Courant Computer Science Report #20

February 1980

The Language REFAL—The Theory of Compilation and Metasystem Analysis

Valentin F. Turchin

Courant Institute of
Mathematical Sciences

Computer Science Department

New York University

Report No. NSO-20 prepared under Grant No.
NSF-MCS76-00116 from the National Science Foundation.

COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

A101 ABRAHAMS, P. The PL/I Programming Language, 1979, 151 p.

C66 COCKE, J. & SCHWARTZ, J. Programming Languages and Their Compilers, 1970, 767 p.

D86 DAVIS, M. Computability, 1974, 248 p.

M72 MANACHER, G. ESPL: A Low-Level Language in the Style of Algol, 1971, 496 p.

M81 MULLISH, H. & GOLDSTEIN, M. A SETLB Primer, 1973, 201 p.

S91 SCHWARTZ, J. On Programming: An Interim Report on the SETL Project. Generalities; The SETL Language and Examples of Its Use. 1975, 675 p.

S99 SHAW, P. GYVE — A Programming Language for Protection and Control in a Concurrent Processing Environment, 1978, 668 p.

S100 SHAW, P. " Vol. 2, 1979, 600 p.

W78 WHITEHEAD, E.G., Jr. Combinatorial Algorithms, 1973, 104 p.

COURANT COMPUTER SCIENCE REPORTS

1 WARREN, H. Jr. ASL: A Proposed Variant of SETL, 1973, 326 p.

2 HOBBS, J. R. A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language, 1974, 266 p.

3 TENENBAUM, A. Type Determination for Very High Level Languages, 1974, 171 p.

5 GEWIRTZ, W. Investigations in the Theory of Descriptive Complexity, 1974, 60 p.

6 MARKSTEIN, P. Operating System Specification Using Very High Level Dictions, 1975, 152 p.

7 GRISHMAN, R. (ed.) Directions in Artificial Intelligence: Natural Language Processing, 1975, 107 p.

8 GRISHMAN, R. A Survey of Syntactic Analysis Procedures for Natural Language, 1975, 94 p.

9 WEIMAN, CARL Scene Analysis: A Survey, 1975, 62 p.

10 RUBIN, N. A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design, 1975, 172 p.

11 HOBBS, J.R. & ROSENSCHEIN, S.J. Making Computational Sense of Montague's Intensional Logic, 1977, 41 p.

12 DAVIS, M. & SCHWARTZ, J. Correct-Program Technology/Extensibility of Verifiers, with an Appendix by E. Deak, 1977, 146 p.

13 SEMENIUK, C. Groups with Solvable Word Problems, 1979, 77 p.

14 FABRI, J. Automatic Storage Optimization, 1979, 159 p. ..

15. LIU, S-C. & PAIGE, R. Data Structure Choice/Formal Differentiation. Two Papers on Very High Level Program Optimization, 1979, 658 p.

16 GOLDBERG, A. T. On the Complexity of the Satisfiability Problem, 1979, 85 p.

17 SCHWARTZ, J.T. & SHARIR, M. A Design for Optimizations of the Bitvectoring Class, 1979, 71 p.

18 STOLFO, S. J. Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces, 1979, 168 pp.

19 LOERINC, B. M. Computing Chromatic Polynomials for Special Families of Graphs, 1980, 111 pp.

20 TURCHIN, V. F. The Language Refal--The Theory of Compilation and Metasystem Analysis, 1980, 245 p.

Notes: Available from Department LN. Prices on request.
Reports: Available from Ms. Lenora Greene. Nos. 1,3, 6,7,8,10 available in xerox only..

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, New York 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-20

The Language REFAL--The Theory of Compilation and Metasystem Analysis

Valentin F. Turchin

Report NSO-20 prepared under
Grant No. NSF-MCS76-00116 from
the National Science Foundation

TABLE OF CONTENTS

INTRODUCTION

This book presents a formal system based on the language "Refal" (i.e. REcursive Functions Algorithmic Language). Besides the language itself, and the techniques of programming in it, the system includes a theory of equivalence transformation of algorithms defined in Refal, and an approach to foundations of logic called *metasystem analysis*.

The origins of Refal are in computer science. It was designed as a universal metalanguage for formal definition of algorithmic languages — oriented towards classes of problems, or invented ad hoc for specific problems. At the same time Refal can be regarded as a regular algorithmic language oriented towards symbol manipulation. It is implemented on computers and has been used in this capacity. However, a programming system using Refal as a *meta*language proper, and including a "supercompiler" is still in the project stage.

The aim of this project is to facilitate the creation and implementation of specialized algorithmic languages at low expense, and also to allow computers to perform a great deal of work on optimization of algorithms and even on algorithmization itself, which is now performed manually. We hope to create a programming system, in which the *ad hoc* introduction of a new special language, or a hierarchy of languages, for each large-scale programming problem is just as natural and practicable as is the introduction of an ad hoc hierarchy of procedures when we are programming, say, in ALGOL-60. We hope to create a system, in which the programmer will have to formulate only the definition of his problem, its mathematical model, without bothering about the details of algorithmic efficiency and data structures in the real computer.

To build an extensible hierarchical system of language, a *metalanguage* must be specified which would allow the system's user to define each new language L_n in terms of the languages of the lower levels: L_{n-1} , L_{n-2} , etc. Also, a ground-level language L_0 must be defined, and must be such that all the languages of the hierarchy could ultimately be expressed in it. There are two ways to formally define a new language L_n in terms of a lower-level language L_k: in a translation mode, in which one specifies the manner in which a text in L_n is transformed into a text in L_k; and in an interpretation mode, i.e. specifying the process of execution of a text in L_n in terms of the language L_k. Accordingly, we can distinguish between two kinds of expansible systems. Systems of the first, translation mode, kind have machine (assembler) language as the ground-level language L_0: such systems may be called macrocode systems, and are widespread now. The metalanguage in this case is the language of macrodefinitions. Although very useful, these systems do not unburden the programmer, but only put him in a bettern environment. The system we are designing is of the second kind. Here, new languages are defined in interpretation mode, and L_0 is then a very elementary language which includes only basic operations on symbolic expressions. The description of a language and of an algorithm in that language takes the shape of a "description of the meaning" rather than a final definition of the program to be executed on a computer. But then one needs an algorithm — which we call *supercompiler* — which would translate this multilevel interpretative semantic definition of a problem into an efficient program for a real computer.

An important feature of our project is that the metalanguage M in which new languages are defined, the ground-level language L_0 , and the language inwhich the supercompiler is written, are all the same language, Refal. As shown in [1], this has the crucial advantage that only *one* supercompiler C_p from the metalanguage M into the language of an object machine M_0 is needed for all languages L_n of all levels. To attain this surprising economy, we use a method, whose essence is self-application of C_p. The result: by writing a simple "metasystem-transition formula" and pushing a button one can obtain a program for M_0 which can be either

(1) an efficient, compiled program P_0 which is the translation of a program P written in L_n (if P is given); or

(2) an interpreter for the language L_n , which takes a program P in L_n and input data D and executes P on M_0 in accordance with the interpretive definition of L_n; or

(3) a compiler for the language L_n , which takes a program P in L_n and translates it into an efficient program P_0 for M_0; or

(4) a compiler compiler (if the definition of L_n is not given), which takes the definition of a new language in M and produces a compiler for it.

For the approach we have sketched to be feasible, the following three requirements must be met by the metalanguage M:

(1) It must be universal — not only in the sense that any algorithmic transformation can be described in it, but also in the sense that it must not be aimed at any special system of concepts tied to a particular object language; this makes it possible for one and the same metalanguage to be used with equal success in describing whatever language we may invent, and at all levels of the conceptual hierarchy. In programming terms, the metalanguage must have a broad symbol manipulation orientation.

(2) The metalanguage must be convenient to use; in particular, a text in it must look not like an intricate program, which in some mysterious way performs algorithms written in the language to be described, but rather must be a semantic description of this language, consisting of a set of sentences which define the meaning of its concepts. Thus, the metalanguage must be essentially a *production language*, rather than an instruction/ statement language of more familiar form.

(3) The language must be *minimal* in the sense that the *defining machine* which executes algorithms written in this language must be simple enough for the rules of dealing with algorithms to be formulated effectively. Otherwise there

will be little hope of creating a supercompiler which could perform really deep optimizing transformations of algorithms. However, this requirement may come into conflict with the requirement of convenience. A simple Turing machine or Markov algorithms languages are simple enough to be used for purposes of theory, but certainly impractical for writing complicated algorithms. A language which deals with itself must be neither too sophisticated nor too elementary, a situation reminiscent of maximizing the product of two factors with a given sum. We can summarize the third requirement of our metalanguage in these words: it must rest upon a minimum of facilities, but still remain convenient enough to be used in practice.

The language Refal was born in response to these requirements.

Independently of our work, one of the ideas of the Refal project became quite widespread during the last two or three years, although expressed in somewhat different terms. It is the idea that one should distinguish between *programming* in a *programming* language, and *specifying* your **algorithm in a** *specification* language; and that a good programming system should allow you to specify your problem only, without actually programming it. The concept of a specification language appeared originally in the context of proving correctness of programs, where a specification was intended to give some information *in addition* to a program; later people started to speak of a specification *instead of* a program. Should we use these terms, we could say that Refal is a specification language. But we shall stick to our terms referring to Refal as a universal algorithmic metalanguage, and not only because our project was initiated long before the current trend in terminology. The term "specification language" is not very meaningful. The fact that we are "specifying" something in Refal is not essential; after all, writing a program is also specifying it. It is essential *how* we do it. It is essential that we allow an extensible system of ad hoc languages, leaving only the metalanguage fixed.

In the algorithmic aspect, the term "specification language" is again unfortunate, not to say misleading. Any specification of a problem to which the solution is an algorithm is, in the last analysis, a definition of an algorithm, even if disguised by mathematical notation of the precomputer era. The important difference, as discussed above, is whether you define your algorithm in interpretation mode, without thinking of efficiency of the process, or in translation mode, aiming at an efficient program for a machine. We use the term *compilation* to designate transformation of an interpretation-mode algorithm into a translation-mode (efficient) algorithm. The relation between these fundamental concepts and their formalization is one of the main themes of the present book.

The philosophical background which initiated the work on the Refal project is developed in [2]. In that book the concept of *metasystem transition* is introduced and taken as the basis for an analysis of the evolutionary process. By a metasystem transition we mean a transition from a system S to a metasystem S*, containing a set of S-type subsystems unified as a whole and somehow controlled, produced, modified, etc. Seen in the functional aspect, this transition is a transition from the activity A typical for system S to a *metaactivity* A* exhibited by S*, which is directed in some way onto the activity A: analyzing it, modifying it, etc. In [2], the metasystem transition is shown to be a sort of "quantum of evolution". Accumulation of these quanta produces more and more sophisticated structures, organized as multi-level hierarchies of control. For a system to be self-developing, *consecutive metasystem transitions* must become possible ("the stairway effect" in the terminology of [2]). It is the author's belief that to make essential progress in programming systems and artificial intelligence, one must formalize and harness the concept of metasystem transition. In the present book we are making first steps in this direction.

A few comments on the contents of the book, together with some historical and bibliographical references follows.

In the first chapter the language Refal is introduced. The second chapter presents the techniques of using Refal as an algorithmic language expecting the program in Refal to be executed in the interpretation mode. We do not say much about implementation of Refal, outlining only the general principles. A detailed description of the interpretative implementation of Refal may be found in [1].

Initially Refal was called *the metaalgorithmic language* [3,4] and had some features which made its efficient implementation difficult. Subsequently it became clear that the decisive role in this language is played by the notion of recursive function, and the methods of programming in it had been worked out [5,6]. The language was simplified and received its present name.

In the creation of Refal the ideas embodied in LISP [7] and COMIT [8] were used; A. A. Markov's work on normal algorithms [9] was an important source of ideas. We must also mention papers by E. Dijkstra [10,11] and A. Van Wijngaarden [12]. Refal shares some ideas with SNOBOL [13], and a striking resemblance to CONVERT [14] can be seen, though in 1965-66, when Refal was designed, the author was not acquainted with either of these languages.

Save for the initial period, there has been no influence of other approaches on the Refal approach which we could mention as appreciable. To be sure, one could find parallels between our work and work done by other authors, but it would have required a special effort which the author did not undertake when preparing this book.

The first Refal interpreter efficient enough for practical purposes was put into operation in 1968, in Moscow, on the computer BESM-6 [15,16]. An interpreter with automatic access to external memory was developed in Leningrad [17]. In 1969 a new method of implementation of Refal was worked ou [18,19], which allowed, in particular, a greater part of the implementation work to be accomplished in a machine-independent form. At that time this method was called compilation, but in fact it should be more precisely called *semicompilation*. There now exist Refal semicompilers for the most popular Soviet computers (ES EVM, BESM-6, M-220, Minsk-32). An extensive exposition of programming techniques in Refal was published in 1971 as a series of preprints of the Institute for Applied Mathematics of the Academy of Sciences of the USSR [20]. As a programming language Refal has been used for writing translators, algebraic maniuulation and theorem proving (see, e.g. [21-23]). The efficiency of the use of Refal in semi-interpretive implementation is comparable to that of LISP or SNOBOL. Debugging in Refal is, in the view of the author, easier than in any of the languages he knows.

In Chapter 3 we introduce basic equivalence transformations of Refal programs. The most important transformation, called *driving* is the "*concretization*" (evaluation) of a function call with only a *partially defined* argument, i.e. an argument containing free variables or not yet evaluated function calls. The notion of driving appeared first in [20]; more systematically the rules of equivalence transformations were formulated in [24] and [25].

Chapter 4 presents *the theory of compilation*. Its main idea is to consider the graph of generalized states (*configurations*) of the Refal-machine, and to reduce the graph to a certain normal form, using driving. In programming terms this procedure can be defined as executing *at compile time* all the evaluations which can be done, and compiling a new graph out of those operations which have been left for the *run time*, as strongly dependent on input data. The normal form depends on the set of configurations which are declared *basic*. This gives us a means to formalize the intuitive notion of the interpretation-compilation axis, and to control the process of compilation. Besides driving we use empirical generalization with subsequent proof by mathematical induction. The notion of a *perfect* graph of states is introduced, which also serves to direct the compilation process.

The first examples of using driving for optimization were published in 1971 (see [20]). The main ideas and results of the theory of compilation were formulated in 1973. Unfortunately, in the years 1974-1977 the author could not publish his work in the USSR because of political circumstances. (In 1974 I was expelled from my job and blacklisted as an active participant in the Human Rights Movement in the USSR. The book [1] was published *anonymously* after my emigration in 1977, and it was only possible to smuggle into it several pages on the theory of compilation. But it was not allowed to use the term "the theory of compilation", nor to mention that the piece was a part of a larger work.)

In Chapter 5 we introduce metasystem transition into our formal system. This is done through *metaderivative* and *metaintegral* functions, which are used in *MST-formulas.* Basically, this is a self-application of the algorithm of equivalence transformation. By this we model a feature of human thinking, which is crucial for creativity: the ability to transfer attention from the use of an instrument (e.g., an equivalence transformation) to the analysis of its use and making a new instrument to improve the existing instrument. In our system this transfer can be repeated indefinitely, just by writing an MST-formula. We show that by including the metasystem transition into our formal system we *do* expand the range of possible equivalence transformation algorithms; e.g., writing an MST-formula for an algorithm which cannot prove the commutativity of addition in formal arithmetic, we receive an algorithm which proves it. Our approach is not based on traditional axiomatic logic, but on direct modeling of the three main aspects of human thinking:

(1) concretization (computation), including driving;

(2) generalization (empirical induction) with subsequent proof by mathematical induction;

(3) metasystem transition.

We call this approach *the metasystem analysis* (see Sec. 5.7).

Application of the principle of metasystem transition to practical needs of the supercompiler system leads to the result which has already been mentioned above: having one supercompiler for an object machine (computer), we are able to produce automatically compilers and other system programs for all languages defined in Refal in interpretation mode. This is the only result of Chpater 5 which was published before (in [1]). It is also mentioned in A. P. Ershov's paper [26].

A brief account in English of the project of the supercompiler system based on the language Refal was recently published in the *SIGPLAN Notices* [27].

ACKNOWLEDGEMENTS

Many people worked with the author of this book on different aspects of the language Refal and its implementation. I want to use this occasion to thank all of them. It is especially pleasant to express my heartfelt gratitude to the most permanent participants of the Refal seminar in Moscow, whose contribution is impossible to overestimate: Andrei Klimov, Arkadi Klimov, Nicolai Kondratiev, Sergei Romanenko, Yelena Travkina, and other participants of the seminar.

I am most grateful to those scientists in America and elsewhere who spoke out on my behalf when I was jobless and on the verge of arrest in the Soviet Union because of my Human Rights activities. The Association of Computing Machinery passed a resolution on my situation. The Federation of American Scientists, in the person of Jeremy Stone, Director, many times appealed to the Soviet authorities on my behalf, and induced American scientists to take analogous actions. Dr. John Backus

circulated a petition on behalf of Shcharansky and myself, which was signed by 65 participants of the Working Conference on Formal Description of Programming Concepts, St. Andrews, Canada, in August 1977, and sent it to Moscow together with his personal letter. Prof. Lipman Bers sent me an invitation to visit Columbia University, and Mrs. Margaret Freeman of MIT sent me a formal guest invitation. I believe that all of these actions greatly contributed to the fact that I was not, in the end, arrested (although still forced to emigrate).

I thank most sincerely Professors Jack Schwartz and Robert Dewar for my 20 months stay at the Courant Institute, which gave me the opportunity to write this book. My special thanks are to Prof. Sal Stolfo for reading the manuscript and correcting my English were it was possible. I realize though, that one cannot radically improve the language by editing, and ask the reader's forbearance.

CHAPTER 1. DESCRIPTION OF THE LANGUAGE

1.1 Informal Description

To sum up the requirements set forth in the Introduction, the language which we intend to design must be: (1) universal, (2) convenient for semantic description of different languages, and (3) minimal. Our purpose now is to present the main features of Refal as derived from these requirements. Since it is not at all evident that recursive functions have something to do with the language we design, we shall start with calling the language we seek *the language* M (Metalanguage). The concept of recursive function will appear in due course as a result of reasoning, and this will once more demonstrate its profundity and importance.

Problem oriented languages are convenient when they reflect concepts specific for a specific field. To be convenient our language must model some very general features of human thinking — or, to be more precise — its manifestation in linguistic activity. This activity consists in manipulating linguistic objects to which certain "meanings" are prescribed. Linguistic objects are composed of signs, but not in an entirely arbitrary fashion because they have an inner structure which reflects the syntax of the language. In fact, linguistic objects are produced from parsing, and this feature being common to all languages, must be taken into account in the language M. Acting on the principle of minimality we shall assume the simplest scheme modeling the syntax of natural and artificial languages. The elementary syntactic unit of the language M will be called the *sign*. The complete set of signs is supposed to be finite, though in accordance with the metasystem nature of the language it is not exactly specified. The next syntactical level is formed by *symbols*.

While signs are analogous to letters in natural language, symbols model elementary semantic units — morphs: word roots, prefixes, etc. A symbol may be *simple* or *compound*; the former is expressed by a sign, the latter — by a sequence of signs bounded by slashes, e.g., /BEGIN/. When an algorithm is executed, each symbol is treated as a whole and cannot be subdivided; neither can a new symbol be formed. The purpose of this syntactical level is to provide a potentially infinite set of indivisible units.

To build symbol structures we introduce the most common means — parentheses. We call an *expression* any string of symbols and parentheses, obeying the usual rules of the parenthesis (bracket) syntax. Strings unbalanced in parentheses are not expressions and are not allowed. The expression is the most general object of manipulation in the language M. It can be paralleled with a word or a group of words in a natural language.

Having specified the object, we proceed to specify the actions on them. An inherent feature of all developed languages is the presence of a hierarchy of concepts. Consider a language object, which has some meaning. What does it mean to *understand* its meaning? It is to know how in any given circumstances to *concretize* the object — that is to express the meaning through concepts which take lower places in the hierarchy, and thus to replace it by the language objects which fix these lower rank concepts. With a natural language, this process comes to an end when the relations are established between a language object and the sensual world; in the case of a formal language we come to the concepts, defined as primitive. We shall take conceretization as the unit of action in the language M. From a formal point of view it is, of course, no more than substitution of one expression for another, and it is up to the user to ensure that these actions have in fact the quality of concretization.

A pair of signs is used to delimit an expression to be concretized: the *concretization sign* k, which precedes the expression, and the *concretization point* $\perp$, which follows it. They obey the rules of bracket pairing, so they are also called the *concretization brackets*. In contrast, parentheses (round brackets), which give a structure to the object, are called the *structure brackets*. A pair of concretization brackets may enclose other concretization brackets; as the language M is algorithmic, we must specify which concretization is to be performed first. Before an expression gets to be concretized it must contain no inner signs of concretization; parallel concretizations will be performed from left to right. In the following example:

$$\overset{3}{k}\ A\ (\ B\ C\ )\ \overset{1}{k}\ D \perp E\ \overset{2}{k}\ F\ (\ )\ G\ H \perp \perp$$

the figures show the order in which the concretization must proceed.

In accordance with one of the requirements, all information that we want to convey in our language must be expressed in *sentences*, which are essentially the rules of concretization. They constitute an analogue of sentences in natural language. Each sentence contains a *left side* and a *right side*. The left side begins with the concretization sign k and ends with the substitution sign ⇒, which simultaneously stands for the concretization point paired with the initial k and separates the left side from the right. The sentences are separated by the sign # placed in the beginning of each. Between the # sign and the initial k there may be a *comment*. This is an example of a sentence, which describes a simple abbreviation:

1 k ACM ⇒ Association for Computing Machinery

Whenever a combination k ACM $\perp$ may enter, it will ultimately be replaced through the application of this sentence by

Association for Computing Machinery

A sequence of sentences describes an algorithm. It is performed as follows. At each step we try to find an applicable sentence, beginning the search from the first; when such a sentence is found, it is applied, and the next step is executed. This procedure is reminiscent of Markov's normal algorithms, the essential difference is that symbol strings in Markov's language are unstructured, for this language, like other "theoretical" algorithmic languages, models only the very fact of man's formal actions, whereas our language uses structured strings (expressions), and models man's formal actions in the framework of a hierarchy of concepts.

