SEMANTIC DEFINITIONS IN REFAL AND AUTOMATIC PRODUCTION OF COMPILERS

Valentin F. Turchin

(The City College, The City University of New York)

1. Introduction

What does it mean to define the semantics of an algorithmic language? The most straightforward definition will be the *interpretive* one: to construct a machine which upon receiving a text (program) written in that language and a work object (the set of data the program is to be applied to), would execute the program, step by step, according to the algorithmic intention of its author. Thus, a metalanguage to define (semantically) algorithmic languages should formally describe machines; i.e., algorithms, which is to say that it must again be an algorithmic language. The language Refal was designed as such a language, which is both algorithmic and a metalanguage to deal with algorithms. An outline of the purpose and the main features of Refal may be found in [1].*

*This paper also contains a bibliography list on Refal.

A detailed presentation of this language, together with a theory of compilation using it, is given in [2]. In the present paper we limit ourselves to the main concepts of the theory of compilation, which we introduce using a very simple language as an example; then we show how a compiler for a language defined in Refal may be produced automatically through a double *metasystem transition*. The formal definition of Refal is presented as an Appendix.

To begin with, let us look into how a programming system employing Refal as the means to introduce new algorithmic languages might work.

Let A be an algorithm written in a certain language, and E a work object. To define the language we define in Refal a recursive function with a determiner L (which identifies the language) in such a way that the process of concretizing the expression

(1) $$\mathrm{k}\ L\ A\ (E)\ \perp$$

could be seen as (or, will model) the application of the algorithm A to the object E. In particular, the result of the concretization (when it exists) should be the result of the use of A on E. In programming terms, the program is *interpreted* here, thus the function L will be called *the interpreting function* of the language. Since Refal allows the use of any object signs, there is no restriction on the composition of program A and work object E: the algorithmic language to be defined is allowed to use any characters different from those depicting the specific signs of Refal. We might consider

A and *E* as arbitrary strings of object signs, but nothing prevents us from introducing Refal parentheses into these strings, thus making them generally *object expressions*. If the object language uses parentheses in the way they are usually used (to create trees), it is convenient to identify them with the structure brackets in Refal.

So, we have a formal description of the algorithmic language *L* through its interpreting function. How do we use it?

If we have a computer implementation of the Refal machine (an interpreter or a semicompiler), we can use the language *L* in the following manner. Each time that we have to execute an algorithm *A* written in *L*, the expression to apply this algorithm to being *E*, we form the working expression (1) in the view-field of the Refal machine and start it into action. We will obtain the desired result in this way, but understandably this is not an efficient way to use a programming language systematically, because this is an interpretation mode. Can we improve the efficiency by turning to a compilation mode? What is compilation?

2. Graph of States

Let us examine it in a very simple example of a language with the interpreting function /L/ defined by the following sentences:

L.1 $k/L/\ e_1;\ e_2(e_a) \Rightarrow k/L/e_2\ (k/L1/e_1(e_a) \perp)$

L.2 $k/L/e_1(e_a) \Rightarrow k/L1/e_1\ (e_a) \perp$

L1.1 $k/L1/CROSS(s_1\ e_2)(s_3\ e_4) \Rightarrow s_1 s_2\ k/L1/CROSS(e_1)(e_2) \perp$

L1.2 $k/L1/CROSS\ (e_1)(e_2) \Rightarrow e_1\ e_2$

L1.3 $k/L1/ADD(e_1)(e_a) \Rightarrow e_a\ e_1$

Here the first sentence indicates that a text in the language /L/ may be formed as a sequence of instructions separated by semicolons, and the instructions are executed from left to right, being applied each time to the result of the execution of the preceding instruction. Function /L1/ defines the execution of separated instructions. There are only two kinds of instructions: CROSS and ADD. Instruction CROSS(P) "crosses" the work object with the word P by putting their symbols in alternation until one of the words is exhausted (we assume that the objects which the language /L/ deals with are strings of symbols). Instruction ADD(P) adds the word P at the end of the work object e_a. Here is an example of a program:

CROSS(CAT); ADD(DOG).

In order to execute it on the word LION as the work object (input data), we put into the view-field of the Refal machine:

$k/L/CROSS(CAT);\ ADD(DOG)\ \ (LION) \perp .$

The concretization of this expression gives:

CLAITONDOG.

Now suppose we have some *object machine* M^0, and we want to translate our program into the language of M^0. Let M^0 have two fields, referred to as *object* and *result* in which the object and the result of work are stored and gradually transformed, and let it be able to perform certain simple operations, which we will describe in English. What do we do to translate the program on the basis of the interpreting function /L/ defined in Refal? We analyze the process of interpretation of this program with some general, not exactly specified input data, and describe the operation of the Refal machine in the language understandable by M^0. We imagine that the following expression is put in the view-field of the Refal machine:

(1) k/L/CROSS(CAT); ADD(DOG) $(e_x) \perp$

which is, of course, impossible literally because of the free variable e_x, which represents the set of all expressions and not a specific expression. Then we *drive*, so to say, the set (1) through the Refal machine; i.e., trace what is happening to its elements when they are put into the view-field, and the Refal machine is started. The rules of driving are algorithmically formulated in [2]; in the present paper we perform driving informally.

