition was introduced and taken as the basis for the analysis of evolutionary processes in the author's book [5]. It was the philosophical background exposed in [5]that gave a push to the work on the Refal project in the mid-1960s. The language Refal was designed as the means to facilitate the formalization of metasystem transition. It finds a compromise between the complexity necessary to write non-trivial algorithms, and the simplicity necessary to formulate effective rules of equivalence transformations.

Metasystem transition is one of the main instruments of creative human thinking. To solve a problem, we first try to use some standard system of operations, rules, etc.If we fail we start to analyze why did we fail, and for this purpose we examine the process of applying our rules and operations. We construct a metasystem with respect to the ground-level system of rules and operations which would give us some new, more elaborate, rules and operations to solve the problem. If we fail once more, we analyze the processes on the first metasystem level, which means that we make a second metasystem transition. This time we create instruments which would help us, on the first metasystem level, create instruments to solve the ground-level problem. This transition from the use of an instrument to the analysis of its use and creation of instruments to produce instruments may be repeated

again and again; it stands behind the two and a half millennia of the development of contemporary mathematics. For a computer system to match the human being, it must model this process.

Since functions in Refal may be defined only on object expressions (i.e. not including free variables and concretization brackets), the representation of function definitions to be used in metasystem transition must transform sentences (and their parts: free variables, pattern expressions, function calls) into object expressions. We call this representation metacode .

We need not describe the metacode in full (although it is very simple); let us only show how free variables are encoded. Note that metasystem transition may be repeated many times, thus generating a multilevel system. The original functions, such as + , = , P^C , etc., will be referred to as functions of the ground (zero) metasystem level. Functions applied to transform (or generate) these functions, such as Q , will be referred to as being on the first metasystem level. Functions transforming the functions of the first metasystem level are said to be on the second metasystem level, and so on.

Variables of the ground metasystem level, like e_x, s_1, etc. represent sets of object expressions. In metacode (i.e. on the first metasystem level) they turn into non-terminals of the first order E_x, S_1, etc., which are not variables, but just regular symbols. The first metasystem level has, of course, its own free variables, which have again the usual form: e_x, s_1, etc. When we make a metasystem transition to the second level, they turn into first-order non-terminals, while first-order non-terminals E_x, S_1, etc. turn in metacode into second-order non-terminals E_x^2, S_1^2, etc.

The formalism which exploits the idea exposed in Sec.1 rests on the concept of integral metafunction, or metaintegral. In our case we are interested in the metafunction which will be denoted as:

$$k\int P^C(E_x)(e_y)\ \perp$$

We read it: the metaintegral of P^C over e_x. This is a function which depends on one variable e_y . For any value $\mathfrak{E}$ of e_y, the value of this function is the metacode representation of the definition of function

(24) $\quad kP^X(e_x) \Rightarrow kP^C(e_x)(\mathfrak{E})_\perp$

of one argument e_x .This function, of course will be different for different values of $\mathfrak{E}$ (a family of functions).

We introduce now the Refal-interpretation function R, whose definition is: if $[\underline{\text{def}}(\mathcal{F})]$ is the metacode of the definition of a function $\mathcal{F}$, and $\mathcal{E}$ is an expression from the domain of $\mathcal{F}$, then

(25) $\quad kR([\underline{\text{def}}(\mathcal{F})])\,\mathcal{E}\,\perp \;=\; k\,\mathcal{F}\,\mathcal{E}\,\perp$

Consider an equivalence transformation function Q. It is a metafunction which has the set of all correct metacodes of function definitions as its domain. By the definition of equivalence,

(26) $\quad kR(\,kQ[\underline{\text{def}}(\mathcal{F})]\,\perp)\mathcal{E}\,\perp = \; kR(\,[\underline{\text{def}}(\mathcal{F})])\,\mathcal{E}\,\perp$

Combining (25) and (26) we have:

(27) $\quad k\,\mathcal{F}\,\mathcal{E}\,\perp \;=\; kR(\,kQ[\underline{\text{def}}(\mathcal{F})]\perp)\,\mathcal{E}\,\perp$

By the definition of metaintegral,

(28) $\quad kR(k\int P^c(E_x)(e_y)\perp)(e_x)\perp \;=\; kP^c(e_x)(e_y)\perp$

Therefore, if we redefine function P^c in this way:

(29) $\quad kP^c(e_x)(e_y) \;\Rightarrow\; kR(k\int P^c(E_x)(e_y)\perp)(e_x)\perp$

this new definition will be equivalent to the old one. We call such definitions as (29) <u>MST-formulas</u> (MST standing for MetaSystem Transition). An MST-formula defines an equivalence transformation using one or more metasystem transitions.