Yet the structurization of language objects alone is not sufficient to get a language powerful enough for practical purposes. We need *free variables* to make our sentences really expressive. In accordance with the syntax of the language M, we introduce three types of variable: those which may take as a value a symbol, a term or an expression. A free variable can be represented by a pair of signs, of which the first is s, t or e, and shows its type, and the second serves as an identifier. For convenience, we will write the identifier in the index position: s_1 , s_2 , s_a , t_1 , e_x , etc. These variables may take as a value any object of the corresponding type.[*] Additionally, we introduce a possibility of specifying a class of allowed values of a symbol variable; details will be seen from the formal description.

As an example of the use of free variables we define the concept of the first symbol of an expression. To define a concept means to make the corresponding concretization possible and produce the needed result. We may wish, for example, that the concretization of

k FIRST SYMBOL OF PROGRAM ⊥

should produce the letter P. To ensure it we write a sentence:

$$\# \; k \text{ FIRST SYMBOL OF } s_1 \, e_2 \Rightarrow s_1$$

[*] We shall call them s-, t-, and e-variables for short.

It will be used in the following way. First we must find out whether the sentence is applicable for the above concretization. For this purpose we try to *syntactically recognize* the expression under concretization as the left side of the sentence. The recognition is possible if the free variables in the left side can be replaced by such values that the left side becomes identical with the expression under concretization. In our case this can be achieved in a unique way by assigning the value P to s_1 and the value ROGRAM to e_2 . So, recognition is possible, and we apply the sentence, that is substitute its right side (in which the variables are replaced by their values) for the expression under concretization.

Now we come to recursive functions. Theoretically, at each step of concretization we look through the set of sentences from its beginning, analyzing the applicability of each sentence. When the number of sentences grows, this becomes cumbersome. We clearly need a way to break down all the concretization rules into separate parts pertaining to different concepts. Such a way is suggested by the above example. If the left side of a sentence begins with a concrete symbol or a string of symbols, but not with a variable, it will be applicable only to such expressions which begin with the same string. As we can use compound symbols, one symbol is quite enough to classify sentences, so we shall agree that every sentence will have the left side beginning with at least one symbol, which will be called *the determiner* of the sentence. Now in an expression to be concretized we can single out the first symbol and ignore all sentences with determiners different from this symbol. Sentences break down into groups with the same determiner, groups with different determiners being commutable with each other. In the spirit of this agreement it is preferable to rewrite the last sentence in this way:

# 2.1 k /FIRSYM/ $s_1 e_2 \rightarrow s_1$

and to use it accordingly.

What we have in fact introduced by our agreement is the concept of a function. Functions are identified by determiners and defined by groups of sentences with the same determiner. A usual function designation $\mathcal{F}(\mathcal{E})$ turns into k $\mathcal{F}$ $\mathcal{E}$ $\perp$, where the concretization sign explicitly shows that the function must be evaluated and its value substituted. As we do not restrict in any way the right sides of sentences, functions are generally recursive. The description of an algorithm takes the form of a recursive function definition.

We can illustrate recursion by extending the definition of the function /FIRSYM/. Indeed, this function is underdefined: if a machine, executing algorithms written in Refal (we call it *the Refal-machine*) happens to concretize an expression which begins with a bracket, it will not find any applicable sentence (a parenthesis is not a symbol!) and will come to an *abnormal stop*. The empty expression will bring about the same result. A natural way to extend the definition is to add two sentences:

# 2.2 k /FIRSYM/ $(e_1)e_2 \Rightarrow$ k/FIRSYM/e_1e_2 $\perp$

# 2.3 k /FIRSYM/ $\Rightarrow$

The language Refal is known in three versions: *strict*, *basic*, and *extended*. We have already exhausted the facilities present in the strict version; in fact, we have exceeded them, for it incorporates certain restrictions on the left side of a sentence as we shall see in Sec. 3.1. Basic Refal will be formally described in the next section. It includes a feature, which allows assignment of expression values to names. When an expression of the form k/BR/($\mathcal{N}$)$\mathcal{E}$ $\perp$ is concretized (where $\mathcal{N}$ and $\mathcal{E}$ are expressions [*]) it disappears, but $\mathcal{E}$ gets "buried" under the name $\mathcal{N}$. It can be "dug out" by writing the expression k /DG/ $\mathcal{N}$ $\perp$, the concretization of which turns it into $\mathcal{E}$.

[*] We shall generally use capital script
($\mathcal{A}$ $\mathcal{B}$ $\mathcal{C}$ $\mathcal{D}$ $\mathcal{E}$ $\mathcal{F}$ $\mathcal{G}$ $\mathcal{H}$ $\mathcal{I}$ $\mathcal{J}$ $\mathcal{K}$ $\mathcal{L}$ $\mathcal{M}$ $\mathcal{N}$ $\mathcal{O}$ $\mathcal{P}$ $\mathcal{Q}$ $\mathcal{R}$ $\mathcal{S}$ $\mathcal{T}$ $\mathcal{U}$ $\mathcal{V}$ $\mathcal{W}$ $\mathcal{X}$ $\mathcal{Y}$ $\mathcal{Z}$)
for metasymbols which denote Refal objects.

You cannot dig out an expression twice, which you have buried only once. The reader may have surmised that this avoids copying expressions when implementing such assignments.

In Extended Refal it is possible to introduce new sentences into an algorithm during its execution, and to use free variables of arbitrary syntactical types described through appropriate recursive functions. We have no place here to describe the details on this subject.

1.2 Formal Description of Basic Refal

I. Syntax

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

I.I 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 conretization point ⌊ 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 s*X*, t*X* or e*X*, 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 compound symbols, respectively. A variable of the form sDX, where 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 $kD\perp$.

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 $kE_p\Rightarrow$ 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. Execution 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 starging sign k *became active*.

3.1. If the **active** term is k/BR/(*N*)*E*⊥, 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*⊥, 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 (e.g. for ES EVM up to $2^{31}-1$) 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.

1.3 Representations and Metacodes

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. If the determiner in a sentence repeats that of the preceding sentence, it may be omitted together with the iniitial sign k. Thus the above definition of the function /FIRSYM/ could be written as follows:

$$\begin{aligned} &k/FIRSYM/\ s_1 e_2 \Rightarrow s_1 \\ &\qquad\qquad (e_1)e_2 \Rightarrow k/FIRSYM/e_1 e_2\perp \\ &\qquad\qquad \Rightarrow \end{aligned}$$

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 /FIRSYM/ may also take the form:

$$\begin{aligned} \alpha &= k/FIRSYM/ \\ \alpha s_1 e_2 &\Rightarrow s_1 \\ \alpha(e_1)e_2 &\Rightarrow \alpha e_1 e_2 \\ \alpha &\Rightarrow \end{aligned}$$

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 /Fl/, and R^{+-} to /R+-/.

To write in Refal algorithms dealing with algorithms written in Refal itself we have to represent sentences by object expressions, therefore, we need a special code for this purpose. It will be called *metacode* A. We further need a code to input Refal programs into a computer, which will be called *metacode* B. It is convenient to represent object signs in Refal by bytes in a computer, and it is convenient to treat each byte as an object sign. Since in I/O operations we are dealing, after all, with sequences of bytes, Refal sentences, and all possible Refal objects, for that matter, must be represented in metacode B by *object strings* (strings of object signs). In metacode A, Refal objects will be represented by *object expressions* — for there is no need to destroy their tree structure.

So, metacode A is a mapping of the set of all Refal objects (that is programs and expressions) on the set of all object expressions. Metacode transformation will be designated by adding an asterisk as a superscript to the designation of a Refal object. If Z is a Refal object, Z^* is its metacode-A transformation, Z^{**} its double transformation, etc. Naturally, the metacode transformation and the reverse must be unique, but there is no need to require that each object expression could be interpreted as the metacode of some Refal object. It would be convenient if the metacode of an object expression were always identical to the expression itself, but this is, obviously, impossible because of the required uniqueness of the inverse transformation. (Indeed, let $\mathcal{E}$ be an expression, which is not an object expression. Then $\mathcal{E}^*$ is an object expression. If its metacode transformation $\mathcal{E}^{**}$ must be identical to $\mathcal{E}^*$, then the inverse metacode transformation, when applied to $\mathcal{E}^*$, must give both $\mathcal{E}$ and $\mathcal{E}^*$, which do not coincide by the definition of $\mathcal{E}$.) Nevertheless, it is desirable to define metacode A in such a way that the subset of those object expressions $\mathcal{E}_0$ for which $\mathcal{E}_0^* \neq \mathcal{E}_0$ is minimized.

We define a metacode A by the following rules.

- The metacode of a sequence of objects is the sequence of the metacodes of these objects. The metacode of $(\mathcal{E})$, where $\mathcal{E}$ is an expression, in $(\mathcal{E}^*)$. This rule applies also to specifiers.
- The asterisk * is a special symbol. Its metacode is *V. All the other symbols are transformed by the metacode into themselves.
- A variable type sign $\mathcal{V}$ is transformed into $*\mathcal{V}$. For example, the metacode of s_x is *SX. *Restriction:* The asterisk cannot be used as a variable index.
- An expression $k\mathcal{E}\perp$ is transformed into $*K(\mathcal{E}^*)$.
- A sentence with the left side $\mathcal{L}$ and the right side $\mathcal{R}$ is transformed into $*((\mathcal{L}^*) = \mathcal{R}^*)$, if the reversion indicator is <empty>, and $*(R(\mathcal{L}^*) = \mathcal{R}^*)$, if it is (R). The comments are ignored, which may be regarded as a restriction on the uniqueness of the inverse transformation.

As an example consider the following program:

$$\begin{array}{lll}
k/RPM/e_1{+}e_2 & \Rightarrow & k/RPM1/e_1\perp - k/RPM/e_2\perp \\
k/RPM/e_1 & \Rightarrow & k/RPM1/e_1\perp \\
k/RPM1/e_1(e_2)e_3 & \Rightarrow & e_1(k/RPM/e_2\perp)\ k/RPM1/e_3\perp \\
k/RPM1/e_1 & \Rightarrow & e_1
\end{array}$$

which describes a function, replacing the symbol + by the symbol - on all levels of parenthesis structure. In the metacode A it will become the following expression:

```
*((/RPM/ *E1 + *E2) = *K(/RPM1/*E1) - *K(/RPM/*E2))
*((/RPM/ *E1)       = *K(/RPM1/ *E1))
*((/RPM1/*E1(*E2)*E3)= *E1(*K(/RPM/ *E2)) *K(/RPM1/*E3))
*((/RPM1/*E1)       = *E1)
```

We shall not describe metacode B here (it may vary with implementation), we will only give an illustration. This is how the above program will appear on the programming form:

```
RPM  E1 '+' E2 = K/RPM1/E1. '-' K/RPM/E2.
     E1 = K/RPM1/E1.
RPM1 E1(E2)E3 = E1 (K/RPM/E2.) K/RPM1/E3.
     E1 = E1
```

CHAPTER 2. INTERPRETIVE IMPLEMENTATION AND PROGRAMMING

2.1 Principles of Interpretive Implementation

In Chapter 4 we shall outline the project of a supercompiler system which uses Refal as the metalanguage to define programming languages, and produces compilers for these languages. But to start using Refal and to initialize a bootstrapping process, one needs an interpreter for Refal. The Refal machine as defined above, is such an interpreter. However, if it is implemented on a computer *,literally*, the efficiency will be so poor that it will be impossible to use Refal as a programming language for serious problems. To make this possible, we set forth the following five requirements of the implementation of the Refal machine:

R1. When a parenthesis or a concretization bracket has been located in the view-field it must be possible to locate the paired bracket immediately, without scanning the enclosed expression.

R2. If a variable enters the right side of a sentence in the same number as, or in a smaller number than it enters the left side, then it must be possible to fulfill substitution of the right side without actual copying or scanning of the values of this variable. In other words, subexpressions in the view-field must be transposable without rewriting.

R3. It must be possible to locate the leading sign k without actually scanning the view-field, which is implied in the formal description of the language.

R4. The time needed to bury or to dig out an expression in the view-field must be independent of its length.

R5. Having found a determiner, it must be possible to locate the corresponding group of sentences without scanning the memory-field.

In existing implementations these requirements are met by organizing the view-field as a symbol list structure and maintaining a push-down store for the addresses of the conretization signs present in the view-field. With such an implementation, Refal becomes a practical programming language, which can be used with the same order of magnitude efficiency as LISP or SNOBOL.

2.2 The Projecting Algorithm. Open and Closed e-Variables.

Requirement 1 concerning brackets is very important for syntactical recognition. In the formal description this concept was introduced from the point of view of its final result only. But to understand the precise algorithmic meaning of what is written in Refal, it is necessary to take into account the actual process of syntactical recognition, the algorithm that is used to recognize an object expression E_0 as a pattern expression E_p. This algorithm can be more conveniently described from the opposite side — as an algorithm of *mapping* or *projecting* E_p on E_0. We proceed to do so.

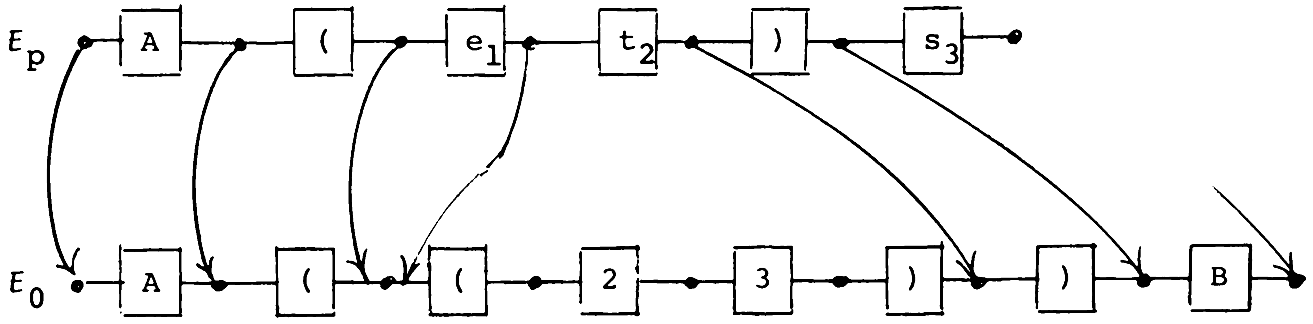
Entries of symbols, brackets and variables will be called *elements* of expressions. Gaps between elements will be called *knots* (see p. 17). The following general rules must be observed at every stage of mapping.

G1. If a knot K_1 is positioned in E_p to the left of a knot K_2 , then its projection P_1 in E_0 cannot be positioned to the right of the projection P_2 of the knot K_2.

G2. Projections of parentheses and symbols must be identical to themselves.

G3. Projections of variables must meet requirements on their values, in particular, different entries of the same variable must have identical projections.

It is assumed that at the moment when syntactical recognition begins, the bounding knots of E_p are projected on the bounding knots of E_0. The mapping algorithm is described by six rules (P1-P6) listed below. They are meant for the left-to-right case, the algorithm for the reversed direction being analogous. At every stage of projecting, the rules P1-P4 determine the element to be projected next; thus each element gets a *projecting number*.



The projection of the pattern expression

$$\mathcal{E}_p = A \; (\; e_1 \; t_2 \;) \; s_3$$

on the object expression

$$\mathcal{E}_0 = A(\; (\; 2 \; 3 \;) \;) \; B$$

P1. After a parenthesis is projected, its paired parenthesis bracket is projected immediately.

P2. If as a result of previous steps both ends (boundary knots) of an e-variable turn out to be projected, this variable is projected. Such entries are called *closed* e-variables. If there are two of them, they are projected from left to right.

P3. An entry of a variable which already has a value is called a *repeated* entry. Parentheses, symbols, s-variables, t-variables and repeated entries of all variables in E_p are called *rigid* elements. If P1 and P2 are not applicable and there are some rigid elements with at least one end projected, the leftmost of them is chosen. If it is possible to project it without contradicting rules G1-G3, then it is projected, and the process goes on. Otherwise a *deadlock situation* is stated.

P4. If P1, P2 and P3 are not applicable and there are some e-variables with the left end projected, the leftmost is chosen. It is called an *open* e-variable. Initially it gets an empty projection, i.e. its right end is projected on the same knot as the left. Other values may be assigned to open variables through *lengthening* (see P6).

P5. If all the elements of E_p are projected, the syntactical recognition is successfully fulfilled.

P6. In a deadlock situation the process comes back to the last (i.e. with the maximum projecting number) open variable, and its value is *lengthened*, which means that the projection of the right end of the variable is moved in E_0 one term to the right. Thereafter the process is resumed. If the variable cannot be lengthened because of the rules G1-G3, the preceding open variable is lengthened. If there is no open variable which could be lengthened, the recognition of E_0 as E_p is impossible.

Examples of Projecting.

On page 17, the variable e_1 is closed. Consider another example (the figures over elements are their projecting numbers):

$$\overset{1}{(}\quad \overset{6}{e_1}\quad \overset{7}{+}\quad \overset{8}{e_2}\quad \overset{2}{)}\quad \overset{9}{e_3}\quad \overset{10}{+}\quad \overset{11}{e_4}\quad \overset{4}{(}\quad \overset{5}{e_5}\quad \overset{3}{)}$$

Here e_1 and e_3 are open variables, e_5 , e_2 and e_4 are closed. It is easy to understand that if several main (not repeated) entries of e-variables are present on the top bracket-structure level of a given expression, the rightmost of them will be closed and the others open. This rule is also applicable to every subexpression enclosed in parentheses.

In this example:

$$\begin{array}{ccccccccccc} 9 & 10 & 11 & 2 & 3 & 5 & 6 & 4 & 8 & 7 & 1 \\ e_1 & + & e_2 & (& (& * & * &) & e_1 & t_3 &) \end{array}$$

it may seem at first glance that the first entry of e_1 is open. In fact is is repeated.

In the rest of this chapter we shall describe the primary methods of interpretive programming in Refal, i.e. prcgramming for an interpreter obeying principles R1-R5.

2.3 Function Formats.

Suppose we want to define in Refal a function, which in a given expression on the top level of bracket structure, removes all repeated blanks, that is replaces each group of adjacent blanks by a single blank. How is the problem approached?

Let us denote the required function by ϕ(k-sign included!); a blank will be represented by the sign $\sqcup$. As there must be no pair of adjacent blanks in the result, we can define the concretization as a recursive removal of one blank of every such pair. This leads to the following two sentences:

$$\phi\ e_1 \sqcup \sqcup e_2 \Rightarrow \phi\ e_1 \sqcup e_2$$

$$\phi\ e_1 \Rightarrow e_1$$

The variable e_1 in the first sentence is open. Initially it takes an empty value and then is lengthened until the first combination $\sqcup\sqcup$ (if any) is found. The variable e_2 is closed, therefore the remaining part of the argument is not scanned, and by applying the sentence one blank gets eliminated. During the next step the projection of e_1 must be scanned again in search of a pair of blanks, and this is clearly useless, because it cannot contain any. So our algorithm is not efficient. We can amend it by taking e_1 out of the conretization range in the right side. Now it takes the form:

$$\phi\ e_1 \sqcup \sqcup\ e_2 \Rightarrow e_1\ \phi \sqcup e_2$$

$$\phi\ e_1 \Rightarrow e_1$$

According to R2 the interchange of ϕ and e_1 takes effort which is independent of the length of e_1. According to R3 the conretization sign in ϕ is located immediately. The part of the argument which follows $\sqcup \sqcup$ is not, as mentioned above, scanned, and when we fail to find this combination, the second sentence is applied without further examination of the argument. Thus no unnecessary actions are implied here.

In the last example the concretization sign was used as a pointer in scanning an expression. This was possible because the scanned part of the expression did not participate in concretization, and therefore it could be taken out. When it is not, we retain the expression in the concretization range and use parentheses as delimieters or pointers. As an example, consider a correction function γ, which in a given string of symbols deletes a symbol if it is followed by the negation sign $\neg$, and if there are several negation signs, deletes the corresponding number of preceding symbols. We can describe γ in a very simple fashion by the following two sentences:

$$\gamma\ e_1 s_a \neg\ e_2 \Rightarrow \gamma\ e_1\ e_2$$

$$\gamma\ e_1 \Rightarrow e_1$$

But this algorithm is not efficient. To construct an efficient algorithm we introduce an auxiliary function γ^1 with the format $\gamma^1(E_1)E_2\bot$, where E_1 is the examined part and E_2 is yet unexamined:

$$\gamma e_1 \Rightarrow \gamma^1(\ )\ e_1$$
$$\gamma^1(e_1 s_a)\ \neg\ e_2 \Rightarrow \gamma^1(e_1)e_2$$
$$\gamma^1(e_1)e_2 s_a\ \neg\ e_3 \Rightarrow \gamma^1(e_1 e_2)e_3$$
$$\gamma^1(e_1)e_2 \Rightarrow e_1 e_2$$

A pair of structure brackets (parentheses) introduced to isolate a part of the argument, thus avoinding unnecessary scans (open variables), will be called a *pouch*. The argument of a function may have any structure with respect to parentheses; this structure will be called its *format*. The notion of the number of arguments, so important for numerical functions, becomes pointless for Refal functions, because any number of expressions, say three expressions E_1 , E_2 , and E_3 , can be brought together into a structure, from which they are easily extractable, for instance $(E_1)(E_2)E_3$, or $((E_1)E_2)E_3$, or $(E_1)(E_2)(E_3)$. Moreover, whatever way we choose to write down a sequence of arguments it will be natural and convenient for the Refal programmer to regard it as *one* composite argument. Accordingly, we will always consider formally that a function in Refal has just one argument, but when its format is explicitly specified, the subexpressions of the argument may also be called arguments — provided the meaning is clear.