A set of workable expressions defined by a general (possibly containing free variables) expression is called a *configuration* (generalized state) of the Refal machine. There are two cases of driving a configuration:

a) The sentence used by the Refal machine does not depend on the value(s) of free variable(s) (if any) in the configuration. E.g., at the first step of driving (1) the sentence L.1 will be used no matter what the value of e_x is. Therefore, we can execute one step of the Refal machine as if e_x were a specific expression. The result will be the following configuration:

(2) k/L/ADD(DOG)(k/L1/CROSS(CAT)(e_x) $\perp$) $\perp$

b) With different values of the free variable(s) in the configuration, different sentences will be used by the Refal machine. In this case it is necessary to split the full set corresponding to the configuration into a number of subsets such that to all the elements (workable expressions) of a given subset the same sentence corresponds. Thus, a branching appears, because the history of each of these subsets must be traced on. When we continue to drive configuration (2), the call of function /L1/ is to be concretized first; if the value of e_x starts with a symbol, sentence L1.1 will be used in driving; if the value of e_x is empty, the sentence used will be L1.2. We say that two *contractions* of the variable e_x are made at this stage of driving:

(c.1) $e_x \to s_1 e_x$

and

(c.2) $e_x \to \square$

(The symbol $\square$ represents the empty expression.)

Actually contracted is, of course, the set of expressions represented by e_x. But when a value of e_x is given, the contraction reads as the predicate which says whether this value is found in the contracted set represented by the right side; in addition, the values of the variables entering the right side are (re)defined. E.g., if the value of e_x is 'LION', then as the result of contraction (c.1) s_1 becomes 'L', and e_x becomes 'ION', while contraction (c.2) is impossible.

The result of the repeated driving of the initial configuration may be represented as the *graph of states* of the Refal machine. The graph of states for the initial configuration (1) is shown in Fig. 1.

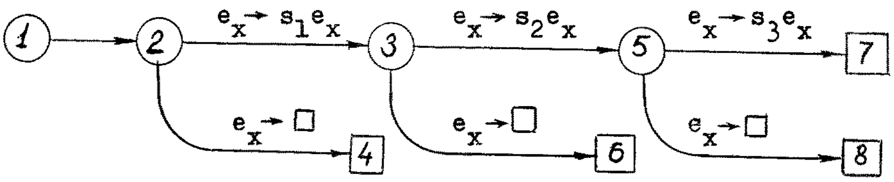

Fig. 1

The vertices of the graph of states are configurations, which are shown as circles if they are *active* (include at least one k-sign) and as squares if they are *passive* (no k-signs). The remaining configurations in Fig. 1 are:

(3) k/L/ADD(DOG)(Cs_1 k/L1/CROSS(AT)(e_x) ⊥) ⊥

(4) CATDOG

(5) k/L/ADD(DOG)(Cs_1As_2 k/L1/CROSS(T)(e_x) ⊥) ⊥

(6) Cs_1ATDOG

(7) Cs_1As_2T$s_3$$e_x$DOG

(8) Cs_1As_2TDOG

The arcs in Fig. 1 are of the *dynamic* type only; they represent one or more steps of the Refal machine, are ordered and bear contractions. There may be two more types of arcs in a graph of states. Configuration (2) could be represented as the *composition* of configurations

(9) $k/L/ADD(DOG)(e_y) \perp$

and

(10) $k/L1/CROSS(CAT)(e_x) \perp$

as shown in Fig. 2a by a vertical (wavy) arc which bears the *computed variable* e_y. The broken line in Fig. 2a is a *representation* arc, which does not depict any operation of the Refal machine, but only a change in the way we represent the current state.

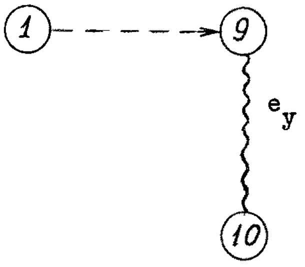

Fig.2a

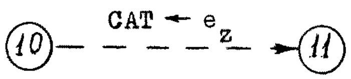

Fig.2b

Configuration (10), in its turn, could be represented as a special case of the more general configuration.

(11) $k/L1/CROSS(e_z)e_x \perp$

as shown in Fig. 2b by a representation arc, which bears the assignment of the expression 'CAT' to the variable e_z.

Instead of the usual form of assignment

$$e_z := E$$

where E is any expression, we use the form

$$E \leftarrow e_z$$

which may seem strange at first glance, but in fact is very natural and convenient in the analysis of graphs of states and permits better understanding of the relationship between the contraction and the assignment. This notation is a part of a consistent system of notation, based on the following principles:

(1) In writing a substitution we always use an arrow which is directed from the variable to be replaced to the sub stituting expression.