Using (26) we obtain from (29) another MST-formula:

(30) $\quad kP^c(e_x)(e_y) \;\Rightarrow\; kR(kQk\int P^c(E_x)(e_y)\perp\perp)(e_x)\perp$

The algorithm of evaluating P^c according to (30) is this:

<u>Step 1</u>. Take the definition of P^c with a specific e_y, but with an arbitrary e_x. It will be a function of e_x; e.g., if e_y = Oll, this function will be

(31) $\quad kP^x(e_x) \;\Rightarrow\; k=(k+(e_x)(Oll)\perp)(k+(Oll)(e_x)\perp)\perp$

<u>Step 2.</u> Translate this function definition into metacode and transform by function Q. A new function definition results.

<u>Step 3</u>. Interpret this last definition with the specified e_x.

We know from Sec.1 that the function resulting from step 2 will always be identical T, because configurations (23) are successfully transformed by Q. But it is only our knowledge, and not yet a fact proven by machine. To have it proved, we must make one more metasystem transition. Let us define:

(32) $\quad kI^x(e_y) \;\Rightarrow\; kQk\int P^c(E_x)(e_y)\perp\perp$

and apply to this function transformation Q again. The metacode of (32) is denoted as $k\int I^X(E_y)_\perp$. According to (27):

(33) $\quad kI^X(e_y)_\perp \quad = \quad kR(kQk\int I^X(E_y)_\perp{}_\perp)(e_y)_\perp$

From (30), (32), and (33) we obtain this MST-formula:

(34) $\quad kP^C(e_x)(e_y) \;\Rightarrow\; kR(kR(kQk\int I^X(E_y)_\perp{}_\perp)(e_y)_\perp)(e_x)_\perp$

Together with (32), it defines an equivalence transformation, but it is not yet the final form of the transformed definition. To get the final form, we use an equivalence transformation once more: to simplify the definition (34). This last procedure transforms (34) into a T-identity. The same function Q may be formally used, but even a much simpler technique will suffice on this stage. The essential part of it is just a computation: that of the value of $kQk\int I^X(E_y)_\perp{}_\perp$. The result will be:

(35) $\quad [kI^X(e_y) \;\Rightarrow\; [kP^X(e_x) \;\Rightarrow\; T]]$

where by bracketing a sentence we denote its metacode.(compare with (32) and (24)). This result is a formal proof by Q that Q proves any definition of type (31) to be a T-identity. It is interesting to note that theorem (22), which in the axiomatic approach must be guessed as a useful subgoal, in our approach appears automatically in the course of computation (and, of course, is easily proven by Q, since it requires only one induction loop).

Using (35) in the right side of (34), we first obtain:

$$kR([kP^X(e_x) \;\Rightarrow\; T])(e_x)_\perp$$

by virtue of the definition of R, and then - by the same definition - T , which completes the formal proof of the commutativity of addition.

It should be stressed that after function Q has been once defined, the use of MST-formulas like (34) is a purely mechanical process: we just write the formula and apply Q to it. One can write many different MST-formulas splitting variables in different ways between two, three,etc. metasystem levels and returning to the ground level by using the interpretation function R .

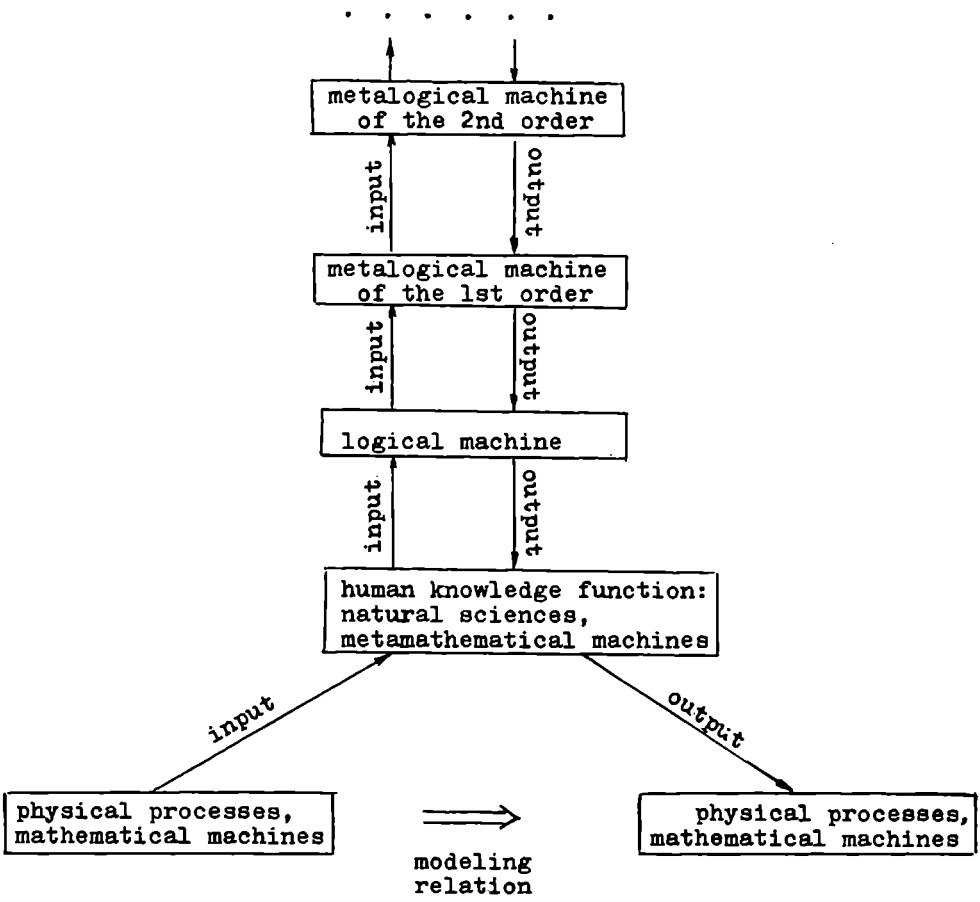
The algorithm Q must be of certain minimal complexity in order to deal successfully with itself. When this level of complexity is achieved, we have a good reason to believe that the second, third, etc. metasystem transition will also be successful (i.e.provide more and more powerful algorithms), because at each next step of this stairway Q is still applied to itself. It seems plausible that the number of metasystem transitions we have to make in the computational approach