Let us give an example of a function with a sophisticated argument. Suppose we are to compare two expressions and compile the list of those terms, which have the same serial number in both expressions and are identical. Both original expressions must be reatined and separated by parentheses in the final result, and the list compiled must be added at the end, also separated by parentheses (cf. Sec. 25.). The format of the function which does this work (we assign determiner α) will be

$$\alpha(\ (E_1)\ E_2)\ (\ (E_3)\ E_4)\ E_5 \perp ,$$

where E_1 and E_2 — are the scanned and yet unscanned parts of the first expression, E_3 and E_4 are the same for the second expression, and E_5 is the list of terms compiled to date. The definition of the function is

$$\alpha((e_1)t_ae_2)((e_3)t_ae_4)e_5 \Rightarrow \alpha((e_1t_a)e_2)((e_3t_a)e_4)e_5t_a$$

$$\alpha((e_1)t_ae_2)((e_3)t_be_4)e_5 \Rightarrow \alpha((e_1t_a)e_2)((e_3t_b)e_4)e_5$$

$$\alpha((e_1)e_2)((e_3)e_4)e_5 \Rightarrow (e_1e_2)(e_3e_4)(e_5)$$

2.4 Scans of Different Orders

The number of open e-variables in the left side of a sentence may be called *the order* of the scan implied in the sentence. A scan of the n-th order requires, generally, a number of elementary operations proportional to N^n, where N is the number of terms in the expression under concretization.

This is a simple example of the second order scan. Suppose we need to find two identical terms in an expression. This goal can be achieved by using a sentence with the left side:

$$\alpha\ e_1\ t_x\ e_2\ t_x\ e_3 \Rightarrow$$

We have here two open variables: e_1 and e_2 , therefore the power of the implied scan is 2. Let us examine the performance of the Refal machine when it applies this sentence. First, the variable e_1 will be assigned the empty value, and t_x will become the first term of the object expression. Then e_2 will take on an empty value. If the second term of the object expression is not equal to the first, the variable e_2 will be lengthened in a search for a term, which would be identical to the first. On coming to the end of the object expression, the Refal machine will lengthen the variable e_1 , i.e. will choose as t_x the second term of the object expression etc. Thus, the Refal machine will perform those and only those actions which are necessary by the essence of the algorithm, therefore there is no loss of efficiency concealed in our program.

Consider, however, a sentence with the left side

$$\alpha\ e_1\ A\ e_x\ Z\ e_2 \Rightarrow$$

It purports to discover the first from the left expression that begins with A and ends with Z. Again this is a scan of the second power: If the expression sought for actually is present in the argument, no unnecessary actions will be performed during the search and the scan will actually be of the first power). But suppose the argument does not contain symbol Z at all. This fact requires only N steps for its discovery (N is the number of the terms). Meanwhile, the Refal machine will unnecessarily lengthen the variable e_1 and perform, in the general case, const.$\cdot N^2$ steps before coming to the conclusion that recognition is impossible: a loss of efficiency. Surely, a sophisticated interpreter (semicompiler) might spot this in the preprocessing (compilation) and introduce corrections, but this must be considered as optimization, which should not be expected from each implementation of Refal. To guarantee efficiency we must redefine our function so as to eliminate unnecessary scans. First, we spot a symbol A by the sentence

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

where β is an auxiliary function, and then we search for Z by using the following left side in the definition of β:

$$\beta(e_1\ A)\ e_x\ Z\ e_2 \Rightarrow$$

(Recall that variables in Refal are local to sentences, so that the same indexes in different sentences, though convenient as a menomonic, should not cause confusion.)

So, we received two sentences with the first power scan instead of one sentences with the second order scan. Generally, it is safe to use first-order scans, because the implications are easily seen, but some caution is needed with higher order scans in order to avoid inefficiency. Redefinintion to eliminate unnecessary scans is always possible and fairly obvious. In particular, it is possible to program in such a way that there will be no open variables at all. In this case, all the scans present in the algorithm will be expressed by functional recursion.

2.5 Reproduction of Variables. Branching and Loops.

If a free variable enters the right side of a sentence more times than it enters the left side, we will say that this variable *is reproduced*. In this case the Refal interpreter has to make one or more copies of the value of this variable when applying the sentence. This must be taken into account in programming, because unnecessary reproduction of variables may lead to dramatic losses in efficiency. When there is no reproduction of variables the interpreter will just rearrange the contents of the view-field, inserting and removing only those elements which are explicitly indicated in the sentence as constants (i.e. symbols and brackets). If there is reproduction of variables, the interpreter will have to make copies of some parts of the view-field which may be very extensive. Thus, of the following two similar sentences:

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

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

the second may require one hundred times more time to perform than the first. Therefore, if a variable may have a long value, it should not be reproduced unless it is needed by the essence of the algorithm.

This consideration has a direct bearing on branching and exchange of arguments and values between functions in Refal. In other languages branches are usually defined through conditional expressions which make use of predicates (Boolean functions). In Refal it would not be difficult to define the semantics of conditional expression so that it could be used in the usual form:

C.1 $\quad k/IF/(T)/THEN/(e_1)/ELSE/e_2 \Rightarrow e_1$

C.2 $\quad k/IF/(F)/THEN/(e_1)/ELSE/e_2 \Rightarrow e_2$

C.3 $\quad k/IF/(e_p)\ e_x \Rightarrow k/IF/(ke_p\perp)\ e_x \perp$

The first two sentences here will be used after the concretization of the predicate into a truth-value has already been performed. The third sentence will bring about concretization of the predicate, should we choose to write it without embracing concretization brackets. It can be translated into English in this way: to concretize a conditional expression, first concretize the predicate.

Logical connectives would not be difficult to define either. But programming branches in this way for a Refal interpreter would be inefficient. We will show this in the following example.

Let a relation (a two place predicate), which is written in the form:

$$k/FOLLOWS/(e_1)/AFTER/e_2 \perp$$

be defined. This relation may be used, e.g., in algebraic manipulation, the variables e_1 and e_2 being very bulky. Suppose we want a procedure of ordering a pair of expressions according to this relation. We define it this way:

$$k/ORDER/(e_1)(e_2) \Rightarrow k/IF(/FOLLOWS/(e_1)/AFTER/e_2)$$
$$/THEN/((e_2)(e_1))/ELSE/(e_1)(e_2) \perp$$

This definition looks familiar to one's eye, but it leads to senseless reproduction of variables during interpretation, and hence to essential loss in efficiency. Each of the variables e_1 and e_2 enters three times the right side of the sentence, while only once on the left. Therefore, the Refal machine will copy it twice — only to destroy both copies shortly afterwards. The first copy is destroyed when the predicate is concretized (because the argument gets lost and replaced by a truth-value); the second copy is destroyed in accordance with one of the sentences C.1 or C.2.

How does one avoid this difficulty?

The Refal machine is very simple and straightforward. To program for it efficiently one has to keep track of the "physical" rearrangements it makes in the view-field when applying a sentence. It must be borne in mind that if a function destroys a free variable the value of which still is to be used later we will be obliged to resort to variable reproduction before using this function. In short, *to avoid unnecessary reproduction of arguments, we must avoid unnecessary destruction of arguments.*

Let us apply this principle to predicates. Instead of the usual predicates, which substitute a truth-value for the argument(s), we shall use recursive functions which retain the argument, and only add to it at the beginning the truth-value resulting from evaluation. Such functions will be referred to as *conservative predicates*. For instance, the conservative predicate α which tells whether an expression contains two identical terms on the top level of parenthesis structure may be defined as

$$\alpha\ e_1 t_x e_2 t_x e_3 \Rightarrow T e_1 t_x e_2 t_x e_3$$

$$\alpha\ e_1 \Rightarrow F\ e_1$$

In our example, we should redefine the predicate /FOLLOWS/ in such a way that the result of concretization is either $T(e_1)e_2$, or $F(e_1)e_2$. This eliminates one unnecessary copying. To eliminate the other, we abandon function /IF/ and make up the following straightforward definition:

$$k\text{/ORDER/}(e_1)(e_2) \Rightarrow k\text{/ORDER1/}\ k\text{/FOLLOWS/}(e_1)\text{/AFTER/}e_2\perp\ \perp$$

$$k\text{/ORDER1/}T(e_1)e_2 \Rightarrow (e_2)(e_1)$$

$$k\text{/ORDER1/}F(e_1)e_2 \Rightarrow (e_1)(e_2)$$

introducing one auxiliary function.

The branching of the algorithmic process is achieved here through the syntactic analysis of the argument: whether it starts with T or F. But the presence of more than one sentence in the definition of a function always generates a branch controlled by syntactic analysis, and conversely, the only way to generate a branch in Refal is to have more than one sentence in the definition of some function. Accordingly, the predicates in Refal lose their role as the only vehicles of branching. The differentiation of functional units into those which only analyze and branch but do not transform, and those which transform without branching, becomes optional and as a general rule unnecessary. A Refal function in a carefully written program performs two jobs: on the input end, it makes branchings and corresponding transformations; on the output end, it leaves clear syntactic indicators to be used for branching by the function which takes up the result of concretization. For example, as the basis for a procedure of ordering a sequence of terms we should take not the predicate of order, but the procedure of ordering two terms, which (1) puts the terms in right order, and (2) adds T or F at the beginning to indicate whether they were initially in right or wrong order.

The number of constant elements in the right sides of sequences also must be considered in effective interpretive programming. If the right side contains many elements which are not present in the left side in the same order, the interpreter will have to insert these elements in the view-field: one by one, if there is no optimization. And if these symbols or brackets become unneeded at the next stage, and get removed from the view-field, this program cannot be recognized as fully efficient. We can amend the program by using functions /BR/ and /DG/ of "burying" and "digging" information.

Consider this example. Let conservative predicate /ORD/ defined on the set of ordered pairs of letters assume value T when the letters are identical or the second appears later in the alphabet than the first, and value F otherwise. The simplest version of the definition is:

$$k/\mathrm{ORD}/s_1 s_2 \Rightarrow \beta\ s_1 s_2\ \mathrm{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$

$$\beta\ s_1 s_2\ e_x s_2 e_y s_1 e_z \Rightarrow F\ s_1 s_2$$

$$\beta\ s_1 s_2 \Rightarrow T\ s_1 s_2$$

This solution has the shortcoming we have just mentioned: each time when the first sentence is used the alphabetical list of letters will be brought into the view-field — and thrown out at the next step. The other solution is to perform the concretization

$$k/\mathrm{BR}/(\mathrm{ALPH})=\mathrm{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \perp$$

at any stage before using hee predicate /ORD/ , which we redefine now in the following way:

$$k/\mathrm{ORD}/s_1 s_2 \Rightarrow \beta\ s_1 s_2 k/\mathrm{DG}/\mathrm{ALPH} \perp \perp$$

$$\beta\ s_1 s_2 e_x s_2 e_y s_1 e_z \Rightarrow F\ s_1 s_2 k/\mathrm{BR}/(\mathrm{ALPH})=e_x s_2 e_y s_1 e_z \perp$$

$$\beta\ s_1 s_2 e_a \Rightarrow T\ s_1 s_2\ k/\mathrm{BR}/(\mathrm{ALPH})=e_a \perp$$

Now the list as a whole is dug and buried, being represented in the form required for the view-field. These procedures take a small time, which is, according to implementation principle R5 independent of the length of the list. Notice, that after using the list, one should not forget that it should be buried again immediately (this is done by function β in this example); otherwise the list will be lost.

In Refal, all iterative processes take the form of functional recursion. However, the difference between simple loops and the use of recursive functions, which is so noticeable in usual programming languages (say, ALGOL 60), has its analogue in Refal, being reflected in the structure of the right sides of sentences. If the right side is a call of the function itself, it is a simple loop: the *configuration* (see Chapter 4) of the view-field does not change, only one argument is replaced by another. Allowing the argument to include calls of other functions, which do not call back the original function, we get nested loops, for example

$$kF^1 \ldots \Rightarrow kF^1 \ldots kF^2 \ldots \perp \perp$$

$$kF^2 \ldots \Rightarrow kF^2 \ldots \perp$$

But if a sentence has the form

$$kF^1 \ldots \Rightarrow kF^2 \ldots kF^1 \ldots \perp \perp$$

each application generates a new pair of concretization brackets — function F^2 calls, which accumulate in the view-field and will be taken up for concretization only after another, nonrecursive, sentence for the function F^1 has been used. This is a recursion in the sense of ALGOL 60.

Consider the factorial function as an example. According to its recursive definition, we can immediately write **this** program which uses recursion in ALGOL:

```
integer procedure FACT(n);
value n;  integer n;
FACT := if n = 0 then 1 else FACT(n-1) × n
```

In Refal, we must first introduce the function α performing arithmetic operations. Let it have the format

$$\alpha\ O\ N_1,\ N_2 \perp$$

where O is the operation sign, N_1 and N_2 are the operands. Then the corresponding description of the function /FACT/ will be:

$$\mathrm{k/FACT/0} \Rightarrow 1$$

$$\mathrm{k/FACT/}e_n \Rightarrow \alpha \times\ \mathrm{k/FACT/}\ \alpha\text{-}e_n, 1 \perp,\ e_n$$

To eliminate the recursive function call in the ALGOL program, we can rewrite it this way:

```
integer procedure FACT(n);
value n; integer n;
begin integer f, m;
f := 1;
for m := 1 step 1 until n do
f := f × m;
FACT := f
end
```

The corresponding program in Refal is:

$$\mathrm{k/FACT/}\ e_n \Rightarrow \mathrm{k/F1/(1)}(\alpha{+}e_n,1)\ 1 \perp$$

$$\mathrm{k/F1/}\ (e_n)(e_n)e_f \Rightarrow e_f$$

$$\mathrm{k/F1/}\ (e_m)(e_n)e_f \Rightarrow \mathrm{k/F1/}(\alpha{+}e_m,1)(e_n)\alpha\times\ e_f,e_m \perp \perp$$

Here the first sentence corresponds to the declaration of the local variables and the initial assignments to them, the second and third sentences correspond to the for statement.

2.6 Decomposition of the Algorithm into Functions

One Refal function is usually attached to the algorithmic problem to be solved. But in a complicated case, to define this function we have to introduce auxiliary functions, which may demand the introduction of new auxiliary functions, etc. For instance, a translator from ALGOL 60 into assembler language takes several dozen functions described, all in all, by a few hundred sentences.

The most usual reasons to introduce an auxiliary function are:

(1) To break down an object into some parts according to a pattern. This pattern will be reflected in the left side of the defining sentence.

(2) To define branching by putting one sentence into correspondence with each particular case.

(3) To change the format of the argument, which is dictated as a rule by the necessity of bringing a new object into the process.

(4) Preliminary processing of the argument in order to describe the main process in a more convenient or efficient way.

Suppose that some object is to be processed by several functions in succession. This can be achieved in two ways. First, we can define all the functions independently, and then define a function which applies them consecutively, for example:

$$k/F/e_x \Rightarrow k/F3/k/F2/k/F1/e_x \perp \perp \perp$$

Second, we can define the first function in such a way that it will call the second when it has finished processing; the second function can call the third in the same manner, etc. The first method has the advantage of independently defined functions, which may be used on different occasions. On the other hand, the second way is more convenient when the functions are introduced *ad hoc*, and the choice as to what function to apply next may depend on the form of the result.

Some functions may not exactly specify what other functions will be called; such functions are called *metafunctions*, they control the use of other functions, dependent on input information. For example, the function /APPLC/ ("apply consecutively") defined as follows:

$$k/APPLC/(\ )\ e_x \Rightarrow e_x$$

$$k/APPLC/(s_1 e_2) e_x \Rightarrow k/APPLC/(e_2)\ ks_1\ e_x \perp \perp$$

takes its first argument (the expression in the pouch) as a list of function determiners and applies these functions consecutively from left to right to the other argument.

In some implementations of Refal it is required that each concretization sign in the program is always followed by a determiner, thus rendering the second sentence inadmissible. However, these implementations provide a special external function /MU/ which works as a universal metafunction. Specifically, the concretization of $k/MU/s_1 e_x \perp$ produces the same results as $ks_1 e_x \perp$ with any values of s_1 and e_x. Therefore, we only need to rewrite the right side of our sentence as follows:

$$k/APPLC/(e_2)\ k/MU/s_1\ e_x \perp \perp$$

2.7 All-Level Scans of Bracket Structures

Remember function ϕ from Section 2.3, which eliminates repeated blanks. It leaves unprocessed those parts of the argument which are enclosed in parentheses. Now we want to modify it in such a way that the argument is processed on all levels throughout its parenthesis structure. The simplest solution to this problem is to insert an additional sentence between the first and the second sentences, which would describe the procedure of entering parentheses:

$$\phi\ e_1 \sqcup\sqcup e_2 \Rightarrow \phi\ e_1 \perp \phi \sqcup e_2$$

$$\phi\ e_1\ (e_2)\ e_3 \Rightarrow e_1\ (\phi e_2)\ \phi e_3$$

$$\phi\ e_1 \Rightarrow e_1$$

Here we also had to modify the first sentence by enclosing the variable e_1 on the right in concretization brackets because it still has to be scanned in order to find out whether it has parenthesized subexpressions. Variable e_1 in the right side of the second sentence can be taken out of concretization brackets, for it **no** longer contains either repeated blanks or parentheses.

This definition is not free of algorithmic inefficiencies. We present two more solutions:

$$\phi\ e_1 \sqcup \sqcup e_2 \ \Rightarrow\ \phi^1 e_1 \perp \phi \sqcup e_2$$

$$\phi\ e_1 \ \Rightarrow\ \phi^1\ e_1$$

$$\phi^1 e_1 (e_2) e_3 \ \Rightarrow\ e_1 (\phi e_2) \phi^1 e_3$$

$$\phi^1 e_1 \ \Rightarrow\ e_1$$

and

$$\phi \sqcup \sqcup e_1 \ \Rightarrow\ \phi \sqcup\ e_1$$

$$\phi\ s_a e_1 \ \Rightarrow\ s_a\ \phi\ e_1$$

$$\phi (e_1) e_2 \ \Rightarrow\ (\phi e_1)\ \phi e_2$$

$$\phi \qquad \Rightarrow$$

leaving it to the reader to analyze the differences between them in the algorithmic aspect.

The language Refal takes parentheses very seriously. The Refal object is a *tree* written in line with the help of parentheses and concretization brackets, and its structure cannot be easily ignored. Whatever way we choose to describe an algorithm in Refal, it remains to be expressed through operations on tree structures.

Consider this example. We want a procedure which scans the object from left to right and of all the entries of each symbol, keeps only the first, deleting the others. This procedure, in fact, ignores the tree structure of the object, it regards parentheses as symbols (but of a special kind, since they should not be deleted). In Refal, we will have to define this procedure as moving around a tree, but unlike the preceding example, we will have to transfer information from one branch of the tree to another.

Let us assign determiner α to our procedure. Obviously, we must maintain a list of symbols already discovered. Let us put it into a pouch, which we will position at the end of the argument. Therefore, the auxiliary function will be introduced as follows:

$$\alpha\ e_1 \Rightarrow \alpha^1\ e_1\ (\ )$$

If there were no parentheses in the arguments, function α^1 would be defined by these three sentences:

$$\begin{aligned}
&\alpha^1\ s_p e_q (e_1 s_p e_2) && \Rightarrow \quad \alpha^1\ e_q (e_1 s_p e_2)\\
&\alpha^1\ s_p e_q (e_1) && \Rightarrow \quad s_p \alpha^1 e_q (e_1 s_p)\\
&\alpha^1\ t_1 && \Rightarrow
\end{aligned}$$

Because of the parentheses, the list of symbols accumulated in one subexpression must be used when scanning another. This means that we must take in this list when entering parentheses (which is easy), and bring it out on exit (which is a bit more difficult). The most universal way to exchange information between any points in tree structures is to use procedures /BR/ and /DG/. We will keep the list of symbols buried under the name LS. On scanning each subexpression delimited by parentheses, function α^1 will bury the up-to-date list, and it will dig it out on the next higher level of the parentheses structure. At the end of the work the list should be dug out and destroyed so as not to waste space.

$$\begin{aligned}
&\alpha\ e_1 && \Rightarrow \quad \alpha^1\ e_1(\ )\perp \text{k/DESTROY/ k/DG/LS} \perp \perp\\
&\alpha^1 s_p e_q (e_1 s_p e_2) && \Rightarrow \quad \alpha^1\ e_q (e_1 s_p e_2)\\
&\alpha^1 s_p e_q (e_1) && \Rightarrow \quad s_p \alpha^1 e_q (e_1 s_p)\\
&\alpha^1 (e_1) e_2 t_s && \Rightarrow \quad (\alpha^1 e_1 t_s)\ \alpha^1 e_2 (\text{k/DG/LS} \perp)\\
&\alpha^1 (e_s) && \Rightarrow \quad \text{k/BR/(LS)} = e_s \perp\\
&\text{k/DESTROY/}e_1 && \Rightarrow
\end{aligned}$$

As an exercise, the reader may define function α without resorting to functions /BR/ and /DG/.

Now we describe some scanning techniques that are coupled with rearrangement of the bracket structure. The parentheses may be removed and restored, if they are replaced by some special symbols which are not used otherwise. Let it be /L/ for the left parenthesis and /R/ for the right. The procedure replacing parentheses is fairly simple:

$$\text{k/REP/ } e_1(e_2)e_3 \Rightarrow e_1 \text{ /L/ k/REP/ } e_2 \text{ /R/ } e_3 \perp$$
$$e_1 \Rightarrow e_1$$

The inverse procedure, which pairs corresponding symbols /L/ and /R/, and replaces them with parentheses, is somewhat more complicated, but, as we shall see, requires only five sentences to define, and no bury-dig functions. We shall give a detailed account of the process of designing this program, in order to illustrate the method of work in Refal.

First, some examples. If after a "quasi-bracket" /L/ immediately (not counting normal symbols) follows a quasi-bracket /R/, they can be paired and replaced by parentheses:

A /L/ B C /R/ D ⇒ A (B C) D

If after an /L/ another /L/ follows, it is the second one that will be paired with the first /R/ to appear, the first /L/ being kept unpaired. The scanned part of the object may be:

A /L/ B (C D E) F G

The general form of the scanned part will be called a *left multibracket*. It contains a number of yet unpaired /L/ , but does not contain any /R/ s. Paired /L/ and /R/ are already replaced by "real" parentheses. How do we represent this type of object in a Refal program? We could keep a multibracket in its "natural" form, putting it in a pouch. But this would require a multiple scan of the argument. The first scan, when we are lengthening the multibracket, is, of course, inevitable.