(2) Seen another way, a substitution may reflect a relationship between two groups of variables: those of the first group are *old* variables; i.e., they are already defined (have values), those of the second group are *new*; i.e., they get defined by the substitution. We shall always put the old variables on the left and the new on the right of the substitution formula. Thus, two types of substitution emerge, contractions and assignments, as presented in the following scheme:

	Old Variables (already defined)		New Variables (being defined)
Contraction	V	$\rightarrow$	L
Assignment	E	$\leftarrow$	V

where L is an L-expression including (possibly) new variables, and E is any expression, which may include old variables; V is a single variable.

(3) In the notation of substitution, the variable which is to be replaced and the expression in which the replacement must be performed make a pair separated by the substitution sign //, and the arrow points to the substituted expression. One form is:

$$E//(V \rightarrow E').$$

Another form, completely equivalent to the first one, is:

$$(E' \leftarrow V)//E.$$

(4) When we construct a graph of states we move from left to right defining new variables. Therefore the lists of both contractions and assignments will be lengthened (and read) from left to right. But because of the different directions of the substitution arrows, the law of composition of substitutions will be different for contractions and assignments, although equally easily suggested by our representation:

$$(V \rightarrow L^1)\ (V \rightarrow L^2) = V \rightarrow (L//\ V \rightarrow L)$$

$$(E^1 \leftarrow V)\ (E^2 \leftarrow V) = (E^1 \leftarrow V//E^2) \leftarrow V$$

Tracing the graph of states of the Refal machine we simultaneously *map* it onto the object machine M^0, compiling instructions for M^0 so as to keep correspondence between the generalized states of the Refal machine and the machine M^0. To each configuration in the graph of states of the Refal machine a control point in the program for M^0 will correspond, while the variables in the graph of states are mapped on the information field in M^0. When a self-sufficient graph of states is constructed the process of compilation is completed. Proceeding in this manner, we *compile* the following object program:

1. *Object* assumes its input value, *result* becomes empty.

2. If *object* begins with a symbol s_1 , it is deleted, and Cs_1 is added to *result*, otherwise *result* becomes CATDOG, and go to *End*.

3. If *object* begins with a symbol s_2 , it is deleted, and As_2 is added to *result* otherwise ATDOG is added to *result*, and go to *End*.

4. If *object* begins with s_3 , and the rest is e_4, then Ts_3 e_4 DOG is added to *result*, otherwise T *object* DOG is added to *result*.

5. End.

In the general case of a language L and an algorithm A in that language, the expression

$$k\ L\ A\ (e_x)\ \perp$$

must be driven through the Refal machine, and the goal of the theory of compilation is to examine this process and describe the operations performed on the argument e_x in the language of the object machine M^0. If this theory were to be elaborated bearing in mind one definite language *L*, that is drawing upon its specific features, then the theory would result in an algorithm of compilation from this language *L*. But we shall not bear in mind any specific language, of course. The theory of compilation should be applicable to any texts in Refal, its goal being to design *one universal algorithm* to compile from any language, had its interpreting function been defined in Refal.

3. Compilation Strategy.

When we have finished the construction of a self-sufficient graph of states, we have represented the set of all possible states (with a given start) as compositions of certain subsets - configurations. Thus, constructing a graph of states produces a set of configurations. Conversely, if we specify, no matter how, a set of configurations which we will call *basic*, and if we agree that only *basic* configurations may enter the graph of states, we will to a considerable extent define the graph of states to be constructed. Into the set of basic configurations we include, of course, only *active* configurations: there is no point in restricting passive configurations to enter a graph of states. The general scheme of constructing a graph of states is as follows. Starting with the initial configuration, we perform driving, and every time that we receive an active configuration we take a decision as to whether to continue driving or to express the configuration through some explored basic configurations and stop driving. Thus a *strategy of compilation* must be defined to construct a self-sufficient graph of states. In particular, a set of basic configurations must be defined which is of a paramount importance for the final product of compilation.

The choice of basic configurations determines the *depth* of compilation. The more specific the basic configurations are, the deeper the compilation process will go, and when the basic configurations are more general, the resulting program will retain a higher level of interpretation. Thus, the characterization of a program in terms of interpretation versus compilation, familiar to every programmer, becomes more comprehensible and receives a formal definition: it is the generality of configurations chosen as basic in constructing the graph of states.

The process of compilation may be controlled by including some specific configurations into the set of basic configurations, or, on the contrary stating that configurations of a certain kind *should not* become basic by any means (and therefore they will never be recipients of dynamic arcs, which means that they can be excluded, if necessary, from the final graph of states). Changing the compilation strategy, and varying the level of compilation thereby, we may receive different programs from the same initial definition of the problem in REFAL. For an example, the graph of states in Fig. 1 is the result of a strategy which declares basic each new configuration appearing in the process of driving. The corresponding program for M^0

is highly compilative and efficient. But suppose that instead of "CAT" in the formulation of the problem we have a word of 100 letters. Then the graph of states will contain 100 branching points, and the resulting object program will be quite bulky. We may desire -- as a trade off between space and time parameters -- to make the program more compact at the expense of retaining a level of interpretation. We declare configuration (11) basic. Then when configuration (2) first appears it gets represented as shown in Fig. 2a and Fig. 2b, and the final graph will be as shown in Fig. 3.