is equal to the number of nested loops of induction, but it has not been demonstrated in a rigorous manner.

Theorem proving and program optimization are indistinguishable in our approach, they are two applications of the same functional equivalence transformation. The algorithm Q, although not very strong in proofs by induction, is strong enough to ensure some important types of optimization (see [3]). Coupled with metasystem transition, it should become a powerful instrument of program optimization.

3. System approach to mathematical knowledge

Our approach to theorem proving is not based on mathematical logic in its traditional form. It should be viewed as constituting first steps toward creation of a system of mathematical knowledge, the general plan of which is represented below:



Human knowledge at any moment of time can be seen as a function which receives a material process as its input, and produces (or does not produce) its model as the output. Mathematics is the part of human knowledge which models special type of processes: mathematical machines. Arithmetic functions may serve as example of mathematical machines. They are used by the human knowledge function to create models of physical processes; at the same time they are an object of study of mathematics, and the result of this study is a metamathematical machine.

Our function Q is an example of a metamathematical machine. Equivalence transformation is a conctruction of a model. Metamathematical machines are created and improved using logic. We used logic when obtained a new Q from the old one. Logic, generally, creates new knowledge, makes the human knowledge function evolutionize. Formalization of logic creates a logical machine, which can be deterministic or nondeterministic.

Traditional formal logic is a nondeterministic machine which generates demonstrations; the metamathematical machine in this approach is trivial: it just keeps the knowledge created (the theorems proven) up to date, and outputs them on request. In our approach all the machines will be deterministic, and we are not going to limit the hierarchy by any definite metasystem level. We start with an intelligent metamathematical machine Q ; then create an intelligent logic, which chooses the MST-formula to be used; then create a metalogic, which tries different logics; etc. No metalogic may be supreme, because a more powerful one can always be created by a metasystem transition. There exists no ultimate criterion of the reliability of a logic or metalogic other than proof in practice.

Starting with Goedel's theorem, metasystem transition has been extensively used in logic and mathematics to obtain negative results (incompleteness, unsolvability). We embark on using metasystem transition in a positive way: to expand actually, and in the needed direction, the transforming power of each specific machine, not only to show that it has limits. Although the power of each machine remains, of course, limited, the process of expansion is unlimited -- as far as we can see it now. Goedel's theorem and other negative results set limits to those systems which do not incorporate metasysten transition. Our approach does incorporate metasystem transition as one of its formalized elements. This is why it is free from Goedel's limits. What other limits it has, if any, is not easily

seen at the present time.

****

REFERENCES

1. Kleene, S.C., Mathematical Logic, J.Wiley & Sons, 1967.

2. Turchin, V.F., A supercompiler system based on the language Refal, SIGPLAN Notices, 14, No 2, pp.46-54, Febr. 1979.

3. Turchin, V.F., The Language Refal -- The Theory of Compilation and Metasystem Analysis,Courant Computer Science Report #20, February 1980, New York University, 1980.

4. Turchin, V.F., Semantic definitions in Refal and automatic production of compilers, Proc. of the Workshop on Semantics-Directed Compiler Generation, Aarhus University, Denmark, January 14-18 1980, Springer Verlag, 1980.

5. Turchin, V.F., The Phenomenon of Science, Columbia Univ. Press New York, 1977.