Other scans are being done when we seek the last symbol /L/ to pair it with the /R/ encountered. We can avoid them if instead of quasi-brackets /L/ which break down the multibracket into separate segments we use structure brackets — parentheses! A multibracket of the form

$$\mathcal{E}_1 \text{ /L/ } \mathcal{E}_2 \text{ /L/ } \mathcal{E}_3$$

can be represented as

$$(\mathcal{E}_1)(\mathcal{E}_2)(\mathcal{E}_3)$$

or else as

$$((\mathcal{E}_1)\mathcal{E}_2)\mathcal{E}_3$$

The latter proves more convenient. Using it, we devise the following definition of the pairing function, which requires exactly one scan of the argument:

$$\begin{aligned}
&\text{k/PAIR/ } e_x && \Rightarrow \quad \alpha(\ )\ e_x \\
&\alpha(e_1) \text{ /L/ } e_2 && \Rightarrow \quad \alpha((e_1))e_2 \\
&\alpha((e_1)e_2) \text{ /R/ } e_3 && \Rightarrow \quad \alpha(e_1(e_2))e_3 \\
&\alpha(e_1)s_p e_2 && \Rightarrow \quad \alpha(e_1 s_p)e_2 \\
&\alpha(e_1) && \Rightarrow \quad e_1
\end{aligned}$$

The notion of a *right multibracket* can be introduced in the same way as the left multibracket. The structure

$$\mathcal{E}_1 \text{ /R/ } \ldots\ \mathcal{E}_{n-1} \text{ /R/ } \mathcal{E}_n$$

will be represented in the form

$$\mathcal{E}_1(\ \ldots\ \mathcal{E}_{n-1}(\mathcal{E}_n)\ \ldots\ )$$

Using these representations, we can describe one more way of performing all-level scans of bracket structures. When we move along the processed expression, the scanned and yet unscanned parts are left and right multibrackets. Suppose, e.g., that we are at the second level down in the bracket structure. This situation may be depicted as follows:

$$\mathcal{E}_1(\mathcal{E}_2(\mathcal{E}_3 \overset{\downarrow}{\ } \mathcal{E}_4)\ \mathcal{E}_5)\ \mathcal{E}_6$$

where the arrow shows the control point. With our representation of multibrackets the argument of the scanning function will be of the form:

$$[((E_1)\ E_2)\ E_3]\ E_4(E_5(E_6))$$

(For the sake of lucidity, the format parentheses are represented here as square brackets.) Therefore this way of scanning is coupled with rearrangement of the processed tree: the subexpression worked on is raised to the top level of bracket structure.

There is a technical point in dealing with the abovementioned argument which ought to be indicated: we must avoid confusing the parentheses introduced to represent multibrackets with "genuine" parentheses representing the original structure of the argument, which may happen if it ends with a right parenthesis. Suppose, e.g., that the processed expression is

(A)BC(D)

At the moment when the control point has just passed symbol C, the argument of the function will be

[(A)BC](D)

Suppose now that the original expression was

A(BC)D

When the control point passes C, the argument of the function will be

[(A)BC](D)

which is indistinguishable from the first case, indicating an amgiguity.

To eliminate this difficulty, we may add to the original expression any symbol, e.g., an asterisk, when we call an auxiliary function with a pouch; thus, a terminal right parenthesis in the argument (and only it) will be always attributed to a multibracket. In the end the asterisk should be of course removed.

The above function α, which removes all repeated entries of symbols, will be defined using this technique in the following way:

1. FORMAT TRANSFORMATION. SECOND POUCH IS FOR SYMBOL LIST

$$\alpha e_1 \Rightarrow \alpha^1(\ )e_1 {*} (\ )$$

2.1. END OF JOB

$$\alpha^1(e_1) * t_s \Rightarrow e_1$$

2.2. REPEATED SYMBOL

$$\alpha^1(e_1)s_p e_q(e_2 s_p e_3) \Rightarrow \alpha^1(e_1)e_q(e_2 s_p e_3)$$

2.3. NEW SYMBOL

$$\alpha^1(e_1)s_p e_q(e_s) \Rightarrow \alpha^1(e_1 s_p)e_q(e_s s_p)$$

2.4. RIGHT PARENTHESIS IN ORIGINAL EXPRESSION

$$\alpha^1((e_1)e_2)(e_3)t_s \Rightarrow \alpha^1(e_1(e_2))e_3 t_s$$

2.5. LEFT PARENTHESIS IN ORIGINAL EXPRESSION

$$\alpha^1(e_1)(e_2)e_3 t_s \Rightarrow \alpha^1((e_1))e_2(e_3)t_s$$

If we have finished the work before completing an all-level scan, the argument in the "inside-out" form

$$[(\ldots(\mathcal{E}_1)\ldots)\mathcal{E}_m]\ \mathcal{E}_{m+1}(\ldots(\mathcal{E}_n^*)\ \ldots\ )$$

can be brought back to normal by applying the backtracking function β:

$$\beta\ (e_1)e_2(e_3) \Rightarrow \beta\ e_1(e_2)e_3$$

$$\beta\ e_1 \Rightarrow e_1$$

with the format parentheses preliminarily removed:

$$\beta(\ldots(\ \mathcal{E}_1)\ldots)\ \mathcal{E}_m\ \mathcal{E}_{m+1}\ (\ldots(\ \mathcal{E}_n\ *\ )\ \ldots)\ \bot$$

2.8 An Example: Translation of Arithmetic Expressions.

As an example of a more complicated program, we list the definition of a function (/TRAREX/), which translates an arithmetic expression into a program for the assembly language of a one-addressed computer. Elementary operands are ALGOL 60 identifiers and integers. Literal constants in the resulting program are formed with the help of an equality sign, e.g.

ADD, =36;

Intermediate results are stored in R + 0, R + 1, etc. Syntactical correctness is checked and error printings are made. Incorrect subexpressions are replaced by the identifier ERROR.

First, the argument expression is parsed into a more convenient form for translation, which has a prefix structure with the prefix being either an operation sign, or an asterisk indicating an elementary operand. The parsing is done from right to left. The pouch in the format of the function /TRANSL/ contains the displacement to store the next intermediate result.

TRANSLATION OF ARITHMETIC EXPRESSIONS

$\text{k/TRAREX/ } e_a \Rightarrow \text{k/TRANSL/ (0) k/PARSE/ } e_a \perp \perp$

$\alpha = \text{k/PARSE/}$

$\alpha + e_1 \Rightarrow \alpha\ e_1$

$\alpha - e_1 \Rightarrow \alpha\ 0 - e_1$

$\text{(R) } \alpha\ e_1 s(+\ -)_f\ e_2 \Rightarrow s_f\ (\alpha\ e_1) \alpha^1 e_2$

$\alpha\ e_1 \Rightarrow \alpha^1 e_1$

$\text{(R) } \alpha^1 e_1 s(x\ /)_f\ e_2 \Rightarrow s_f (\alpha^1\ e_1)\ \alpha^2\ e_2$

$\alpha^1\ e_1 \Rightarrow \alpha^2\ e_1$

$\text{(R) } \alpha^2\ e_1 \uparrow e_2 \Rightarrow \uparrow (\alpha^2\ e_1)\ \alpha^3\ e_2$

$\alpha^2\ e_1 \Rightarrow \alpha^3\ e_1$

$\alpha^3\ (e_1) \Rightarrow \alpha\ e_1$

$\alpha^3 \text{s/LETTER/}_1\ e_2 \Rightarrow \text{k/IDENT/}(s_1) e_2 \perp$

$\alpha^3 \text{s/DIGIT/}_1\ e_2 \Rightarrow \text{k/NUMBER/}(s_1)\ e_2 \perp$

$\alpha^3\ e_x \Rightarrow \text{ERROR k/P/ ERROR: } e_x \perp$

$\text{k/IDENT/}(e_1)\ \text{s/LETTER/}_2\ e_3 \Rightarrow \text{k/IDENT/}(e_1 s_2)\ e_3 \perp$

$\qquad\qquad (e_1)\ \text{s/DIGIT/}_2\ e_3 \Rightarrow \text{k/IDENT/ } (e_1 s_2)\ e_3 \perp$

```
         (e1) ⇒ * e1
         (e1) e2 ⇒ ERROR  k/P/ ERROR: e1 e2 ⊥
k/NUMBER/ (e1) s/DIGIT/2 e3 ⇒ k/NUMBER/ (e1s2) e3 ⊥
           (e1) ⇒ * = e1
           (e1) e2 ⇒ ERROR k/P/ ERROR: e1 e2 ⊥
k/TRANSL/ (en) sf(e1) * e2 ⇒ k/TRANSL/ (en) e1 ⊥
                             k/CODE/ sf ⊥ , e2 ;
           (en) s(+×)f (* e1) e2 ⇒ k/TRANSL/ (en) e2 ⊥
                                 k'CODE' sf ⊥, e1;
           (en) sf(e1)e2 ⇒ k /TRANSL/ (en) e2 ⊥
                          STORE, R + en;
                          k/TRANSL/ (k/PLUS1/en ⊥ ) e1 ⊥
                          k/CODE/ sf ⊥, R + en;
           (en) * e1 ⇒ LOAD, e1;
k/CODE/ + ⇒ ADD
        - ⇒ SUB
        × ⇒ MULT
        / ⇒ DIV
        ↑ ⇒ POWER
```

The functions /LETTER/, /DIGIT/ and /PLUS1/, the meaning of which is obvious, can be easily described in Refal, but it is preferable to have them implemented as external functions.

CHAPTER 3. EQUIVALENT TRANSFORMATION

3.1 Strict Refal

When a formal system is developed, the first step to take along the way of metasystem transition is to introduce a concept of equivalencè and elaborate a system of equivalent transformations of formal objects. In the case of Refal it has been discovered that a compact yet powerful system of equivalent transformations of algorithms and functions can be formulated ([21]) if the basic version of the language is somewhat restricted. These restrictions are:

(1) The functions /BR/ and /DG/ are excluded;

(2) t-variables are excluded;

(3) in the left sides of sentences there must be no open variables, neither repeated entries of e-variables (repeated s-variables are permitted).

For the sake of simplicity we shall assume later on that every sentence has a determiner, and sentences are grouped into function descriptions.

We call this language Strict Refal. The translation of a program from Basic Refal into Strict Refal can be easily made automatically. We note though, that the restrictions are not very severe and leave the language expressive enough to conveniently describe most complicated algorithms.

A pattern expression will be called an *L-expression* if no one of its subexpressions contains more than one e-variable not enclosed in parentheses, and no one e-variable enters the expression twice. In strict Refal the left sides of sentences are L-expressions. From the definition of L-expression there immediately follows

Theorem 3.1. Each subexpression of an L-expression is an L-expression.

Examples of L-expressions:

ABC

$/RS/(s(+-)_1 e_2 (Ae_3 B))s_1$

$(e_1)(e_2)(e_3)e_4$

Examples of pattern expressions which are not L-expressions:

$e_1 + e_2$

$s_1\ e_2\ (s_3\ e_2)$

Due to the absence of open e-variables the following proposition holds:

Theorem 3.2. Let E_0 be an object expression and E_ℓ an L-expression. There can exist only one set of values of the free variables in E_ℓ such that the substitution transforms E_ℓ into E_0.

To demonstrate this we only need to review the projecting algorithm in 2.2. All rigid elements are projected uniquely. Having projected them all, we come to a situation where the only (if any) e-variable present on each level of bracket structure in each subexpression has both its ends projected, and therefore, gets its value uniquely also.

3.2 Classes and Subclasses

To every pattern expression E_p a set corresponds, which comprises all object expressions syntactically recognizable as E_p. We call this set a *class depicted by* E_p, or simply a *class* E_p. A class depicted by an L-expression is called an *L-class*. To denote set-theoretic relations and operations on classes we shall use the usual signs: $\subseteq$, = , $\cap$, $\cup$. No distinction will be made between object expression and the class it depicts. Thus the relation $E_0 \subseteq E_p$, where E_0 is an object expression, means that E_0 is recognizable as E_p.

In the formal description of Refal the requirements 2.1.1 to 2.1.3 for the values of free variables were formulated. Now we generalize the notion of value by allowing values to contain free variables. Points 2.1.1 and 2.1.2 are modified in this way:

2.1.1*. The value of an unspecified s-variable can be any symbol or any s-variable. The value of an e-variable can be any pattern expression.

2.1.2*. The value of a specified variable can be a symbol from its specifier or a specified s-variable with a specifier containing a subset of the set of symbols contained in the specifier of the replaced variable.

The simultaneous substituion of expressions E_i for variables V_i , where $i = 1,2,\ldots,n$, will be written as

$$V_1 \to E_1,\ V_2 \to E_2\ ,\ \ldots,\ V_n \to E_n$$

(It will always be implied that the substitution is legitimate, i.e., the above mentioned requirements to the values are satisfied.)

The result of applying a substition Δ to a pattern E_p expression will be denoted by $E_p\ //\ \Delta$. By $E\ //\ \Delta_1\ //\ \Delta_2$ we mean $(E\ //\ \Delta_1)\ //\ \Delta_2$.

Theorem 3.3. Let E_p be a pattern expression and Δ a substitution. Let $E'_p = E_p\ //\ \Delta$. Then $E'_p \subseteq E_p$.

Proof: Let us take an arbitrary object expression E_0. Suppose it is recognizable as E'_p. This means that a substitution Δ_0 exists, such that $E_0 = E'_p\ //\ \Delta_0$. Therefore $E_0 = E_p\ //\ \Delta\ //\ \Delta_0$. It is easy to see that the composition of two legitimate substitutions is a legitimate substitution again. Hence, E_0 can be seen as a result of applying composition $\Delta\ \Delta_0$ to E_p , which means that E_0 is recognizable as E_p. Thus for any E_0 it follows from $E_0 \subseteq E'_p$ that $E_0 \subset E_p$. This is just what is implied in relation $E'_p \subseteq E_p$.

A class E'_p , obtained as $E_p\ //\ \Delta$ will be referred to as a *subclass* of E_p; the corresponding operation will be called *the contraction of* E_p *through* Δ.

An Example. The class of expressions which begin and end with the same letter, this letter being A or B, is depicted by the pattern expression $s(AB)_1\ e_2\ s_1$. Through the substitution $s_1 \to A,\ e_2 \to =(e_2)$ we contract it to the subclass $A = (e_2)A$.

We note, that a subclass of an L-class is not, generally, an L-class. For instance, any class is a subclass of the L-class e_1.

We now take up the question of finding the union and intersection of classes. The union of two classes may or may not be a class. For example, the union of classes $s(12)_a$ and $s(34)_b$ is class $s(1234)_a$. On the other hand, the set of all terms is the union of two classes: s_1 and (e_1) , and it cannot be represented as a class. The best representation for this set is the union $s_1 \cup (e_1)$. The intersection of two classes also may not be a class. We put the problem of representing the intersection of two classes in the form of a union of classes. The solution of this problem will be given for the special case when at least one of the two classes is an L-class. To perform equivalent transformations, this case will suffice, which, of course, reflects the fact that in strict Refal the left sides of sentences are L-expressions.

Thus, let E_ℓ be an L-expression, and E_p an arbitrary pattern expression. We can see the intersection $E_p \cap E_\ell$ as the set of all those object expressions E_0 from E_p which are recognizable as E_ℓ. The problem, therefore, is a generalization of the problem of syntactical recognition. In particular, if E_p happens to be an object expression, then $E_p \cap E_\ell$ is either E_p itself, in the case when E_p is recognizable as E_ℓ , or empty when this is not the case. The answer here may be obtained by using the algorithm of projecting (mapping) E_ℓ on E_p. In the general case we also will find a solution by projecting E_ℓ on E_p , but we must keep in mind that E_p represents now *a set* of object expressions, not a single one; in the process of projecting we shall contract this set, excluding from it subsets of those elements (object expressions) which certainly are not recognizable as E_ℓ.

Those terms of which an expression is by definition a sequence will be referred to as its *constituent terms*. If object expressions E_1 and E_2 are identical, their first constituting terms must also be identical. Of all constituting terms of E_ℓ only the e-variable may be transformed by a substitution into an expression consisting of several (or no) terms; other terms are always transformed into exactly one term. Therefore, if E_ℓ can be represented in the form $T_\ell E'_\ell$, where T_ℓ does not have the form eZ , the term T_ℓ can be projected on the first constituting term of E_p separately, and if this latter one is T_p , a legitimate substitution must exist for successful recognition, which transforms T_ℓ into T_p. For such a substitution to exist, we will sometimes have to make contractions in T_p , and therefore in E_p.

Analogous considerations are applicable, of course, to the last (rightmost) constituting term. In defining the projecting algorithm, there is a freedom of which constituting term, the leftmost or rightmost, to choose at each step. The final result will of course be the same independent of the strategy taken, but the length of the process may be very different in some cases. The simplest version is to move left to right until an e-variable is met, and then start from the right end back to this variable. This version will be used in the examples below.

We proceed now to describe the generalized rules of projecting. Contractions in E_p will be written as substitutions $V \rightarrow E$. When a variable from E takes a value, this will be written as an assignment $V \leftarrow E$. Clearly, this is also a substitution, which produces a contraction, but in E_ℓ , and not in E_p. We use arrows directed in different sides to show where the contraction is made. This is important, because a variable from E_p may occasionally be identical (syntactically) to a variable from E_ℓ. To avoid confusion we will sometimes denote the empty expression (and the class consisting of the empty expression) by <empty>. The empty set (which is not a class) will be denoted as $\emptyset$. For the sake of brevity, we will assume that all s-variables have specifiers. An unspecified s-variable can be interpreted as having all possible symbols in its specifier.

The following points cover all possible cases which may arise in projecting. By X , Y , Z and Y_i (with any i) we shall denote arbitrary indexes, by S and S_i — any symbols, and by P, Q , and R — any specifiers.

1. E_ℓ is empty. Recognition is possible if E_p is either empty or has the form

$$e^{\mathcal{Y}}_1\ e^{\mathcal{Y}}_2\ \ldots\ e^{\mathcal{Y}}_n$$

where some indexes $\mathcal{Y}_i$ may coincide. In the latter case, contraction

$$e^{\mathcal{Y}}_1 \rightarrow \text{<empty>},\ e^{\mathcal{Y}}_2 \rightarrow \text{<empty>},\ \ldots,\ e^{\mathcal{Y}}_n \rightarrow \text{<empty>}$$

is needed. Otherwise, recognition is impossbble, i.e. not a single object expression from E_p is recognizable as E_ℓ.

2. E_ℓ is eZ. Recognition is possible with any E_p. As a result, the variable eZ takes the value E_p , which fact is depicted as

$$eZ := E_p\ .$$

In all the following points E_ℓ takes the form of either $T_\ell E_\ell^{'}$, or $E_\ell^{'} T_\ell$, where T_ℓ is any term which is not an e-variable. In the first case we represent E_p in the form $T_p E_p^{'}$, in the second case — in the form $E_p^{'} T_p$, where T_p is any term. If this is impossible (that is E_p is empty) then recognition is impossible. We formulate the rules for the first case (leftmost constituting term), the rules for the other cases are analogous and will be referred to by the same pair of numbers with an added asterisk, such as 3.1*. On applying one of the rules 3.1* - 6.4*, we take into account the contractions and assignments indicated in the rule by making appropriate substitution in E_p and E_ℓ , after which we proceed to project yet unprojected parts of the original E_ℓ.

3. T_ℓ is S. This holds for all points 3.n with any n.

3.1. T_p is S_1. Recognition is possible only if S_1 is identical to S.

3.2. T_p is s(Q)$\mathcal{Y}$. If S enters Q we make the contraction s$\mathcal{Y}$ → S and continue projecting, otherwise recognition is impossible.

3.3. T_p is (E_p^i). Recognition impossible.

3.4. T_p is e$\mathcal{Y}$. Let us divide the class E_p into two subsets: the first subset will be the subclass obtained by applying to E_p the substitution e$\mathcal{Y}$ → <empty>, the second will be the complement, including all the remaining object expressions. We cannot state anything concerning the first subset, we will just continue to project E_ℓ on this subclass. In the second subset (which is not a subclass) recognition can be possible only if the leftmost term of the value of e$\mathcal{Y}$ is S, that is, only in the subclass of E_p formed by the substitution e$\mathcal{Y}$ → Se$\mathcal{Y}$. To sum up, we form two subclasses of E_p by the substitutions

e$\mathcal{Y}$ → <empty>

e$\mathcal{Y}$ → Se$\mathcal{Y}$

and continue the process of projecting in each of these, independently of one another.

4. T_ℓ is s(P)Z.

4.1. T_p is S_1. Recognition is possible if S_1 enters P. In this case sZ takes the value S_1.

4.2. T_p is s(Q)$\mathcal{Y}$. The needed contraction is s$\mathcal{Y}$ → s(R)$\mathcal{Y}$, where R is the intersection of the sets P and Q (if R is empty, recognition is impossible). The variable sZ is taking s(R)$\mathcal{Y}$ as a value. Now we face a problem. We have to make the corresponding substitution in E_ℓ, but we do not want to mistake the variable s$\mathcal{Y}$ from E_p for one of the variables in E_ℓ. To resolve the conflict, we will introduce a new type of element, the *alien* s-variable, with the variable type sign *a* instead of s. In our case the substitution will amount to replacing all entries of s(P)Z in E_ℓ by *a*(R)$\mathcal{Y}$. The projecting of a-variables will be described in point 6. Since E_ℓ now has references to the variables from E_p , a contraction of E_p may demand some modification of E_ℓ. Specifically, when a contraction of the form s(P)$\mathcal{Y}$ → E is made in E_p , we must replace all the alien variables with the index $\mathcal{Y}$ in E_ℓ by the value E in which the variable type sign s, if present, is changed to *a*.

4.3. T_p is (E_p^i). Recognition impossible.

4.4. T_p is e$\mathcal{Y}$. Two subclasses of E_p are generated by the contractions:

$$e\mathcal{Y} \rightarrow \text{<empty>}$$

$$e\mathcal{Y} \rightarrow s(P)\mathcal{Y}_1 \; e\mathcal{Y}$$

Here $s(P)\mathcal{Y}_1$ is a new variable with an index $\mathcal{Y}_1$ differing from all other indexes.

5. T_ℓ is (E^i_ℓ).

5.1 & 5.2. T_p is S_1 or $s(Q)\mathcal{Y}$. Recognition impossible.