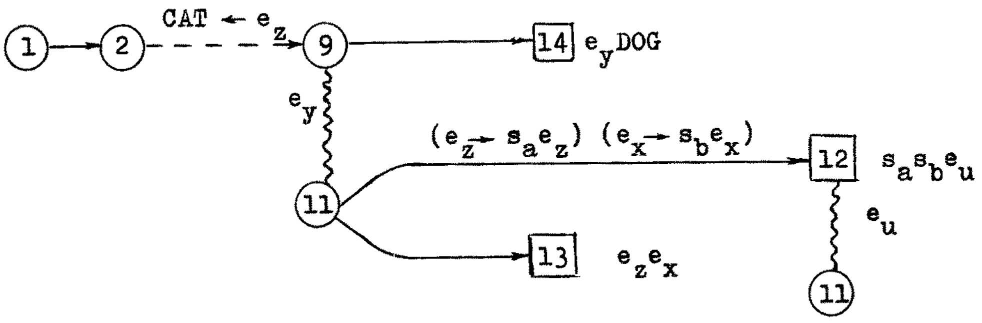

Fig.3

We see here an example of mixed strategy: decomposition of the text in the language /L/ into statements and execution of the first statement are done at compile time, but the second statement -- procedure of "crossing", which, of course, could have a longer word than "CAT" as the first argument -- is interpreted.

4. Perfect Graphs.

A *walk* in a graph of states is a sequence of alternate vertices and arcs $V_1A_1V_2A_2 \ldots V_{k-1}A_{k-1}A_k$ which "might be" followed (passed) by the Refal machine with some definite values of the input variables (*exact input state*). When we say "might be" here, we mean that the actual existence of an exact input state which brings the Refal machine to make this walk is not presupposed; a walk exists if certain rules are observed in its construction, which ensure that in order to define a computed variable we first go down the composition arc and on coming to a passive configuration on this function call come back up the same arc. In referencing to walks we shall list their vertices only, and show downs and ups by left and right brackets respectively.

An *input set* is a set of exact input states. In particular, an input set may be an *input class*; it is specified when contractions (possibly trivial) are specified for each input variable. To each walk an input set corresponds which comprises all those exact input states starting from which the Refal machine will make this walk. A walk is called *feasible* if the corresponding input state is not empty, otherwise it is *unfeasible*. A graph of states in which all possible walks are feasible is called *perfect*.

The graph in Fig. 1 is perfect. We can easily find a corresponding input set for each possible walk in it, and this set will not be empty. E.g., the input set for the walk

1,2,4 consists of one element, which is the empty expression. For the walk 1,2,3,5,8 the input set is $s_1\ s_2$, etc. The graph in Fig. 3 is not perfect, however. The walk 1,2,9[11,12[11,12[11,12[11,12[11,13]]]]],14 is, e.g., unfeasible because with the value 'CAT' assigned to e_z the Refal machine will never make more than three cycles of the loop. This illustrates the general point that the more interpretive an algorithm is, the less perfect is its graph of states. The process of compilation using a strategy compilative enough can considerably improve an algorithm with imperfect (in the defined sense) graph of states. An algorithm whose graph of states is perfect cannot be improved by compilation process alone.

As one more example, consider the algorithm which scans the argument (supposed to be a string of characters) twice, changing every A to B during the first scan, and every B to C during the second scan:

$$kFe_1 \Rightarrow kF^b\ kF^a e_1 \perp \perp$$

where functions F^a and F^b are defined by

$$\begin{aligned}
&kF^aAe_1 &&\Rightarrow B\ kF^ae_1 \perp\\
&kF^as_1e_2 &&\Rightarrow s_1\ kF^ae_2 \perp\\
&kF^a &&\Rightarrow\\
&kF^bBe_1 &&\Rightarrow C\ kF^be_1 \perp\\
&kF^bs_1e_2 &&\Rightarrow s_1\ kF^be_2 \perp\\
&kF^b &&\Rightarrow
\end{aligned}$$

The corresponding graph of states is represented in Fig. 4.

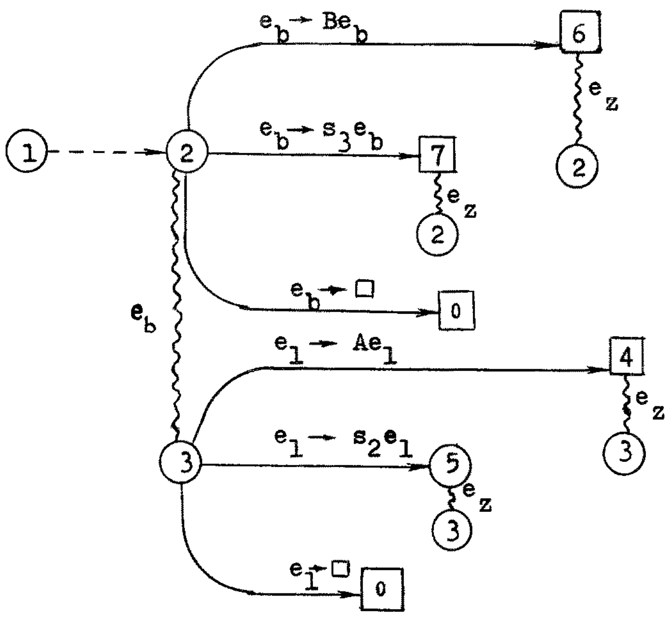

Fig.4

It is far from being perfect. For example, the walk 1,2[3,0],6,2,0 is unfeasible. Moreover, any walk of the form 1,2[3,W_3],W_2 , where the numbers of arcs in the walks W_3 and W_2 are not equal, is unfeasible. This is the reflection of the organization of the procedure as a double passage of the argument.