5.3. T_p is (E^i_p). Recognition is possible if E^i_p is recognizable as E^i_ℓ. Therefore we proceed to project E^i_ℓ on E^i_p, and for those subclasses on which the projection is possible continue the process by resuming projection of E'_ℓ on E'_p. The contractions and assignments made in projecting E^i_ℓ on E^i_p must be taken into account when we resume projecting on the main level of bracket structure.

5.4. T_p is $e\mathcal{Y}$. Two subclasses of E_p are generated by the contractions

$$e\mathcal{Y} \rightarrow \text{<empty>}$$

$$e\mathcal{Y} \rightarrow (e\mathcal{Y}_1) \; e\mathcal{Y}$$

where $e\mathcal{Y}_1$ is a new variable with index $\mathcal{Y}_1$ differing from all other indexes.

6. T_ℓ is an *alien* variable $a(P)\mathcal{Y}$.

6.1. T_p is S. If S enters P then we make the contraction $s\mathcal{Y} \rightarrow S$ (which means, incidentally, that if there are other entries of the same alien variable in E_ℓ, they will be replaced by S, see Rule 4.2). Otherwise, recognition is impossible.

6.2. T_p is $s(Q)X$. Let R be the intersection of P and Q. If it is empty, recognition is impossible. If R is not empty, we make the contraction in E_p through the following simultaneous substitution:

$$s(Q)X \rightarrow s(R)\mathcal{Y} \; , \quad s(P)\mathcal{Y} \rightarrow s(R)\mathcal{Y}$$

(in E_ℓ , the alien variable $a\mathcal{Y}$ will be replaced by $a(R)\mathcal{Y}$). We note that index X may be identical to $\mathcal{Y}$.

6.3. T_p is (E_p^i). Recognition impossible.

6.4. T_p is eX. Two subclasses of E_p are generated by the substitutions:

$$eX \rightarrow \text{<empty>}$$

$$eX \rightarrow s(P)Y\ eX$$

As we mentioned above, when the term being projected T_ℓ is the rightmost, the rules are analogous, and the formulations differ only if an e-variable from E_p is involved. For example, instead of 6.4, we will have:

6.4*. T_p is eX. Two subclasses of E_p are generated by the substitutions:

$$eX \rightarrow \text{<empty>}$$

$$eX \rightarrow eX\ s(P)Y$$

In the process of projecting, the variables of E_ℓ take some values, which may contain some variables from E_p and may, therefore, alter when contractions take place. In order to get the correct list of the values of variables in the end of the projection, we must update this list when each new contraction is made by making the substitution in the values.

Applying these rules step by step, we generate a branching process, each branch corresponding to one subclass of the original class E_p. Some of these branches may be terminated by the verdict "Recognition impossible", others may come to a successful end through the use of rules 1 or 2. Obviously, the process of projecting will always be finite. Thus we obtain the following theorem:

Theorem 3.4. The Generalized Projecting Algorithm (GPA), when applied to the projection of an L-expression E_ℓ on an arbitrary pattern expression E_p , provides a representation of the set $E_\ell \cap E_p$ in the form

$$E_p^1 \cup E_p^2 \cup \ldots \cup E_p^n$$

where E_p^i are subclasses of the class E_p and n is a finite number, which may be in particular equal to zero ($E_\ell \cap E_p = \emptyset$).

Let us consider several examples of generalized projecting. For the sake of brevity, we will introduce the following conventions. To indicate the rule used at each step, its number designation will be placed at the beginning of the line. Period, comma and equality signs will be used as delimiters; to avoid confusion, these will not be used as object signs in Refal expressions. In branching, we will always first take up the subclass generated by the substitution $e^y \to$ <empty>. Having finished the whole tree which is produced by this branch, we come back to the remaining alternative. To note this, we mark each branching point by placing its sequential number in parentheses immediately after the rule's designation. Thus each sequential number will appear exactly twice.

1) Let us find the intersection of the following classes:

$$\mathcal{E}_\ell = A\ s_1\ (e_2)\ e_3\ s_1\ A$$

$$\mathcal{E}_p = A\ s_a\ (C + e_b)\ e_c$$

3.1. The sign A is projected on the identical sign in $\mathcal{E}_p$.

4.2. $s_1 := s_a$, $\mathcal{E}_\ell' = (e_2)\ e_3\ a_a A$. Now $\mathcal{E}_\ell'$ takes on the role of $\mathcal{E}_\ell$. Therefore in the following we shall not distinguish between them and use $\mathcal{E}_\ell$ as the notation.

5.3. $\mathcal{E}_\ell^i = e_2$, $\mathcal{E}_p^i = C + e_b$. We enter the parentheses.

2. $e_2 := C + e_b$. Return to the main level of the bracket structure.

$$\mathcal{E}_\ell = e_3\ a_a\ A\ , \quad \mathcal{E}_p = e_c\ .$$

3.4*. (1) $e_c \to$ <empty>. Recognition impossible. Return to (1).

3.4*. (1) $e_c \to e_c\ A$

3.1*. $\mathcal{E}_\ell = e_3\ a_a$, $\mathcal{E}_p = e_c$.

6.4*. (2) $e_c \to$ <empty>. Recognition impossible. Return to (2).

6.4*. (2) $e_c \to e_c\ s_a$

$\mathcal{E}_\ell = e_3$, $\mathcal{E}_p = e_c$.

2. $e_3 := e_c$.

Recognition attained. Collecting all the substitutions made, we find that $E_\ell \cap E_p$ is the subclass of E_p, obtained by the substitution

$$e_c \rightarrow e_c\ s_a\ A\ ,$$

that is the class

$$A\ s_a\ (C + e_b)\ e_c\ s_a\ A\ .$$

It can be recognized as E_ℓ if the free variables take the values:

$$s_1 := s_a\ ,\quad e_2 := C + e_b\ ,\quad e_3 := e_c\ .$$

2) The second example. Let

$$E_\ell = s_1\ (e_2)\ e_3\ s_4\ A$$

$$E_p = e_1 + e_2$$

(In the following we shall use the notation X(n) meaning: "Recognition impossible. Return to n" .)

4.4. (1) $e_1 \rightarrow$ <empty>

4.1. $s_1 := +$, $E_\ell = (e_2)\ e_3\ s_4\ A$, $E_p = e_2$.

5.4. (2) $e_2 \rightarrow$ <empty>, X(2)

5.4. (2) $e_2 \rightarrow (e_3)\ e_2$

2. $e_2 := e_3$

$E_\ell = e_3\ s_4\ A$, $E_p = e_2$.

3.4*. (3) $e_2 \rightarrow$ <empty>, X(3)

3.4*. (3) $e_2 \rightarrow e_2\ A$

4.4*. (4) $e_2 \rightarrow$ <empty>, X(4)

4.4*. (4) $e_2 \rightarrow e_2\ s_4$

4.2*. $s_4 := s_4$

2. $e_3 := e_2$

Recognition attained. We have received the subclass

$$E_p^1 = +\ (e_3)\ e_2\ s_4\ A$$

Return to (1).

4.4. (1) $e_1 \to s_3 e_1$

4.2. $s_1 := s_3$

$\mathcal{E}_\ell = (e_2)\ e_3\ s_4\ A\ , \quad \mathcal{E}_p = e_1 + e_2\ .$

5.4. (5) $e_1 \to$ <empty>, X(5)

5.4. (5) $e_1 \to (e_4)\ e_1$

2. $e_2 := e_4$

3.4*. (6) $e_2 \to$ <empty>, X(6)

3.4*. (6) $e_2 \to e_2\ A$

$\mathcal{E}_\ell = e_3\ s_4\ , \quad \mathcal{E}_p = e_1 + e_2\ .$

4.4*. (7) $e_2 \to$ <empty>

4.1*. $s_4 := +$

2. $e_3 := e_1$

Recognition attained. We have received the subclass

$$\mathcal{E}_p^2 = s_3\ (e_4)\ e_1 + A$$

Return to (7).

4.4*. (7) $e_2 \to e_2\ s_5$

4.2*. $s_4 := s_5$

2. $e_3 := e_1 + e_2$

Recognition attained. We have received the subclass

$$\mathcal{E}_p^3 = s_3\ (e_4)\ e_1 + e_2\ s_5\ A$$

and finished the process of projecting. Thus

$$\mathcal{E}_\ell \cap \mathcal{E}_p = \mathcal{E}_p^1 \cup \mathcal{E}_p^2 \cup \mathcal{E}_p^3$$

The three subclasses which resulted from the application of the GPA in this case are overlapping. Thus, the object expression +(A)+A belongs to the first two classes, the expression +(A)++A belongs to all three classes.

3) Consider an example of generalized projecting which involves specified symbol variables.

$\mathcal{E}_\ell = s_4\ (e_2\ s(ABC)_1\ )\ s_4\ s(ABC)_1$

$\mathcal{E}_p = s(AB)_1\ (BCe_x\ s(AB)_1\ )e_z\ s(BC)_2$

4.2. $s_4 := s(AB)_1$

$\mathcal{E}_\ell = (e_2\ s(ABC)_1)a(AB)_1\ s(ABC)_1$

5.3. $\mathcal{E}^i_\ell = e_2\ s(ABC)_1$, $\mathcal{E}^i_p = BCe_x\ s(AB)_1$

4.2*. $s(AB)_1 \to s(AB)_1$, $s(ABC)_1 := s(AB)_1$

2. $e_2 := BC\ e_x$

The projection inside the parentheses is completed. The values assigned to the variables s_1 and e_2 must be substituted into the remaining part of $\mathcal{E}_\ell$, and the contraction in $\mathcal{E}^i_p$ (in this case trivial) must be expanded on the whole of $\mathcal{E}_p$. As a result we get:

$\mathcal{E}_\ell = a(AB)_1\ a(AB)_1$, $\mathcal{E}_p = e_z\ s(BC)_2$

6.4. (1) $e_z \to$ <empty>

6.2. $s(BC)_2 \to s(B)_1$

$\mathcal{E}_\ell = a(AB)_1$, $\mathcal{E}_p =$ <empty>, X(1)

6.4. (1) $e_z \to s(AB)_1\ e_z$

$\mathcal{E}_\ell = a(AB)_1$, $\mathcal{E}_p = e_z\ s(BC)_2$

6.4. (2) $e_z \to$ <empty>

6.2. $s(BC)_2 \to s(B)_1$, $s(AB)_1 \to s(B)_1$

Recognition attained. We get the sublcass

$\mathcal{E}^1_p = s(B)_1\ (BCe_x\ s(B)_1\ )\ s(B)_1\ s(B)_1$

If the specifier of a symbol variable consists of one symbol, we can replace the variable by this symbol:

$$\mathcal{E}^1_p = B(BC\ e_x\ B)BB$$

Now we return to the branching (2).

6.4. (2) $e_z \to s(AB)_1\ e_z$

$\mathcal{E}_\ell =$ <empty>, $\mathcal{E}_p = e_z\ s(BC)_2$.

Recognition impossible. Hence, the subclass E_p^1 is equal to the intersection $E_\ell \cap E_p$.

As we could see above, the classes E_p^k resulting from the use of the GPA may generally overlap. But if E_p is also an L-expression, these classes will be nonoverlapping.

Theorem 3.5. The intersection of two L-classes obtained through the use of the GPA is the union of nonoverlapping L-classes.

Proof: The subclasses E_p^k are obtained from the L-expression E_p by the substitutions listed in the rules of generalized projecting 1 to 6.4*. In these substitutions symbol variables never generate expression variables, and expression variables may generate a new expression variable only confined by parentheses (Rules 5.4 and 5.4*). Therefore, an L-expression may only generate an L-expression.

More than one subclass may be generated by the application of one of the *branching* rules: n.4 and n.4* with n equal to 3, 4, 5 or 6. These rules provide substitutions for e-variables in E_p. If E_p does not contain e-variables, the GPA will never give more than one subclass. If on the main (top) level of bracket structure, the expression E_p does not have an e-variable, all the generated subclasses will be identical on the main level, so that to compare two subclasses we must compare their subexpressions confined by parentheses. By induction, we see that now we only have to consider the case of an e-variable e*Z* on the main level of bracket structure.

Let us denote by n the number of constituting terms in E_p without the term e*Z*. Application of any legitimate substitution to any other constituting term will not alter the number of constituting terms in E_p. Consider the first branching. When we schoose the first alternative, i.e. e*Z* → <empty>, we receive a subclass, each element of which consists of exactly n constituting terms. When we choose the second alternative, we receive a subclass, each element of which consists of at least n+1 constituting terms. Consequently, these two subclasses, and any pair of subclasses which may be obtained from them through subsequent substitutions, will not overlap. The same consideration is true for the second and all subsequent branchings, which proves the theorem.

The following theorem, which is a generalization of Theorem 3.5, is central for equivalent transformations:

Theorem 3.6. Let L_1 and L_2 be L-expressions, and E_p an arbitrary pattern expression, which may contain only those free variables which are present in L_1. Let $<\Delta^k>$, with k = 1,2,...r be the set of substitutions in E_p generated by projecting L_2 on E_p.(*) Then $L_1 // \Delta^k$, where k = 1,2,...,r, are non-overlapping L-classes.

To prove this theorem we only need to note that the demonstration of theorem 3.5 remains valid if the substitutions applied to the L-expression E_p (which becomes L_1 in Theorem 3.6) are obtained by projecting E_ℓ (which becomes L_2 in Theorem 3.6) not necessarily on itself, but on any pattern expression (E_p in Theorem 3.6).

3.3. Algorithmic Equivalence

We shall make a distinction between *algorithmic* and *functional* equivalence. By *algorithm* we mean the ordered set of sentences in the memory-field of the Refal machine. An algorithm A' will be called *strictly equivalent* to an algorithm A, if the replacement of A by A' in the memory field will not change the result of any step, performed by the Refal machine with any contents of the view-field.(**) This means that if the result is *recognition impossible* in one case, then it also must be *recognition impossible* in the other, and if the step is successfully performed, the resulting view-field must be the same in both cases.

(*) When new variables are introduced in the process of projecting, they must be different from all variables entering L_1, not only E_p.

(**) Instead of "contents of the view-field" we shall say later on just "view-field".

If A' is strictly equivalent to A, then *the domain* of A', i.e. the set of all those view-fields which do not lead to an abnormal stop (recognition impossible) after any number of steps of the Refal machine, is equal to the domain of A. It is useful to weaken this requirmment. We shall call an algorithm A' just *equivalent*(*) to A, if the results of executing one step of the Refal machine with A' and with A in the memory-field are related in the following way: if A does not lead to an abnormal stop, then A' produces the same new view-field as A; if A leads to an abnormal stop, then A' may produce any result. Therefore, the domain of the algorithm may be expanded when we transform A into A'.

For the algorithms which never lead to an abnormal stop, the notions of (nonstrict) equivalency and strict equivalency are coextensive. It is not difficult to write algorithms in Refal in such a way that abnormal stops become impossible. For this end, one has only to see that for each function F, the union of all classes in the left sides of the sentences is the complete set of object expressions. In particular, one may add to the description of the function F a sentence with the left side

$$k\ F\ e_1 \Rightarrow$$

thereby expanding the domain as is deemed convenient. However, it may lead to an unwarranted lengthening of the program. Suppose, for example, that we need a function, which would remove the first symbol of its argument. Suppose, furthermore that all calls of this function are such that the argument in fact begins with a symbol. Then we need only one sentence to define this function (to which we attach, say, the determiner α):

$$\alpha\ s_1\ e_2 \Rightarrow e_2$$

The algorithm which uses the function α may have the domain equal to the complete set, although the description of the function α has a narrower domain. No matter how we expand

(*) or *nonstrictly* equivalent.

the domain of the function α, this will not change the algorithm as a whole. Thus a *nonstrictly* equivalent transformation of some parts of an algorithm may turn useful even if we are interested in a *strictly* equivalent transformation of the whole algorithm.

We formulate now the following five rules of algorithmically equivalent transformation.

A1. At the end of the description of a function F an arbitrary sentence with the same determiner F can be added.

(This rule is the result of our definition of equivalency, which allows expansion of the domain.)

A2. If the intersection of the left sides of two adjacent sentences is empty, these sentences can be transposed.

A3. If a sentence with the left side L_1 precedes a sentence with the left side L_2 , and $L_2 \subseteq L_1$, then the second sentence can be eliminated.

A4. Suppose that for a pair of adjacent sentences

$$k\ L_1 \Rightarrow R_1$$

$$k\ L_2 \Rightarrow R_2$$

a substitution Δ exists, such that $L_1 = L_2 // \Delta$, and $R_1 = R_2 // \Delta$. Then the first sentence can be eliminated.

A5. Let L and R be the left and right sides of a sentence, and Δ - a substitution. Then a new sentence

$$k\ L // \Delta \Rightarrow R // \Delta$$

can be inserted inmediately before the original sentence.

Notice that all we need to apply these rules is the GPA. Relation $L_2 \subseteq L_1$ takes place when $L_2 \cap L_1 = L_1$, i.e. in projecting L_1 on L_2 , no contractions are needed in L_2 for recognition to be possible.

An Example. Suppose we have the following definition of a recursive predicate α:

.1 $\alpha\ s_1 \Rightarrow F$

.2 $\alpha\ s_1\ s(+\ -)_2 \Rightarrow F$

.3 $\alpha\ s_1\ s_2 \Rightarrow F$

.4 $\alpha\ s(+\ -)_1\ s_2 \Rightarrow T$

.5 $\alpha\ s_1\ s_2\ s_3 \Rightarrow F$

.6 $\alpha\ s_1\ s_2\ s_3\ s_4 \Rightarrow \beta\ e_5$

The sentence .4 will never be used because it is *screened* by sentence .3 (Rule A3). Therefore we eliminate it. Now, the sentence .2 is *submerged* by the sentence .3 in accordance with Rule A4, and we eliminate it too. It is easy to see that all the sentences left are transposable (Rule A2), thus we can rewrite the algorithm in this manner:

$$\begin{aligned}&\alpha\ s_1\ s_2\ s_3\ s_4\ e_5 \Rightarrow \beta\ e_5\\ &\alpha\ s_1 \Rightarrow F\\ &\alpha\ s_1\ s_2 \Rightarrow F\\ &\alpha\ s_1\ s_2\ s_3 \Rightarrow F\end{aligned}$$

Using Rule A1, we add at the end one more sentence:

$$\alpha\ e_1 \Rightarrow F$$

which expands the domain of the function α by including the cases when its argument is either <empty>, or contains parentheses. But it was not this domain expansion that we aimed at. It became possible now to *submerge* (Rule A4) the preceding three sentences into the last. As a result the algorithm is greatly simplified:

$$\begin{aligned}&\alpha\ s_1\ s_2\ s_3\ s_4\ e_5 \Rightarrow \beta\ e_5\\ &\alpha\ e_1 \Rightarrow F\end{aligned}$$

It should be stressed that unlike Rule A3, rules A2 and A4 are applicable only if one of the sentences is *immediately* followed by the other. For example, in the algorithm

$$\begin{aligned}&\beta\ A \Rightarrow T\\ &\beta\ s_1 \Rightarrow F\\ &\beta\ s_1\ e_2 \Rightarrow T\end{aligned}$$

the first sentence cannot be submerged by the third.

The system of five rules A1-A5 is not complete; it only gives the most useful transformation rules. The incompleteness of this system can be seen from the following example. Consider this definition:

$$\begin{aligned}&\alpha\ s_a\ e_1 \Rightarrow A\\ &\alpha\ (e_1)\ e_2 \Rightarrow B\\ &\alpha \Rightarrow C\\ &\alpha\ e_x \Rightarrow D\end{aligned}$$

The fourth sentence will never be used, because every expansion begins with either a symbol or a parenthesis, or else is empty. Therefore, this sentence may be eliminated, but this cannot be done using rules A1-A5 only. In Section 4.3 we shall present an algorithm which performs transformations of this kind.

3.4 Functional Equivalence.

By the *domain of a function* F we mean the set M of those object expressions E_0 , for which the process of concretization of the expression $k\ F\ E_0 \perp$ will be brought by the Refal machine to a normal end. Note the difference between this definition and the definition of the domain of an algorithm. Speaking of an algorithm, we mean the process itself irrespective of whether it is finite or not, hence the only reason for an expression E_0 to be outside the domain is to result in an abnormal stop (recognition impossible). Speaking of a function, we mean the result of a process, thus if the concretization will never end, the expression E_0 is outside the domain.

Let an algorithm A define inter alia a function F. We shall say that an algorithm A' is *functionally equivalent* to A with respect to the function F if for every E_0 from the domain of F, the concretization of $k\ F\ E_0 \perp$ with the algoright A' leads to a normal end and gives the same result as with the algorithm A. We shall speak of *strict* functional equivalency, if this relation holds for any object expression E_0, and the two domains, therefore, are identical. As in the case of algorithmic equivalency, we will be interested in and formulate the rules for simple (*nonstrict*) functional equivalence. It should be noted that nonstrict equivalence (both algorithmic and functional) is not *a relation of equivalency* in the sense the term is used in mathematics, because it is not symmetric. But it is reflexive and transitive: a quasi-ordering relation.

Clearly rules A1-A5 of algorithmic equivalence are applicable for functionally equivalent transformations. Two additional rules, F1 and F2, specific for functional equivalence, will be formulated below. We shall call them the rules of *driving*. Their main idea is to execute one or more steps of the Refal machine in a situation where the expression under concretization is not completely defined, but contains free variables. The expressions containing free variables are taken, of course, from the right sides of the sentences; we are as if "driving" some expressions from the right sides through the left sides in order to execute one step of a Refal machine in a general form — hence the nickname of the operation.