A simple compilation strategy, formulated and discussed in [2] transforms this graph into the algorithm which passes the argument once and changes both A's and B's into C's. Its graph is shown in Fig. 5. It is perfect.

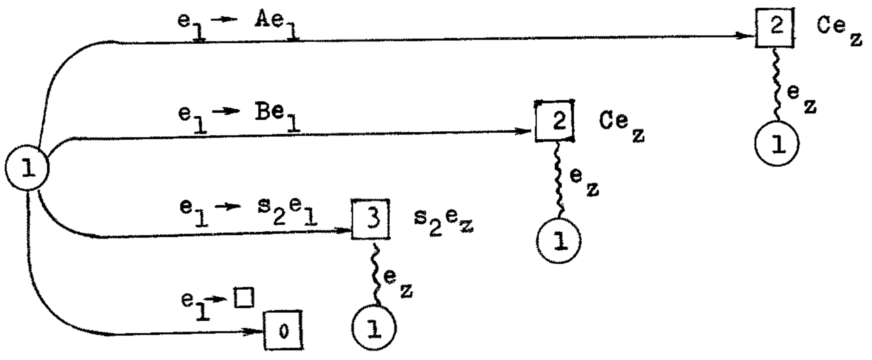

Fig.5

The following theorem is proved in [2].

THEOREM. There exists no algorithm which could transform any graph of states into an equivalent perfect graph.

5. Automatic Production of Compilers by a Supercompiler

A program which transforms an interpretive Refal program into a compilative program for an object machine M^0 is called a *supercompiler*. If the supercompiler is also written in Refal, it allows, for any language L defined in Refal, to produce automatically compilers which translate from L into the language of M^0 and are run on M^0, so that the user of such a compiler may never discover that Refal was used in its creation.

For a greater clarity of presentation, let us represent the functioning of M^0 with the help of "concretization sign"

k^M, like the regular sign k represents the functioning of the Refal machine. Thus

$$k^M(P^M)(e_1)(e_2)\dots(e_n\rangle \perp$$

will signify the work of M^0 with the program P^M and n pieces of input information $e_1,e_2,\dots,e_n$.

Let us denote the supercompiler function defined in Refal C^S, so that the concretization of

$$k\ C^S\ P \perp$$

where P is some representation of a Refal program, gives an equivalent program P^M for M^0. As the basis for the representation in question we choose the graph of states corresponding to a text in Refal, not the sequence of sentences. The transformation of a graph of states into the corresponding expression P will be called *the metacode.* We do not need here a full definition of the metacode, only some major points.

The first problem we have is to transform free variables into expressions. It will be achieved by changing e, s, and t into *E, *S, and *T, respectively. E.g., the variable e_1 will become *E1 in metacode, s_b will become *SB etc. Because of this agreement the asterisk * becomes a special symbol, and it will turn into *V in metacode, to avoid ambiguity. Other symbols and parentheses will remain themselves.

Arcs in the graph of states will be represented as concatenations of parenthesized contractions and assignments-branchings being rendered by parallel parentheses structures. E.g., if there is a triple branching at the start, the metacode will have the structure

$$(G_1 G_2 G_3).$$

Syntactically, a graph is always represented by a term, so that if G_v represents the graph for a vertex v, then to add an assignment or contraction S_v leading to v, we just write $S_v G_v$.

The graph of state for a function F, which has, say, two arguments e_a and e_b will be denoted as

$$\gamma F(\text{*EA},\text{*EB}).$$

By the definition of the supercompiler

$$kF(e_a)(e_b)\perp \equiv k^M(kC^s\gamma F(\text{*EA},\text{*EB})\perp)(e_a)(e_b)\perp$$

Suppose now that we have a language defined in Refal by its interpreting function L, so that concretizing

(I) $$k\ L\ \mathcal{P}\ (\mathcal{D})\perp$$

is applying program $\mathcal{P}$ in L to input data $\mathcal{D}$. Let us examine different ways of using language L.

First of all, we can just run the Refal machine (implemented interpretively on a computer) with the initial view-field (I). This will be a pure interpretation.

The most straightforward way to use the supercompiler and the object machine is to translate the Refal program for L into language of the object machine with the supercompiler and turn over the result to the target machine for execution. Symbolically, we must perform the following actions:

(CI.1) $kC^s\ \gamma L(\text{*EP},\text{*ED})\perp$ result denoted P^L

(CI.2) $k^M(P^L)(\mathcal{P})(\mathcal{D})\perp$

Program P^L is an interpreter of L compiled for M^0. To produce P^L we use the Refal interpreter only once (step CI.1). Then for each pair $\mathcal{P},\mathcal{D}$ we use M^0. Although this is much more efficient than using the Refal interpreter each time according to (I), it is not yet efficient enough, because step CI.2 remains interpretive.