F1. (*Rule of driving*) Let one of the sentences defining a function F be of the form:

$$\text{F.X} \qquad k\ F\ L_f \Rightarrow C_1\ k\ G\ E_p \perp C_2$$

where E_p is a pattern expression. C_1 is a left and C_2 a right multibracket, and G is the determiner of a function which is defined by the sentences:

$$\text{G.1} \qquad k\ G\ L_g^1 \Rightarrow R_g^1$$

$$\text{G.2} \qquad k\ G\ L_g^2 \Rightarrow R_g^2$$

. . . .

$$\text{G.n} \qquad k\ G\ L_g^n \Rightarrow R_g^n$$

Using the Generalized Projecting Algorithm we find n sets of substitutions $<\Delta_1^i,\ldots,\Delta_{r_i}^i>$, i=1,2,...,n, which specify n intersections

$$E_p \cap L_g^i = \bigcup_{j=1}^{n} E_p \mathbin{//} \Delta_j^i , \qquad i=1,2,\ldots,n.$$

Then we can replace the sentence F.X by $r_1 + r_2 + \ldots + r_n$ sentences of the form

$$k\ L_f \mathbin{//} \Delta_j^i \Rightarrow \{C_1 \mathbin{//} \Delta_j^i\}\ R_g^{ij}\ \{C_2 \mathbin{//} \Delta_j^i\}$$

which are arranged in the order of increasing i . Here R_g^{ij} are obtained from R_g^i through the replacement of the free variables by those values they take when $E_p \mathbin{//} \Delta_j^i$ is recognized as L_g^i. We note that if for some i the number of subclasses r_n is one, and substitution Δ_1^i is trivial (which means that $E_p \subseteq L_g^i$), then by virtue of Rule A3, all the sentences with greater i can be omitted.

Proof: As a result of the application of the sentence F.X in the process of concretization, a term of the form

$$k\ G\ E_0 \perp$$

will enter the view-field of the Refal machine as one of its subexpressions. Here E_0 is some object expression. Until the concretization sign k in this term becomes *active*, the term, whatever it is, will not influence the work of the Refal machine in any way. When the sign k becomes active, the Refal machine will start the next step by trying to recognize E_0 first as L_g^1 , then, if recognition is impossible, as L_g^2 , etc. We do not know E_0 exactly, but we know that $E_0 \subseteq E_p$, and this is something. Using the GPA to recognize E_p as L_g^1 , we come to one of the following three cases:

F1.1. $E_p \subseteq L_g^1$, therefore every E_0 belonging to E_p will be recognized as L_g^1 . Consequently, we can, anticipating the action of the Refal machine, replace the term k $G\ E_p \perp$ in F.X by the expression R_g^{11} , obtained from R_g^1 by substituting the values (which are generally pattern expressions with free variables from E_p) assigned to the free variables in L_g during the process of projecting.

F1.2. $E_p \cap L_g^1 = \emptyset$. Recognition of E_0 as L_g^1 is impossible, and we proceed to the second sentence.

F1.3. $E_p \cap L_g^1$ is the union of r_1 subclasses of E_p obtained through substitutions Δ_j^1. Since the variables of E_p are defined in the left side L_f, the conditions of Theorem 3.6 are satisfied, and L_f // Δ_j^1 for $j = 1,2,\ldots,r_1$, will be nonoverlapping L-classes. Using Rules A5 and A2, we transform sentence F.X into $r_1 + 1$ sentences:

$$k\ F\ L_f\ //\ \Delta_1^1 \Rightarrow C_1\ k\ G\ E_p \perp C_2\ //\ \Delta_1^1$$

$$\cdots\cdots\cdots\cdots$$

$$k\ F\ L_f\ //\ \Delta_{r_1}^1 \Rightarrow C_1\ k\ G\ E_p \perp C_2\ //\ \Delta_{r_1}^1$$

$$\text{F.X}\quad k.\ F\ L_f \Rightarrow C_1\ k\ G\ E_p \perp C_2$$

Now, since

$$E_p\ //\ \Delta_j^1 \subseteq L_g^1\ ,\quad \text{for } j = 1,2,\ldots,r_1$$

we will have case F1.1 for each of the new r_1 sentences and we can make the corresponding substitutions in the right sides.

We have isolated all the subclasses of L_f for which any E_0 belonging to E_p can be recognized as L_g^1, which makes sentence G.1 applicable. Therefore, sentence F.X will be used now for those E_0 only for which sentence G.1 proved unapplicable. Thus we can continue the transformation of sentence F.X ignoring G.1 and starting the step by trying to recognize $G\ E_p$ as the left side of the second sentence G.2. Repeating this procedure, we will come ultimately either to the case F1.1, which ends driving, or to the end of the definition of function G. In the last case, the group of sentences which has resulted from the original sentence F.X due to driving will still have sentence F.X at the end. But it can be dropped now. Indeed, this sentence will be, possibly, used with the transformed algorithm A' for those $E_0 \subseteq E_p$ only, for which no sentence in the original algorithm A proved applicable to concretize k G $E_p \perp$, thus bringing the Refal machine to an abnormal stop. This means that whatever function has demanded this concretization, its argument was out of its domain, and the transformed algorithm may produce any result. We have completed the proof of the correctness of Rule F1.

Widening of the domain as a result of the use of Rule F1 will be shown in the following example. Let functions /F1/ and /F2/ be defined by the algorithm

F.X k/F1/ s_1 s_1 ⇒ k/F2/ s_1 $\perp$
F.2 k/F1/ e_1 ⇒ e_1
G.1 k/F2/ A ⇒ A
G.2 k/F2/ (e_1) e_2 ⇒ e_1 e_2

Let us drive the function call k/F2/ s_1 $\perp$ in sentence F.X. Driving it through the first sentence G.1, we get one subclass, obtained by the contraction $s_1 \to$ A. The second sentence G.2 produces no subclasses where recognition is possible ($r_2 = 0$). As a result, we receive the following definition of the function /F1/:

F.1 k/F1/ A A ⇒ A
F.2 k/F1/ e_1 ⇒ e_1

and the definition of function /F2/ is not needed any more. Now, according to **F.1-2**, function /F1/ is defined everywhere on the set of expressions. According to the original definition, it was not defined on the set of all double symbols with the exception of double A. If we retained sentence F.X, we would derive the definition:

F.X1 k/F1/ A A ⇒ A
F.X k/F1/ s_1 s_1 ⇒ k/F2/ s_1 $\perp$
F.2 k/F1/ (e_1) e_2 ⇒ e_1 e_2

which is strictly equivalent to the original, but we would not be able to eliminate the **definition** of function /F2/.

The following rule provides a generalization of Rule F1.

F2. Suppose one of the sentences describing a function F is of the form:

$$k\ F\ L_f \Rightarrow C_1\ k\ G\ E \perp C_2$$

where E is an arbitrary (general) expression. We replace in E every term which begins with the sign k and is not itself situated in the range of another sign k by a new e-variable different from all the others. If the sentence thus modified allows the application of the rule F1 under the condition that in all the substitutions Δ^i_j no one of the auxiliary e-variables is affected by contractions, then we can use F1, and afterwards replace the auxiliary e-variables by the terms they represent and perform in these the necessary substitutions.

This rule adds more to "nonstrictness" of functional equivalence. Now the original function may lead for some arguments to the nonstop situation, while the transformed function will have this argument inside its domain. Consider the following example:

1. $k/F/\ e_1 \Rightarrow k/BEGPAR/\ (k/X/e_1 \perp) \perp$
2.1 $k/BEGPAR/\ (e_1)\ e_2 \Rightarrow T$
2.2 $k/BEGPAR/\ e_1 \Rightarrow F$
3.1 $k/X/\ {*}e_1 \Rightarrow k/X/\ {*} \perp$
3.2 $k/X/\ e_1 \Rightarrow e_1$

We can use Rule F2 to drive the call of function /X/ in the definition of function /F/. Then its definition reduces to one simple sentence:

$$k/F/ \Rightarrow T$$

Since the argument of /BEGPAR/ in the original sentence 1 will always begin with a parenthesis whatever the result of concretization of the function /X/ call is, sentence 2.1 will always be used giving T as the result — but concretization of the /BEGPAR/ call will start only after, and if, concretization of $k/X/e_1 \perp$ has been successfully finished. If it is infinite, which occurs for the arguments e_1 beginning with an asterisk, function /F/, like function /X/ will be undefined. Using Rule F2 we greatly simplify function /F/, but pay the price of expanding its domain.

3.5. Iterative Usage of Driving

The problems which can be expressed in terms of symbol strings or trees composed of such strings (expressions) can be conveniently formulated in Refal. Repeatedly using equivalent transformations we can find partial or complete solutions of such problems, the leading role being played of course by the rules of driving. We shall illustrate this by two examples.

(1) The problem is: what must the string X be in order for the composite string ABXBX to be symmetric?

The property of a string being symmetric can be defined by the recursive predicate σ:

S.1 $\sigma \Rightarrow T$

S.2 $\sigma\ s_1 \Rightarrow T$

S.3 $\sigma\ s_1\ e_2\ s_1 \Rightarrow \sigma\ e_2$

S.4 $\sigma\ e_1 \Rightarrow F$

Now our problem can be formulated as finding all those arguments of the predicate α:

A $\alpha\ e_1 \Rightarrow \sigma\ AB\ e_1\ B\ e_1$

for which it takes the value T. We shall transform sentence A by using rule F1 repeatedly.

At the first step of driving sentences S.1 and S.2 prove to be inapplicable. We proceed to project the left side of S.3 on the expression to be driven (the notation will be the same as in 3.2).

$\mathcal{E}_\ell = s_1\ e_2\ s_1$, $\mathcal{E}_p = A\ B\ e_1\ B\ e_1$.

4.1 $s_1 := A$

$\mathcal{E}_\ell = e_2\ A$, $\mathcal{E}_p = B\ e_1\ B\ e_1$.

3.4*. (1) $e_1 \rightarrow$ <empty>, $\mathcal{E}_p = B\ B$ X(1)

3.4*. (1) $e_1 \rightarrow e_1\ A$, $e_2 := B\ e_1\ A\ B\ e_1$

Recognition is attained by isolating one subclass. This gives us the sentence

A.1 $\alpha e_1 A \Rightarrow \sigma\ A\ B\ e_1\ A\ B\ e_1\ A$

which we insert before sentence A. By driving in A.1 we immediately transform it into

A.1 $\alpha\ e_1\ A \Rightarrow \sigma\ Be_1\ ABe_1$

Continuing driving in sentence A we transform it into

A $\alpha\ e_1 \Rightarrow F$

Translating these two sentences into English, our first result is: the argument e_1 must end with A. Now we drive in A.1.

$\mathcal{E}_\ell = s_1 e_2 s_1$, $\mathcal{E}_p = Be_1 ABe_1$

4.1 $s_1 := B$

$\mathcal{E}_\ell = e_2\ B$, $\mathcal{E}_p = e_1\ AB\ e_1$

3.4*. (1) $e_1 \rightarrow$ <empty>,

$\mathcal{E}_\ell = e_2\ B$, $\mathcal{E}_p = AB$

3.1*. $\mathcal{E}_\ell = e_2$, $\mathcal{E}_p = A$,

2. $e_2 := A$

Recognition attained. We return to (1).

3.4*. (1) $e_1 \rightarrow e_1\ B$

3.1*. $\mathcal{E}_\ell = e_2$, $\mathcal{E}_p = e_1\ BAB\ e_1$

2. $e_2 := e_1\ BAB\ e_1$.

Again, recognition is attained. Thus, we have received two subclasses, and we insert two new sentences:

A.1.1 $\alpha\ A \Rightarrow \sigma\ BAB$

A.1.2 $\alpha\ e_1\ BA \Rightarrow \sigma\ Be_1\ BAB\ e_1\ B$

Continuing driving in A.1 we transform it into

A.1 $\alpha\ e_1\ A \Rightarrow F$

Using the algorithmic equivalence rule A4 with respect to sentences A.1 and A we eliminate sentence A.1. Making straightforward driving in sentences A.1.1 and A.1.2 we receive the following description of α:

$$\alpha\ A \Rightarrow T$$
$$\alpha\ e_1\ BA \Rightarrow \sigma\ e_1\ BAB\ e_1$$
$$\alpha\ e_1 \Rightarrow F$$

If the definition of a predicate begins with a number of sentences whose right sides are T the left sides of these sentences define a list of classes, which is a partial solution to the problem. By now, we have obtained only one class, consisting of the only expression A. Driving in the second sentence we obtain this description of α:

$$\alpha\ A \Rightarrow T$$
$$\alpha\ BA \Rightarrow T$$
$$\alpha\ s_2\ BA \Rightarrow T$$
$$\alpha\ s_2\ e_1\ s_2\ BA \Rightarrow \sigma\ e_1\ s_2\ BA\ s_2\ e_1$$
$$\alpha\ e_1 \Rightarrow F$$

We have obtained two new classes: BA and s_2 BA. One more driving will give additional classss $s_2\ s_2$ BA and $s_2\ s_3\ s_2$ BA. This process can be carried on indefinitely. The set of all object expressions E_0 for which concretization of $\alpha\ E_0 \perp$ gives T is not a class. It is easy to see that the described procedure gives the exact decomposition of this set into an infinite union of nonoverlapping classes.

(2) Let us consider the function of addition of binary numbers defined by the following sentences:

P.1 $k + (e_x)\ (\ ) \Rightarrow e_x$

P.2 $k + (\ )\ (e_y) \Rightarrow e_y$

P.3 $k + (e_x s_1)\ (e_y\ 0) \Rightarrow k + (e_x)\ (e_y) \perp s_1$

P.4 $k + (e_x\ 0)\ (e_y\ 1) \Rightarrow k + (e_x)\ (e_y) \perp 1$

P.5 $k + (e_x\ 1)\ (e_y\ 1) \Rightarrow k + (k + (e_x)\ (1) \perp)\ (e_y) \perp 0$

The predicate of equality will be defined in this way:

E.1 $k = (\ )\ (\ ) \Rightarrow T$

E.2 $k = (e_x\ s_1)\ (e_y\ s_1) \Rightarrow k = (e_x)\ (e_y) \perp$

E.3 $k = e_1 \Rightarrow F$

We put the question: with the first argument of the function + being 101, what must the second argument be for the sum to be equal to 1011 ? The predicate α which is to be transformed to obtain the answer to this question is defined by the sentence:

A $\alpha\ e_x \Rightarrow k = (k + (101)\ (e_x) \perp)\ (1011) \perp$

In transforming this sentence we shall use a strategy of driving, which may be called *from without within.*(*) It is this. In an attempt to make a simplifying transformation, we start with driving the subexpression delimited by the first from the left concretization sign in the right side of the sentence and the conjugated concretization point. If there are no concretization signs in this subexpression, or they can be ignored according to Rule F2, we complete the driving according to Rule F1, and this is the end of the first rtep of transformation. If some of the auxiliary e-variables, formed according to Rule F2, should be found to require contraction, we take the first of them and try to drive the corresponding subexpression, applying the same principles as in the previous attempt. Obviously, sooner or later we shall find a drivable subexpression: in the worst case it will be the one deliminated by the leading concretization sign and the conjugated concretization point.

In our case, we first try to drive the call of the equality function, and find out that it is impossible because of the k-sign in its first argument, which is a plus function call. Therefore, we drive on this call. Sentence P.1 is applicable with the contraction $e_x \rightarrow$ <empty>. This gives us the sentence

$$\alpha \Rightarrow k = (101)\ (1011) \perp$$

which we immediately transform into

$$\alpha \Rightarrow F$$

In transforming predicates, it is very convenient to put, when it is possible, the sentences with F in the right side in the last places of the definition, through the use of the rule

(*) Also to be referred to as inside from outside, or outside-in strategy.

of transposition A2. Then we add at the end of the sentence

$$\alpha\ e_x \Rightarrow F$$

by virtue of Rule A1, and eliminate the F sentences by using Rule A4. In our case all new sentences will be transposable, so we shall keep track only of the sentences with the right side different from F.

Sentence P.2 proves to be inapplicable. Sentence P.3 generates the sentence

$$\alpha\ e_x\ 0 \Rightarrow k = (k + (10)\ (e_x) \perp 1)\ (1011) \perp$$

Using the strategy *from without within* for this sentence, we drive the equality function call, which gives:

A.1 $\quad \alpha\ e_x\ 0 \Rightarrow k = (k + (10)\ (e_x) \perp)\ (101) \perp$

Continuing the driving of the initial sentence A, we find sentence P.4 inapplicable, and sentence P.5 gives

$$\alpha\ e_x\ 1 \Rightarrow k = (k + (k + (10)\ (1) \perp)\ (e_x) \perp 0)\ (1011) \perp$$

Again, driving it inside from outside we obtain:

$$\alpha\ e_x\ 1 \Rightarrow F$$

and lose interest in it.

Thus, we have obtained only one sentence A.1 for further transformation. In fact, we have performed the first step of the usual algorithm of subtraction from right to left. Our general strategy formulated in terms of Refal notions produced a familiar algorithm in this particular case. If we continue the transformation, we finally get:

$$\alpha\ 110 \Rightarrow T$$
$$\alpha\ 0110 \Rightarrow T$$
$$\alpha\ e_x \Rightarrow F$$

which gives a complete formal answer to the problem in clear form.

It is worth noting that we derived two solutions instead of one, because we did not introduce the equivalence of numbers which differ by leading zeros. As the problem is defined, there are exactly two solutions. The number 00110 if added to 101 gives 01011, not 1011 .

4.1. Formulation of the Problem

What does it mean to define the semantics of an algorithmic language? The most direct definition is 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. Now we shall 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 (identifying 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 A 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.

When defining a language through its interpreting function, we do not give an explicit definition of the set of correct (legitimate) texts in that language. Instead, it is natural to introduce the following definition: a pair of expressions *A* and *E* is called *a correct text-object pair in language L* , if concretization of (1) does not lead to an abnormal stop of the Refal machine (note: a nonstop situation is allowed). Now the set of correct texts *A* may be defined as: a text *A* is correct if the set of those work objects *E* that make a correct pair with *A* is not empty. If we have an independent definition of the set of correct work objects then we can give an alternative definition of the correct text: a text is correct if it makes a correct pair with any correct work object.

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?

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

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

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. Chapter 3 of the present work provides us with the necessary apparatus. We imagine that the following expression is put in the view-field of the Refal machine:

$$k\ /L/\ \mathrm{CROSS(CAT)}\ ;\ \mathrm{ADD(DOG)}\ (e_x) \perp$$

(which is, of course, impossible literally and must be understood as *a set* of view-fields) and use the rules of driving to follow the process of concretization. Since we defined driving for the strict Refal only, we should rewrite the definition of /L/ in the corresponding way:

L. $\quad k/L/\ e_x \quad \Rightarrow k\ /LL/\ (\ )\ e_x \perp$

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

LL.2 $\quad k/LL/(e_1)s_x\ e_2(e_a) \quad \Rightarrow k/LL/(e_1s_x)\ e_2(e_a)\ |$

LL.3 $\quad k/LL/(e_1)(e_a) \quad \Rightarrow k/L1/\ e_1(e_a) \perp$

(The definition of /L1/ remains unchanged.)

A number of initial steps of the Refal machine is done without contractions of free variables, i.e. with *any* input data. This is the part of the job which is performed once and forever *at compilation time*. Then we receive the following view-field:

$$k\ /L/\ \mathrm{ADD(DOG)}\ (k\ /L1/\ \mathrm{CROSS(CAT)}\ (e_x) \perp) \perp$$

Contraction $e_x \to s_ie_x$ is needed here according to Rule F1, which means that a conditional statement depending on unknown input data must be added to the program for M^0. 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

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

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. As in dealing with equivalent transformations, the theory should be applicable to any texts in Refal, and therefore, the goal of the theory is to design *one universal algorithm* to compile from any language, had its interpreting functions been defined in Refal. This algorithm may have variations, though. In particular, it must vary from one object machine M^0 to another.

The object machine should have facilities for symbol manipulation, since this is what the Refal machine is doing. In addition, it may have any specific opeations, however complicated, and these may be even undefinable in Refal, such as generation of a random number. The only requirement is that the universal compiler (*supercompiler*) could recognize the corresponding Refal expressions as external function calls, and translate them into the standard notation for M^0.

In particular, M^0 may be the Refal machine. Then as the result of compilation we get a program in Refal again! The

(*) In fact, both alternatives in this statement lead to the same result, but to discover it one has to use Rule A4 , not just Rule F1.

expression (2) can be introduced as the right side of the sentence defining a new function:

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

Then compilation amounts to an equivalent transformation of this function, or more exactly, the theory of compilation provides a new class of equivalent transformations, which cannot result from simple application of the rules described in Chapter 3.

In our case:

$$\alpha\ e_x \Rightarrow k\ /L/\ \mathrm{CROSS(CAT);\ ADD(DOG)}\ (e_x)\ \perp$$

As the result of compilation (which in this particular case is nothing more than an iterative application of Rule F1 and algorithmic equivalency rules) we get

$$\alpha\ s_1\ s_2\ e_x \Rightarrow \mathrm{C}\ s_1\ \mathrm{A}\ s_2\ \mathrm{T}\ e_x\ \mathrm{DOG}$$
$$\alpha\ s_1\ e_x \Rightarrow \mathrm{C}\ s_1\ \mathrm{A}\ \mathrm{T}\ e_x\ \mathrm{DOG}$$
$$\alpha\ e_x \Rightarrow \mathrm{CAT}\ e_x\ \mathrm{DOG}$$

This definition is shorter and is executed much faster than the original definition using function /L/. Thus, optimization is one of the aspects of the theory of compilation.

In the expression (2), metasymbol A represents some definite expression, therefore the argument of the function L is *partially* (but not *completely*, because of the variable e_x) specified. The problem to be solved by the theory of compilation is to eliminate the redundancy of the general definition of the function L in the circumstances when we need it only in a specific context. From this formulation, one can see that the structure of the expression (2) has actually no significance; the only important thing is that it is more specific than the general format of the function L. Also, it is of no significance that function L has been introduced as the interpreting function of a language; it may have any meaning. Removing these preconditions, we get the most general formulation of the problem.

The method of solving this problem may be described (informally, for the time being) in this way. Let there be an algorithm given, and a general expression in the view-field of the Refal machine. We consider this expression as *a generalized state* of the Refal machine, and *map* the generalized states of the Refal machine onto the generalized states of the object machine M^0. When we execute each next step of the Refal machine, its state changes, which generates a *graph of states*. We shall trace this group and *model* it on M^0. That is, compile such a program for M^0, that if the Refal machine and the machine M^0 with corresponding initial states work in parallel, then their states will remain corresponding.

4.2. Graph of States.

The workable expression in the view-field of the Refal machine will be called its *exact state*, for it uniquely determines all the consequent states of the Refal machine (recall that we use *strict* Refal, so that digging and burying are not allowed). A set of exact states will be called *a generalized state*. Our main concept in describing generalized states will be *a configuration*, by which we mean a set of workable expressions produced when the free variables of a general Refal expression assume some values. Thus a configuration is defined by (or *as*) a general expression. If the free variables in this expression may assume all syntactically permitted values, we refer to the resulting generalized state as a *full* or *unrestricted* configuration; if the values of the free variables are somewhow restricted, we call this state a *restricted* configuration.

The dynamics of the Refal machine with a given memory field may be represented by its *graph of states*, which is essentially a *graph of configurations*. The vertices of this graph represent generalized states of the Refal machine, and the arcs (directed edges) represent basic relations between them. There are three types of arcs, which we shall be introducing in the course of exposition. The first type is *a dynamic arc*, which depicts a possible transition from one generalized state to another resulting from one or more steps of the Refal machine. The condition when the transition occurs will be indicated on the arc as a contraction of free variables in the original configuration. The contractions on the dynamic arcs of the graph of states are essentially the left sides of Refal sentences. To model the Refal machine with the help of a graph of states, we apply the contractions to specific values of free variables in exactly the same way as we are applying left sides. For example, the contraction

$$e_1 \to s_2 e_1$$

is not only a conditional statement, which determines whether or not the value of e_1 starts with a symbol, but it also assigns this symbol to s_2 and redefines e_1 correspondingly. Contractions on the arcs in a graph serve as definitions of new variables appearing in the subsequent states.

The dynamic arcs outgoing from the same vertex are ordered in conformity with the use by the Refal machine of different sentences. We shall picture dynamic arcs by more or less horizontal solid lines in the order from top to bottom. The states will be numbered, and for each number the corresponding configuration will be given in *the list of configurations* accompanying the graph.

A configuration may be either *active*, if it includes at least one concretization sign k, or *passive*, if this is not the case. Active configurations will be depicted as circles, and passive as squares, with the number of state inside. There will be two standard designations:

$$\boxed{} \quad \text{and} \quad \boxed{?}$$

representing the empty configuration and the state *recognition impossible* of the Refal machine. Each graph of states will have as a starting point a *full* configuration; the corresponding state will be called *the start* of the graph. During the construction of the graph of states, we will be applying the rules of driving to some configurations, thereby exploring their evolution in time. These configurations (more precisely, the corresponding vertices, i.e. the states) will be referred to as *explored*, and the others as *unexplored*, these notions being applicable of course only to active configurations. The formal indication that a vertex is explored is the presence of at least one outgoing dynamic arc. If it is found for this configuration that the recognition is impossible, we use a dynamic arc leading to vertex $\boxed{?}$ as mentioned above.

A finite graph of states without unexplored vertices will be called *complete*.

Theorem 4.1. Let S be the start configuration of a complete graph of states G. Then for every exact state from S we can find all the subesquent states of the Refal machine using only graph G.

At the present time we are proving this theorem for the case when graph G consists only of vertices and dynamic arcs, and we shall complete the demonstration later. Replace free variables in the start configuration by their values. We shall be able to choose uniquely one of the dynamic arcs which originate from the start, or else establish that recognition is impossible. In the former case we come to the next exact state. If it is passive this is a normal stop of the Refal machine, if it is active it is explored, and we continue to apply the same procedure.

Our goal in the theory of compliation is to know how to construct complete graphs of states. Let us start with examining this process on the following example. The algorithm in the memory-field is:

$$\alpha\ e_1 \rightarrow \beta(\ )\ e_1$$
$$\beta\ \ (e_1)\ /PLUS/\ e_2 \rightarrow \beta(e_1\ +)\ e_2$$
$$\beta\ (e_1)\ s_2 e_3 \rightarrow \beta(e_1 s_2)\ e_3$$
$$\beta\ (e_1) \rightarrow e_1$$

And the initial configuration is

(1) α A /PLUS/ e_1 /PLUS/ B $\perp$

Executing three steps of driving we get the following graph (Figure 1):

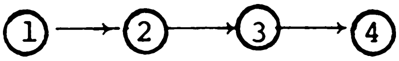


Figure 1

with the configurations:

(2) β( ) A /PLUS/ e_1 /PLUS/ B $\perp$

(3) β(A) /PLUS/ e_1 /PLUS/ B $\perp$

(4) β(A+) e_1 /PLUS/ B $\perp$

The vertices from which only one dynamic arc without contractions originates, such as 2 and 3 in Figure 1, will be called *transitory*. A transitory vertex different from the start vertex (this restriction is to avoid confusion with the identification of the initial state) may be removed from the graph of states. We shall remove transitory vertices in this example, and will do so later on, if the opposite is not explicitly stated.

A straightforward application of Rule Fl to our graph transforms it into Figure 2.

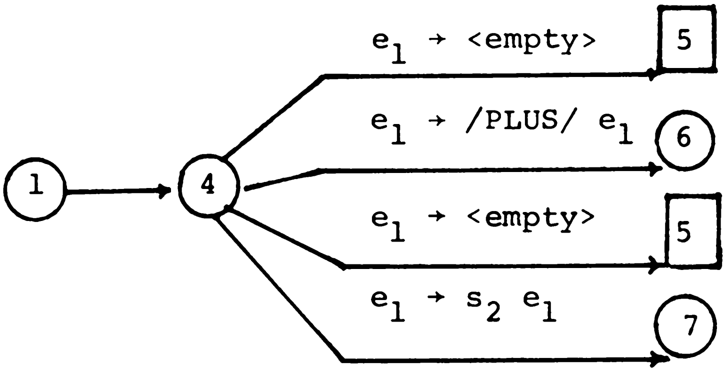


Figure 2

The list of new configurations is:

(5) $A + + B$

(6) $\beta(A + +)\ e_1$ /PLUS/ $B \perp$

(7) $\beta(A + s_2)\ e_1$ /PLUS/ $B \perp$

The third arc outgoing from state 4 will never be used, because it is screened by the first arc with the same contraction. To see more clearly how the graph of states transformations correlate with equivalent transformations of Refal programs as formulated in Chapter 3, we shall at this point start outlining the procedure of mapping the graph on the Refal machine.

Each configuration of the source Refal machine M^s will be mapped on a configuration of the object Refal machine M^0, which has the special form:

$k\ C^n\ (V_1)\ (V_2)\ \ldots\ (V_m) \perp$ or $k\ P^n\ (V_1)\ (V_2)\ \ldots\ (V_m) \perp$

where n is the sequence number of the configuration, m is the number of (different) free variables, and V_i are these free variables. C^n stands for an active, and P^n for a passive configuration. While a configuration is yet unexplored, its definition in Refal will consist of one sentence, the left side of which is its notation in M^0 terms and the right side is the defining expression in M^s terms. For instance, the starting point in our case is:

$$k\ C^1\ (e_1) \Rightarrow \alpha\ A \text{ /PLUS/ } e_1 \text{ /PLUS/ } B \perp$$

Performing driving, we first of all express the right side of the sentence in terms of M^0 configurations. As a result, the sentences defining an explored configuration will have both sides expressed in M^0 terms, thus representing not a condition of correspondence between the states of M^s and M^0, but a dynamic transformation taking place inside M^0. When there are no unexplored configurations (a complete graph), we get a completed program for M^0.

The stage of work represented in Figure 1 corresponds to the following program

$k\ C^1\ (e_1) \Rightarrow k\ C^4\ (e_1) \perp$
$k\ C^4\ (e_1) \Rightarrow \beta\ (A\ +)\ e_1\ /PLUS/\ B \perp$

At the stage of Figure 2 we get:

$k\ C^1\ (e_1) \Rightarrow k\ C^4\ (e_1) \perp$

C4.1 $k\ C^4\ (\ ) \Rightarrow k\ P^5 \perp$

$k\ C^4\ (\ /PLUS/\ e_1) \Rightarrow k\ C^6\ (e_1) \perp$

C4.3 $k\ C^4\ (\ ) \Rightarrow k\ P^5 \perp$

$k\ C^4\ (s_2\ e_1) \Rightarrow k\ C^7\ (e_1)\ (s_2) \perp$

$k\ C^5 \Rightarrow A\ +\ +\ B$

$k\ C^6\ (e_1) \Rightarrow \beta(A\ +\ +)\ e_1\ /PLUS/\ B \perp$

$k\ C^7\ (e_1)\ (s_2) \Rightarrow \beta\ (A\ +\ s_2)\ e_1\ /PLUS/\ B \perp$

We want to make the object code in Refal as efficient as possible, so before continuing exploration we will drive configuration C^4 in the definition of C^1, thus eliminating C^4 altogether. First, this reduces by one the number of steps necessary to co cretize C^1; second, this has some additional implications, which will be discussed later. Also we simplify the definition of C^4 using the rule of screening (A3) to C4.3 and C4.1 to eliminate C4.3.

$k\ C^1\ (\ ) \Rightarrow k\ P^5 \perp$

$k\ C^1 (/PLUS/e_1) \Rightarrow k\ C^6\ (e_1) \perp$

$k\ C^1 (s_2\ e_1) \Rightarrow k\ C^7\ (e_1)\ (s_2) \perp$

If we execute one more step of driving in Figure 2 we get the graph in Figure 3, with the new configurations:

(8) $A\ +\ +\ +\ B$

(9) $\beta(A\ +\ +\ +)\ e_1\ /PLUS/\ B \perp$

(10) $\beta(A\ +\ +\ s_3)\ e_1\ /PLUS/\ B \perp$

(11) $A + s_2 + B$

(12) $\beta(A + s_2 +)\ e_1$ /PLUS/ $B \perp$

(13) $\beta(A + s_2\ s_3)\ e_1$ /PLUS/ $B \perp$

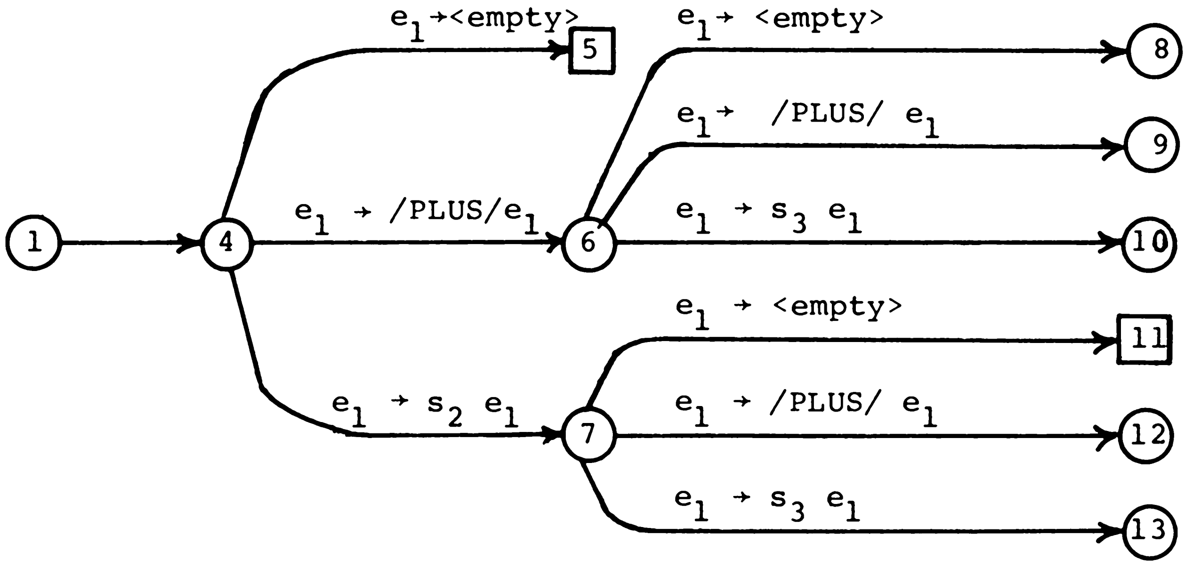


Figure 3

Mapping this graph onto the object Refal machine M^0, we again perform "M^0 to M^0" driving. Configurations C^6 and C^7 disappear, and we receive:

$$\begin{aligned}
&k\ C^1\ (\ ) && \Rightarrow k\ C^5 \perp \\
&k\ C^1\ (\text{/PLUS/}) && \Rightarrow k\ P^8 \perp \\
&k\ C^1(\text{/PLUS/ /PLUS/}e_1) && \Rightarrow k\ C^9\ (e_1) \perp \\
&k\ C^1(\text{/PLUS/ } s_3\ e_1) && \Rightarrow k\ C^{10}(e_1)\ (s_3) \perp \\
&k\ C^1(s_2) && \Rightarrow k\ P^{11}\ (s_2) \perp \\
&k\ C^1(s_2\text{/PLUS/ } e_1) && \Rightarrow k\ C^{12}\ (e_1)\ (s_2) \perp \\
&k\ C^1(s_2\ s_3\ e_1) && \Rightarrow k\ C^{13}\ (e_1)\ (s_2)\ (s_3) \perp
\end{aligned}$$

There are as many sentences here as there are paths on the graph, which start from C^1 and end at an unexplored or passive configuration. The left side of each sentence is the composition of the contractions which appear on all arcs of the corresponding path.

It is easy to see that we can proceed in this way ad infinitum. But we need a *complete* graph, which is, first of all, finite. To accomplish this, we shall use the method of *generalization.* Comparing all active configurations in our graph we perceive that with the exception of C^1, all are of the form

$$\beta\ (E)\ e_1\ /PLUS/\ B \perp$$

where E is different for different configuration expressions. Let us introduce a *generalizing configuration*

(14) $$\beta\ (e_2)\ e_1\ /PLUS/\ B \perp$$

that is such a configuration that all the others could be produced from it by substitution. For instance, C^{12} can be produced from this configuration by the substitution $e_2 \rightarrow A+s_2+$. Note that there is a great deal of arbitrariness in this choice. There is always the possibility of taking $\beta\ e_x \perp$ as a generalization. But because of the reason to be presented later, too "sweeping" a generalization does not lead to efficient programs. On the other hand, there is a more specific configuration in our case, which could also be taken as a generalization, namely,

$$\beta\ (A + e_2)\ e_1\ /PLUS/\ B \perp$$

but the first configuration is easier to discover and it leads to the same result with respect to efficiency.

Now we represent configuration C^4 as a special case of C^{14}, i.e. configuration C^{14} where variable e_2 has taken the value A+. On the graph of states, we shall draw *a representation arc*, depicted by dashes, from C^4 to C^{14} (see Figure 4). The substitution expressing the values of the new configuration through the values of the original will be indicated as an assignment statement on the arc. But instead of the usual form

$$e_x := E$$

we used earlier, we shall write this assignment in the form

$$E \leftarrow e_x$$

This form, which may seem strange at first glance, is in fact 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 substituting 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 valiables 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
Assigmment	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.

We have already used one form:

$$E \mathbin{//} V \rightarrow E'$$

The other form, completely equivalent from a syntactical point of view, is

$$E' \leftarrow V \mathbin{//} E$$

(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 substitu tions will be different for contractions and assignments, although equally easily suggested by our representation:

$$(V \rightarrow L^1)\ (V \rightarrow L^2) = V \rightarrow (L^1 \mathbin{//} V \rightarrow L^2)$$
$$(E^1 \leftarrow V)\ (E^2 \leftarrow V) = (E^1 \leftarrow V \mathbin{//} E^2) \leftarrow V$$

A variable like e_2 in Figure 4, which is introduced by an assignment statement, will be called *a generalization variable.* When we model the concretization process using the graph of states, all generalization variables will take definite values expressed in terms of the values of free variables and constant expressions. But in the process of exploration (when constructing the graph) we treat them as if they were free: hence generalization.

So, when we derive, for the first time in the process of driving, configuration (4), we come over to configuration (14) (see Figure 4 below). This procedure will be called *submission*: we will say that we *submitted* C^4 to C^{14}. Then we continue to drive C^{14}, and find out that the two active configurations we receive on the next step, C^{15} and C^{16}, also can be submitted to C^{14}, thus making the graph in Figure 4 *complete*: there are no unexplored states any more.(*)

(*) Sometimes we shall use a small square □ instead of <empty>.

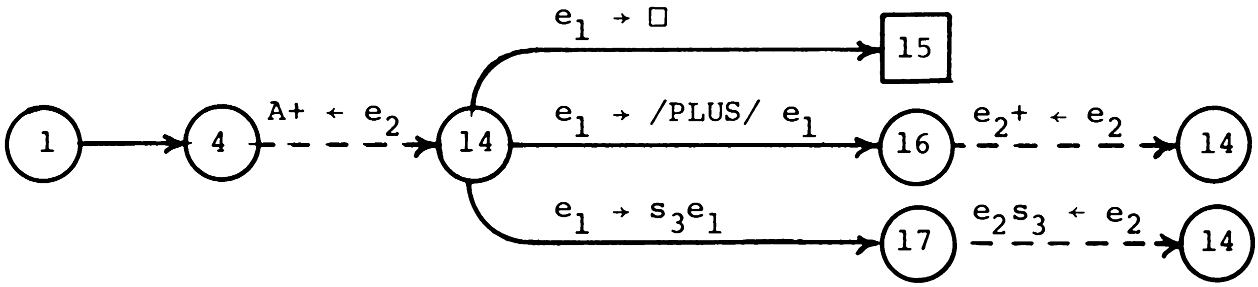


Figure 4

(15) e_2 + B

(16) $\beta(e_2$ +) e_1 /PLUS/ B $\perp$

(17) $\beta(e_2\ s_3)$ e_1 /PLUS/ B $\perp$

Mapping this graph on the object Refal machine and returning to passive configurations in the final result their natural form, we derive:

$$
\begin{array}{ll}
k\ C^{1}\ (e_1) & \Rightarrow k\ C^{14}\ (A+)\ (e_1) \perp \\
k\ C^{14}\ (e_2)\ (\ ) & \Rightarrow e_2 + B \\
k\ C^{14}\ (e_2)\ (/PLUS/e_1) & \Rightarrow k\ C^{14}\ (e_2\ +)\ (e_1) \perp \\
k\ C^{14}\ (e_2)\ (s_3e_1) & \Rightarrow k\ C^{14}\ (e_2s_3)\ (e_1) \perp
\end{array}
$$

Hurrah! We have brought our first process of compilation to a successful end. Let us evaluate the result. It is fairly good. In the original program, the string to be processed consisted of three parts: A /PLUS/ was the first, e_1 the second, and /PLUS/ B the third. Only the second, middle part was unknown; the other two (which might have been much longer, of course) could be processed beforehand in order to optimize the algorithm. This is just what the compilation has accompliseed. The new algorithm stores the processed first part, then processes the unknown part, and upon completion, adds on at the end the processed third part. The algorithm of processing the unknown part is exactly the same (disregarding format differences) as the original one.

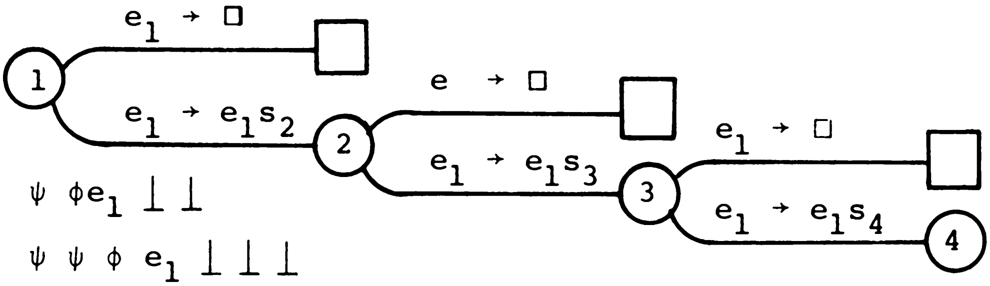
Let us consider another example. This is the program:

$$\begin{array}{lll} \phi & \Rightarrow & \\ \phi\ e_1\ s_2 & \Rightarrow & \psi\ \phi\ e_1 \perp \perp \\ \psi\ e_1 & \Rightarrow & \end{array}$$

and this is the initial configuration:

(1) $\psi\ \phi\ e_1 \perp \perp$

After several steps of driving we come to the graph of states shown in Figure 5:



(1) $\psi\ \phi e_1 \perp \perp$

(2) $\psi\ \psi\ \phi\ e_1 \perp \perp \perp$

(3) $\psi\ \psi\ \psi\ \phi\ e_1 \perp \perp \perp \perp$

(4) $\psi\ \psi\ \psi\ \psi\ \phi\ e_1 \perp \perp \perp \perp \perp$

Figure 5

Obviously, this graph can be continued endlessly, and it cannot be made finite by submission of configurations (2), (3), etc., to a generalization: there is no generalizing configuration of these configurations, because they have different numbers of concretization signs. To represent such situations on the graph of states we introduce the third (and the last) type of arc: *a composition arc*, which will be depicted by a vertical "wavy" line. Using this device we construct a graph of states shown in Figure 6:

(1) $\psi\ \phi\ e_1 \perp \perp$

(2) $\psi\ e_x \perp$

(3) $\phi\ e_1 \perp$

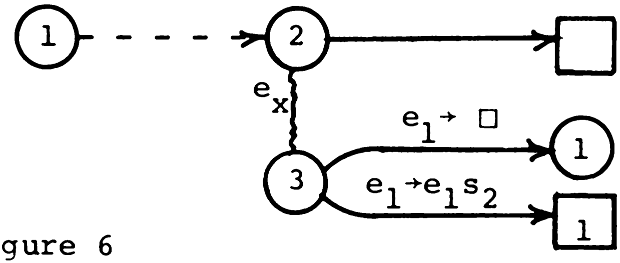
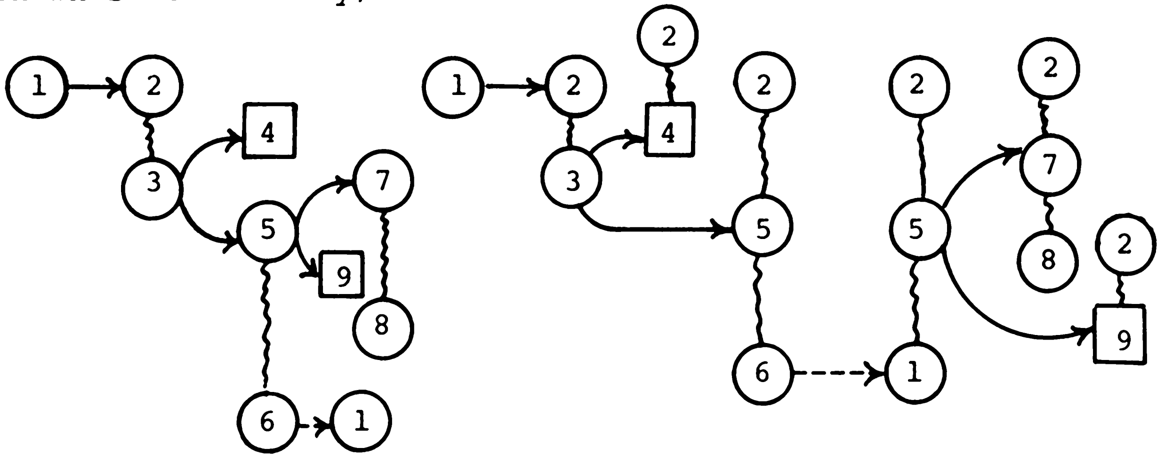


Figure 6

Now, with the introduction of composition arcs, the full state of the Refal machine (exact or generalized) is characterized not by a single vertex, but, generally, by a vertical row of vertices connected by composition lines. When we proceed to explore the lower (internal) configuration, the upper (external) one does not go anywhere: it stays put. It is only for the sake of convenience that we do not represent external parts in trailing the fate of the internal parts, and this becomes possible because the external part remains unchanged (except for possible contractions) until the transformation of the internal part is completed. So, the graph shown in Figure 7a stands actually for what is depicted in Figure 7b (both graphs are showh schematically).