To produce a compiled (efficient) equivalent of $\mathcal{P}$ we must use supercompiler with program $\mathcal{P}$ specified. The argument of C^s will be the graph of state for the initial configuration

$$k\ L\ \mathcal{P}\ (e_d)\ \perp$$

with the first argument given and the second arbitrary (free variable). This graph of states in metacode is:

$$(\mathcal{P} \leftarrow \text{*EP})\gamma\text{L}(\text{*EP},\text{*ED}).$$

Thus, the first step will be

(C.1) $\qquad kC^s(\mathcal{P} \leftarrow \text{*EP})\gamma\text{L}(\text{*EP},\text{*ED})\ \perp$

result denoted P^{LP}.

Program P^{LP} is an efficient program for M^0, which is the translation of program $\mathcal{P}$ in L. Since the variable *EP in (C.1) has been assigned a value, P^{LP} demands only one argument e_d, and the second step is

(C.2) $\qquad k^M(P^{LP})(\mathcal{D})\ \perp$

Although the execution of the algorithm $\mathcal{P}$ is now compilative and efficient, the compilation process defined by (C.1) is still in the interpretation mode and uses the Refal interpreter. Can we execute it on M^0 and in the compilation mode?

The process of compilation (C.1) depends on P. Let us introduce and define in Refal the function C^L which is compiler for L and may have any program e_p as argument:

$$kC^L(e_p) \Rightarrow kC^S(e_p \leftarrow *EP)\gamma L(*EP,*ED) \perp$$

Now, instead of just concretizing $kC^L(P) \perp$, as in (C.1) we first compile this function using C^S and run it on the machine M^0. The graph of states for configuration $kC^L(e_p)$ is shown in Fig. 6.

$$\begin{tikzcd} \textcircled{1} \arrow[rr, dashed, "((e_p \leftarrow *EP)\ \gamma L(*EP,*ED) \leftarrow e_g)"] & & \textcircled{2} \\ kC^L\ e_p \perp & & kC^S\ e_g \perp \end{tikzcd}$$

Fig.6

The metacode of this graph is:

$$\gamma C^L(*EP) \equiv ((*EP \leftarrow *VEP)\gamma L^*(*VEP,*VED) \leftarrow *EG)\gamma C^S(*EG)$$

where the asterisk at γL^* shows that the whole of the metacode γL should be subject to the (second) metacode transformation, not only its input variables (*EP,*ED), which are shown transformed: (*EVP,*EVD).

Thus the use of the language L will include now (CC case: Compiled Compiler) three steps:

(CC.1) kC^{S}((*EP ← *VEP)γL*(*VEP,*VED) ← *EG)γC^{S}(*EG) ⊥

result denoted C^{LM}

(CC.2) $k^{M}(C^{LM})(\mathcal{P})$ ⊥

result denoted P^{LP}

(CC.3) $k^{M}(P^{LP})(\mathcal{D})$ ⊥ .

The result of the last step will be, of course, the same as that of (I): application of $\mathcal{P}$ to $\mathcal{D}$.

In the CC case only the first step, production of the compiler C^{LM}, is executed on the Refal interpreter, and only once for each language L. But even this step can be moved to the M^{0}-machine by one more "metasystem transition", which will give us a compiler compiler. One can see that step CC.1 depends only on the definition of L in double metacode: γL*(*VEP,*VED). Thus, we define the function which produces compilers, having the definition of a language e_{ℓ} as input:

$kC^{C}(e_{\ell})$ => kC^{S}((*EP ← *VEP)e_{ℓ} ← *EG)γC^{S}(*EG) ⊥ .

The use of a compiler compiler (case CCC: compiler of compiled compilers) includes four steps, the first one being:

(CCC.1) kC^{S}(((*VEP ← *VVEP)*EL ← *VEG)

γC^{S*}(*VEG) ← *EG)γC^{S}(*EG) ⊥

result denoted C^{CL}.

In the second step we use the definition $\mathcal{L}$ of the language L and produce a compiler for L:

(CCC.2) $k^M(C^{CL})(L) \perp$

result denoted C^{LM}.

The last two steps are the same as in case CC.

One can see that in (CC.1) the supercompiler C^S is applied to its own definition. The derivation of this formula was referred to in A. Ershov's work [3] as "Turchin's theorem of double driving." Formula (CCC.1) involves triple driving: use of C^S on the application of C^S to C^S.

Let us sum up the main features of the supercompiler system.

(1) *Refal* is used both as the *algorithmic language* and as the *metalanguage* of the system. Formally, all algorithms are written in Refal, but in fact one can define any language through its interpreting function, and then write in that language. One can construct hierarchies of languages, defining one language through others.

(2) The system includes a *Refal-interpreter*, so as to *debug* programs in the interpretation mode. This makes the debugging process closest to the terms in which the program is written.

(3) The system includes a *supercompiler*, which transforms a Refal program into an efficient program for an *object machine*. Counting on the supercompiler, we can program in a much freer style than if the program is expected to be interpreted. We can use very general algorithms, which are not efficient when executed literally; i.e., interpreted, but with the arguments

partially specified, may be turned into efficient algorithms by the supercompiler. The use of a language defined through its interpreting function is only one special case of this style.

(4) Operations and algorithms not defined in Refal can be used as *external functions*, provided that *translation statements*, which show how these operations should be performed in the target machine, are available to the supercompiler.

(5) One part of the supercompiler's job is the *compilation process*, which is one of the basic *optimization* tools. The user may control this process by choosing a *compilation strategy* and modifying it depending on the results of compilation. Making a number of trials, an optimal point on the interpretation-compilation axis may be chosen; i.e., the desired trade-off between the size and the speed of the program achieved.

(6) The second part of the supercompiler's job is the *mapping* of the Refal-machine on the target machine. When the user programs in Refal, he defines his formal objects (data structures) as Refal-expressions, in a mathematical style. After debugging, which, as we mentioned above, should be done with the Refal-interpreter and in terms of Refal-expressions, the user may partially or completely specify the mapping of the Refal-configurations on the object machine. Different mappings may be tried to achieve better performance. Those configurations for which no mapping was indicated will be *mapped automatically* by the supercompiler. Since the mapping is made when the algorithm has already been formally defined, it is possible

to adjust automatic mapping to the algorithm to achieve high efficiency. On this way it is possible to free the user completely of so tedious a job as organizing and describing data for a real computer system. He will be dealing only with a mathematical model.

(7) If an algorithmic language L defined in Refal is expected to be used for a class of problems, an *efficient compiler* from L can be *produced automatically*. It will be run on the object machine and will translate programs in L into the language of the target machine. The user of the language L may or may not know anything about Refal and the way the complier from L was made.

APPENDIX

Formal Definition of Basic Refal

I. Syntax

A considerable part of the syntax will be described in the Backus Normal form.

I.1 Signs.

```
<sign> ::= <specific sign> | <object sign>
<specific sign> ::= #|/|<bracket>|<variable type sign>
<bracket> ::= <structure bracket>|<concretization bracket>
<structure bracket> ::= (|)
<concretization bracket> ::= k| ⊥ | =>
<variable type sign> ::= s|t|e
```

Object signs are capital Latin letters and other signs which are different from specific signs. The set of all object signs is assumed to be finite.

I.2 Symbols and Expressions.

```
<symbol> ::= <object sign>|<compound symbol>
<compound symbol> ::= /<object string>/
<object string> ::= <object sign>|<object string><object sign>
<expression> ::= <empty>|<expression><term>
<empty> ::=
<term> ::= <symbol>|<variable>|(<expression>)|k<expression>⊥
<variable> ::= <simple variable>|<specified variable>
```

<simple variable> ::= <variable type sign><index>

<index> ::= <object sign>

<specified variable> ::= s <specifier><index>

<specifier> ::= (<object string>)|<compound symbol>

A *pattern expression* is an expression, which does not contain concretization signs (but generally contains variables). A *workable expression* is an expression, which does not contain variables (but generally contains concretization signs). An *object expression* is an expression, which contains neither concretization signs nor variables.

I.3 Sentences and Programs.

<sentence> ::= #<comment><reversion indicator><left side>
<right side>

<comment> ::= <object string>|<empty>

<reversion indicator> ::= <empty>|(R)

<left side> ::= k<pattern expression> =>

<right side> ::= <expression>

<program> ::= <empty>|<program><sentence>

No sentence can contain variables with identical indexes but different type signs. The right side of a sentence can contain only those variables appearing on its left side. Specifiers in right sides are omitted.

By the *range* of a concretization sign k in an expression we mean the subexpression bounded by this sign and the concretization point $\perp$ paired with it. We call the *leading sign* k in a given expression the leftmost sign k with no other signs k in its range.

2. Syntactical Recognition

2.1 We say that an object expression E_0 can be *syntactically recognized* as a pattern expression E_p, if the variables in E_p can be replaced -- observing the rules listed below -- by such expressions, called their *values*, that E_p becomes identical to E_0. The rules are as follows.

2.1.1 A variable of the form sX, tX or eX, where X is an index, can take as a value any symbol, term and expression, respectively.

2.1.2 A variable of the form s(P)X, where P is an object string, can take as a value any symbol, which enters P. Variables s/SIGN/X and s/COMP/X take as values object signs and com pound symbols, respectively. A variable of the form s$\mathcal{D}X$, where $\mathcal{D}$ is a compound symbol different from those two, is equivalent to a variable s(P)X, where P is the result of concretization of k$\mathcal{D}\perp$.

2.1.3 All entries of the same variable; i.e., those with the same index, must be replaced by the same value.

2.2 If there are several alternative ways of assigning values to the variables, the ambiguity is resolved in one of the following two ways, which will be called recognition *from left to right* and *from right to left*. If recognition from left to right (from right to left) takes place, then of all alternatives the one is chosen in which the leftmost (rightmost) expression variable in E_p takes the shortest value. If this does not resolve ambiguity, the analogous selection is made with respect to the second from the left (right) expression variable etc.

2.3 To recognize a term $kE_0\perp$ as a left side kE_p => means to recognize E_0 as E_p.

3. Refal Machine.

The *Refal machine* is an abstract device which executes algorithms written in Refal. It consists of two potentially infinite stores, which are called the *memory-field* and the *view-field*, and a processor. At every moment in time the memory-field contains a finite sequence of sentences, and the view-field contains a workable expression.

The Refal machine works by steps. Having fulfilled a step, the machine proceeds to execute the next one, provided that the former has not led to a normal or abnormal stop. Exe cution of the step begins with the search for the leading sign k in the view-field. If there is no sign k, the Refal machine comes to a *normal stop*. On finding the leading sign k the Refal machine examines the term which begins with it; it is called *the active term*, and we say that the starting sign k *became active*.

3.1 If the active term is k/BR/(N)$E\perp$, where N and E are some expressions, the machine writes down a new sentence

# k/DG/N => E

into the memory field, putting it before the first sentence. The active term is removed from the view field, and the step is completed.

3.2 If the active term is k/DG/$N\perp$, the Refal machine finds in the memory field the first sentence of the form

# k/DG/*N* => *E*

with the same *N*, removes it from the memory field and substitutes *E* for the active term, thus finishing the step. If there is no such sentence, the active term is merely removed.

3.3 In other cases the Refal machine compares the active term with the consecutive sentences in the memory field, beginning with the first one, searching for an *applicable* sentence, by which we mean such a sentence, that the active term can be recognized as its left side. Recognition is performed from left to right if the reversion indicator is empty, and from right to left if it is (R). Having found the first applicable sentence, the Refal machine copies its right side, replacing the variables by the values they have taken in the process of recognition. The workable expression thus formed is substituted for the active term, and the step is finished. If there is no applicable sentence, an *abnormal stop* occurs.

4. External Functions

In real implementations of Refal, as distinct from the abstract Refal machine described above, one more action is taken at each step before using the sentences: the examination of whether the active term is or is not an *external function* call. By *external* we mean those functions which are not described in Refal. Some symbols must be specified in every implementation as external function determiners. If the active term has the form k*FE*⊥, where *F* is such a determiner, control goes to a program (or whatever) that performs the concretization. It may result in the replacement of the active term by some workable expression, and may produce any effect in the environment. After it is over, the current step is finished and control goes back to the Refal machine.

The functions which provide input-output facilities clearly must be external. In all implementations a function /PR/ is available, which is defined so that when a term k/PR/$E\perp$ becomes active, the expression E is printed and the term is transformed into E. Another function, /P/, prints the argument and deletes the active term.

We do not introduce into the formal description of Refal the concept of number, but in implementations it is possible to code positive integer numbers in a certain range as compound symbols of a special kind. The arithmetic operations on them are performed with the aid of appropriate external functions.

A compound symbol which enters a symbol variable as a specifier may also represent an external function.

5. Representations.

In written and printed representations, variable indexes are lowered. The sign # may be omitted in which case each sentence must begin in a new line.

It is also possible to use *the shorthand notation*, in which Greek letters are introduced as representing combinations of a sign k and a function determiner. Additionally we agree that if a concretization point paired with a k-sign implicit in a Greek letter closes a subexpression it may be omitted (because concretization points closing subexpressions can be unambiguously restored). Therefore, the definition of the function /FIRSYM/ whose value is the first symbol of an expression may take the form:

$$\alpha = k/FIRSYM/$$

$$\alpha s_1 e_2 \Rightarrow s_1$$

$$\alpha (e_1) e_2 \Rightarrow \alpha e_1 e_2$$

$$\alpha \Rightarrow$$

At last we introduce one more facility into the shorthand notation: *upper indexes* can be used without any further definitions. If α is defined as above, then α^a means k/FIRSYMA/ and α^{25} is equivalent to k/FIRSYM25/. An upper index used with an object sign turns it into a compound symbol. So, F^1 is equivalent to /F1/, and R^{+-} to /R+-/.

The representation of the Refal program to input into the computer may depend on implementation. In current implementations (semi-compilers) of Refal the above definition of the function /FIRSYM/ will take this form:

```
FIRSYM S1E2 = S1
       (E1)E2 = k/FIRSYM/E1E2.
```

REFERENCES

1. Turchin, V.F., "A Supercompiler System Based on the Language REFAL", *SIGPLAN Notices* 14(2) (Feb. 1979), pp. 46-54.
2. Turchin, V. F. The Language Refal, the Theory of Compilation and Metasystem Analysis. Technical Report No. 018, Comp. Sci. Dept., Courant Inst. of Math. Sciences, New York, 1980.
3. Ershov , A. P., "On the Essense of Translation," in: Neuhold, E. J., Editor, Formal Description of Programming Concepts, North-Holland Publ. Co. 1978, pp. 391-418.