(a) (b)

Figure 7

Note that unlike external parts, all the internal parts of a configuration do participate in the transformations, therefore when an upper vertex is explored, the lower part disappears, as can be seen in Figure 7b: transitions C^5 to C^7 and C^5 to P^9.

Like two other types of arc, a composition arc represents a substitution, but this time it is the substution of the *result of concretization*, for the *computed variable* of the tail-vertex indicated at the arc (e_x in Figure 6). Computed variables are absent in the original configuration. When we *decompose* configuration (1) in Figure 6 into the composition of (2) and (3), we introduce a variable, e_x , which, like a generalization variable, is *not free* to assume any syntactically allowed value: its value will be uniquely determined by the values of free variables in (3) after the concretization of (3) is performed. But when we explore the *external* (tail vertex) configuration, we treat the computed variable as if it were free. This is why decomposition is also a type of generalization — the external configuration represents a wider set of exact states than the set of those states which are actually possible at that moment. Generalization variables and computed variables will be referred to as *redundant* variables.

We had no choice as to how to decompose configuration (1) into two configurations, because it contained only two concretization signs. In the general case, when there are more than two concretization signs, there is some freedom in selecting the *internal* configuration, in other words, the subexpression to be driven first. The most reasonable way is to use the *from without within* strategy mentioned in Section 3.5. After the selected subexpression has been replaced by a computed variable, the external configuration may again be decomposed in two, using the same startegy. Another possibility is always to select for driving the subexpression beginning with the *leading* sign k.

Now we can complete the proof of Theorem 4.1. The rules of construction of representational and compositional arcs, like those of dynamic arcs, are such that the variables in the head-vertex of a representation arc assume unique values if the variables of the tail-vertex have taken definite values, and the noncomputed variables in both ends of a composition arc assume unique values if the variables of the tail-vertex of the preceding dynamic or representation arc have taken definite values.

The dynamic transition from a generalized state which is not at the bottom of a vertical segment is, normally, conditional, depending on the value of the computed variable. When we model the Refal machine with the help of the graph of states, we will face the problem of making this conditional jump only after the lower part of the segment has been concretized, and therefore the computed variable has taken on a definite value. Hence, dynamic transitions will be as unique in the case of full states represented by vertical segments as they are with full states represented by single vertcies. Thus, at every step of the Refal machine we will know its exact state, which completes the proof.

4.3. Clean Graphs

Let us sum up what the graph of states is.

—The vertices of a graph of states are (generalized) states of the Refal machine, but not every possible state of the Refal machine may be represented by a single vertex. Generally, a state of the Refal machine is a *vertical segment*, i.e. a sequence of vertcies connected by composition arcs.

—With each vertex a configuration is associated, which is, generally, restricted in accordance with the location of the vertex in the graph. Each vertex is identified by its number. More than one vertex may be characterized by the same configuration.

—Generally, a graph of states is not a tree, but it is convenient to represent it as a tree, some terminal vertcies of which may be identical to (have the same number as) a non-terminal (explored) vertex. We assume now that the graph of states will always be represented in the form of a tree(even if it is pictured with loops), because this simplifies dealing with graphs. We shall call a *path* in a graph a sequence $V_1A_1V_2A_2...V_{k-1}A_{k-1}V_k$, whose terms are alternately vertcies

V_i and arcs A_i , such that $k > 1$ and for $1 \leq i < k$ the arc A_i leads from V_i to V_{i+1} , and all the vertices are distinct from each other.

—There are arcs of three types in the graph of states. Dynamic and representation arcs will be occasionally called *horizontal*, as opposed to vertical composition arcs. Vertices may be called active or passive, corresponding to the configuration they belong to. From an active vertex either one representation arc or several dynamic arcs must originate. Dynamic arcs are ordered. From a passive vertex no horizontal arcs may originate. Independently of horizontal arcs, one vertical arc may or may not originate from a vertex. In a vertical segment, a horizontal arc may lead only to its topmost vertex.

—Dynamic arcs bear contractions, representation arcs bear assignments. Assignments are, at the same time, generalizations. Let assignment $E \leftarrow V$ be borne by an arc leading from C^n to C^m. Then variable V in C^m has the full scope of values, i.e. represents the set of all syntactically allowed values. Those variables which are not indicated in the right side of one of the assignments have the same meaning as in the preceding configuration. Contractions and assignments define new variables or redefine old ones. The variables defined by assignments will be referred to as *generalization variables*. A *computed* variable is defined in a vertex if it is borne by the composition arc originating from this vertex. When this configuration gets explored, the computed variable is treated as a free variable having its full scope of values. Free variables appearing in the start configuration are, *ipso facto*, defined, and have their full scopes of values. Only those variables may appear in a configuration which have been in some of these ways defined on the path from the start to this configuration.

—An active vertex is *explored*, if on the path from it to a terminal (in the tree representation) vertex there is at least one dynamic arc. A graph is *complete* if all terminal vertices are either passive or identical to one of the explored vertices.

We shall introduce now several new definitions. *Input variables* in a graph of states are the free variables of the start configuration. Input variables, generalization variables and computed variables will be referred to as *quasiinput variables*. An *exact quasiinput state* is specified when a value of each quasiinput variable is specified. A *quasiinput set* is a set of exact quasiinput states. In particular, a quasiinput set may be a *quasiinput class*; it is specified when contractions (possibly trivial) are specified for each quasiinput variable. A quasiinput class can be represented by a single pattern expression if we choose a way of combining contractions for all the variables into one expression. We shall write a quasiinput class for the case of n ordered quasiinput variables in the form

$$(L_1)\ (L_2)\ \ldots\ (L_n)$$

where L_i are the right sides of contractions for the variables.

To each exact quasiinput state a *terminal* path (that is a path ending with a terminal vertex) corresponds : the one taken by the Refal machine if the quasiinput variables are assigned corresponding values. We shall say that the exact quasinput state *takes* this path and all its *subpaths* (paths which are parts of the terminal path). To each path in the graph a quasiinput set corresponds which comprises all exact quasiinput states taking this path. The same set corresponds to the vertex which ends this path. A path is called *feasible* if the corresponding quasiinput set is not empty, otherwise it is *unfeasible*. A graph in which there are no unfeasible paths will be called *clean*.

To refer to a specific path in a graph of states, we shall represent it as a sequence of vertex numbers, separated by the following symbols: a comma "," representing a dynamic arc; an equality sign "=" representing a representation arc; a bracket "[" representing a composition arc. If different dynamic arcs lead to vertices characterized by the same configuration (or just to draw attention to a specific arc), we may indicate the arc by placing the contraction borne by it in parentheses before the corresponding comma, e.g. $6(e_y \rightarrow T),\ 8$.

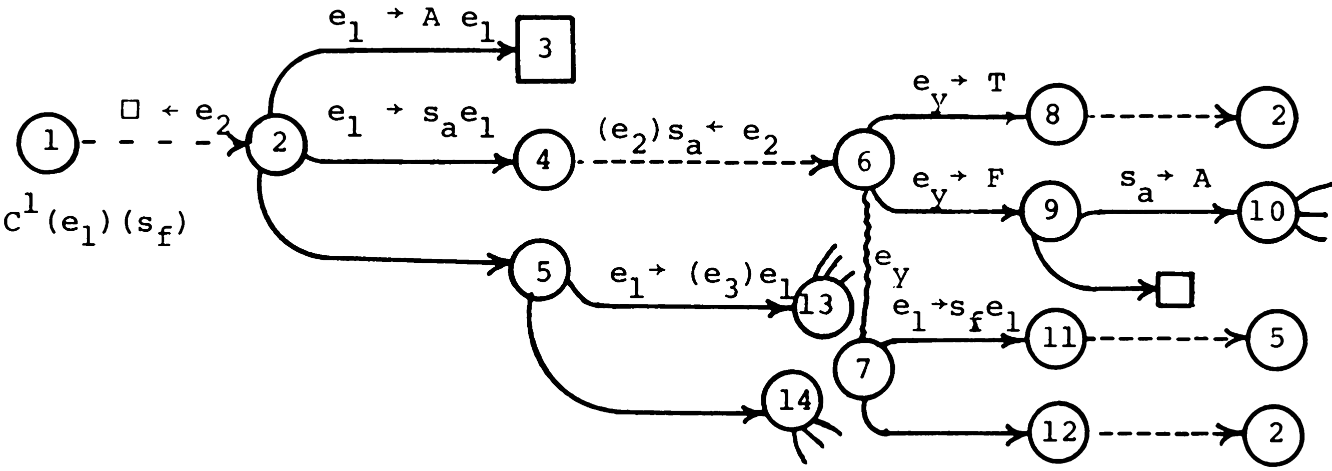


Figure 8

In Figure 8 the path leading to vertex 11 is 1 = 2, 4 = 6 [7, 11 . In configuration (11) the following variables may appear: e_1 and s_f as input variables, indicated, to make it clear, at the start vertex (1); e_2 defined as a generalization variable at the path 1 = 2 and redefined at 4 = 6; s_a defined by the contraction on the path 2, 4. Input variable e_1 is subject to two contractions, and we can combine them into one contraction: $e_1 \to s_a s_f e_1$. This is possible because between the two contractions there is no redefinition of e_1. If the arc 4 = 6 bore assignment $(e_2)s_a \leftarrow e_1$, it would give back to e_1 its fully generality, and the contraction for e_1 at vertex 11 would be simply $e_1 \to s_f e_1$.

We shall join input variables into a composite expression as shown in the argument of configuration C^1 in Figure 8 adding newly defined redundant variables in the order of their appearance. What is the quasiinput set corresponding to the path leading to vettex 11 ? It may seem at first glance that it is the class $(s_a s_f e_1)(s_f)(e_2)$. In fact, it is a subset of this class, because of the restriction on the variable s_a resulting from the position of vertex 4 in the graph: s_a should be distinct from A, otherwise arc 2,3 will be chosen by the Refal machine. Thus, configuration (4) is an example of a restricted configuration. Restrictions on variables may produce unfeasible paths if they are not taken into account during the process of graph construction. In Figure 8 the path leading to vertex 9 is feasible, but adding to it arc 9, 10 we get an unfeasible path: contraction s → A is impossible because of the above-mentioned restriction. Consider configuration (5). It is also restricted. Variable e_1 cannot begin with a symbol. This fact may be taken into account in drawing arcs originating from vertex 5. There is no arc corresponding to the impossible contraction, and at the vertex 14 the only possible value for e_1 is <empty>, which should be reflected in treating vertex 14, to simplify the graph.

Now let us discuss, using Figure 8 as an example, the problems we encounter trying to *submit* a new configuration to an already existing one. Suppose we try to submit vertex 11 to vertex 5, as shown in the figure. We must answer two questions. The first is: if we consider configurations corresponding to vertices 11 and 5 as *full* configurations, is it true that configuration 5 is more (or equally) general than 11 ? To answer this question we have to find out whether there exists a substitution which changes configuration 5, being applied to it, to configuration 11. This is a more special problem than that for which the Generalized Projecting Algorithm was designed. We do not need to find intersections; we only want to know whether one patterh expression can be recognized as the other. The algorithm for this problem is applicable to any pair of pattern expressions and is an obvious generalization of the projecting algorithm in Section 2.2, in which one allows free variables in an "object" expression and treats them as unknown wholes.

If we found configuration (5) to be no less general than (11), we must answer the second question: are not the restrictions on the variables in the earlier vertex more severe than the restirctions at the current point? From this point of view, we should not unconditionally submit vertex 11 to vertex 5, because the variable e_1 as redefined at 11 may have its full scope of values, while at vertex 5 it is restricted. But it may happen that for some reason we are sure that the actual scope of e_1 is limited to what it is at 5; then the submission will be legitimate. If we do not have such (rather extraordinary) information, we should either explore vertex 11 as an independent configuration, or generalize vertex 5 by making a seemingly trivial assignment $e_1 \leftarrow e_1$, and reexplore it. If all variables in a vertex have their full scopes, this vertex may be made the start of a separate subgraph, not connected with the main graph (and other subgraphs) in any other way than by submissions.

To deal with quasiinput sets we need some additional means of representing sets of object expressions. First we define eight types of the simplest contractions which will be called *elementary*. They are:

(1) $s(P)_i \rightarrow S$ where $S \in P$

(2) $s(P)_i \rightarrow s(Q)_i$ where $Q \subset P$

(3) $s(P)_i \rightarrow s(P)_j$ where $i \neq j$

(4) $e_i \rightarrow$ <empty>

(5) $e_i \rightarrow s'_j e_i$ where s'_j is new

(6) $e_i \rightarrow (e'_j) e_i$ where e'_j is new

(7) $e_i \rightarrow e_i\ s'_j$ where s'_j is new

(8) $e_i \rightarrow e_i\ (e'_j)$ where e'_j is new

As in Chapter 3, unspecified s-variables are considered here as having a special "any symbol" specifier which comprises the infinite set of all symbols. Primed variables s'_j and e'_j stand for variables which are distinct from any already introduced variable. They are, in fact, *generators* of variables, the index of the newly generated variable being assigned as the value to the index of the generator variable.

It is easy to see that all contractions which appear in using the Generalized Projecting Algorithm (see Section 3.2) are either elementary or compositions of elementary contractions. For example, a contraction of the form $e_i \to S\ e_i$ can be represented as

$$(e_i \to s_j'\ e_i)\ (s_j \to S)\ ,$$

where contractions are to be applied from left to right.

Theorem 4.2. A composition of any number of elementary contractions is a contraction to an L-expression. Conversely, any L-expression can be represented as the composition of a number of elementary contractions applied to an e-variable.

Proof. There are no elementary contractions which would change one e-variable into another, and only new e-variables may be introduced. This means that no repeated e-variables may result from a composition of elementary contractions. New e-variables emerging due to elementary contractions are always at a different level of parenthesis structure, therefore no pair of e-variables on the same level may appear. So, both requirements to L-expressions are satisfied, which proves the first part of the theorem. To prove the second part, we only have to notice that applying the GPA to project any given L-expression on a single variable e_1 , one receives a representation of this L-expression in the form $e_1\ //\ \Delta$, where Δ is a composition of elementary contractions.

By $A \setminus B$ we denote *the difference* between sets A and B, that is the set of all elements of A which are not at the same time elements of B . As usual, $A \setminus B \setminus C$ means $(A \setminus B) \setminus C$ etc. If the definition of a function in Refal consists of sentences with left sides $L_1, L_2, \ldots, L_n$, then the set of all object expressions for which the k-th sentence will be used is

$$(1) \qquad L_k \setminus L_1 \setminus L_2 \setminus \ldots \setminus L_{k-1}$$

We call this set *a restricted class*; expressions $L_1, L_2, \ldots, L_{k-1}$, which are negative, will be called *restrictions* on the class L_k.

It is very difficult to establish from a record of such a form whether the set is empty or not. As the first step to amend the situation, we replace restrictions on the *class* by restrictions on the *variables*. Using the GPA we find k-1 intersections:

$$L_i \cap L_k = \bigcup_{j=1}^{r_i} L_k // \Delta_i^j , \qquad i = 1,2,\ldots k-1.$$

where Δ_i^j are contractions of the variables in L_k. Now we represent our restricted class in the form

(2) $L_k \backslash\backslash \Delta_1^1 \backslash\backslash \Delta_1^2 \ldots\backslash\backslash \Delta_1^{r_1} \backslash\backslash \Delta_2^1 \ldots\backslash\backslash \Delta_{k-1}^{r_{k-1}}$

where Δ_i^j will be referred to as *restrictions on the variables* in L_k. To determine whether a given object expression belongs to this restricted class we first recognize it as L_k. If we have succeeded in that, we find out whether the values assigned to the variables in L_k allow at least one of the contractions Δ_i^j. If it is the case, the object expression does not belong to the set, otherwise it does belong.

Operation \\ (read: "restriction"), applicable to a set E of object expressions and a substitution Δ , produces, like operation // (substitution), a subset $E \backslash\backslash \Delta$ of the set E, but unlike the case of substitution this subset is not generally a class when E is a class.

The expression

$$E \backslash\backslash \Delta_1 \backslash\backslash \Delta_2$$

means

$$(E \backslash\backslash \Delta_1) \backslash\backslash \Delta_2$$

The operation of restriction is commutative in the sense:

$$E \backslash\backslash \Delta_1 \backslash\backslash \Delta_2 = E \backslash\backslash \Delta_2 \backslash\backslash \Delta_1$$

For the operation of substituting a simple rule holds with respect to the composition of substitutions:

$$E // (\delta_1 \delta_2 \ldots \delta_n) = E // \delta_1 // \delta_2 \ldots // \delta_n$$

which is true by the definition of composition. For restriction the corresponding rule is more complicated:

$$\text{(RC)} \qquad E \setminus\setminus (\delta_1 \delta_2 \delta_3 \ldots \delta_n) = E \setminus\setminus \delta_1 \cup (E // \delta_1) \setminus\setminus \delta_2 \cup (E // \delta_1 // \delta_2) \setminus\setminus \delta_3 \ldots \cup (E // \delta_1 // \delta_2 \ldots \delta_{n-1}) \setminus\setminus \delta_n$$

We shall refer to this rule as RC (Restriction-Composition) and use it for the case when δ_i are elementary contractions. The rule permits decomposing a class restricted by any contraction into a union of classes restricted by single elementary contractions. Now, the following distributive law holds, obviously, with respect to union and restriction:

$$\text{(UR)} \quad (E_1 \cup E_2 \cup \ldots \cup E_n) \setminus\setminus \Delta = (E_1 \setminus\setminus \Delta) \cup \ldots \cup (E_n \setminus\setminus \Delta)$$

We shall refer to this law as Rule UR (Union-Restriction). It is easy to see that by algebraic manipulation based on Rules RC and UR we can represent any restricted class (1) (represented first in the form (2) through the use of the GPA) in the form:

$$L_k \Sigma^1 \cup L_k \Sigma^2 \cup \ \ldots \cup L_k \Sigma^n$$

where Σ^i are compositions

$$\Sigma^i = \sigma_1^i \sigma_2^i \ldots \sigma_p^i$$

of "*constrictions*" (i.e. contractions or restrictions) σ_j^i:

$$\sigma_j^i \text{ is } // \delta_j^i \text{ or } \setminus\setminus \delta_j^i$$

where δ_j^i are elementary contractions.

For example, let a restricted class X be written in the form (2) as:

$$X = L \setminus\setminus \Delta_1 \setminus\setminus \Delta_2$$

where

$$\Delta_1 = \delta_1 \delta_2 \delta_3$$

$$\Delta_2 = \delta_4 \delta_5$$

We proceed as follows (using + instead of $\cup$):

$$L \setminus\setminus \Delta_1 = L \setminus\setminus \delta_1 + L // \delta_1 \setminus\setminus \delta_2 + L // \delta_1 // \delta_2 \setminus\setminus \delta_3$$

The set X will be the sum (union) of the following three sets:

$$(L \backslash\backslash\ \delta_1) \backslash\backslash\ \Delta_2 = L \backslash\backslash\ \delta_1 \backslash\backslash\ \delta_4 + L \backslash\backslash\ \delta_1 // \delta_4 \backslash\backslash\ \delta_5$$

$$(L // \delta_1 \backslash\backslash\ \delta_2) \backslash\backslash\ \Delta_2 = L // \delta_1 \backslash\backslash\ \delta_2 \backslash\backslash\ \delta_4 + L // \delta_1 \backslash\backslash\ \delta_2 // \delta_4 \backslash\backslash\ \delta_5$$

$$(L // \delta_1 // \delta_2 \backslash\backslash\ \delta_3) \backslash\backslash\ \Delta_2$$
$$= L // \delta_1 // \delta_2 \backslash\backslash\ \delta_3 \backslash\backslash\ \delta_4 + L // \delta_1 // \delta_2 \backslash\backslash\ \delta_3 // \delta_4 \backslash\backslash\ \delta_5$$

A composition product $\Sigma^i = \sigma_1^i\ \sigma_2^i\ \ldots\ \sigma_p^i$ of constrictions will be referred to as a *constriction term*. When we start algebraic manipulation with a class E somehow restricted, all further transformations and subset formations will produce unions (sums) of the form

$$E\ \Sigma^1 \cup E\ \Sigma^2 \cup \ldots \cup E\ \Sigma^n$$

Therefore, in formulating the rules of manipulation we can omit arbitrary E and consider *constriction sums*

$$\Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^n$$

or

$$\Sigma^1 + \Sigma^2 + \ldots + \Sigma^n$$

For example, Rule (RC) may be written as

$$\backslash\backslash\ (\delta_1\ \delta_2\ \ldots\ \delta_n) = \backslash\backslash\ \delta_1 \cup // \delta_1 \backslash\backslash\ \delta_2 + // \delta_1 // \delta_2 \backslash\backslash\ \delta_3 + \ldots + // \delta_1 // \delta_2\ \ldots // \delta_{n-1} \backslash\backslash\ \delta_n$$

Our task now is to develop algebraic rules for simplification of constriction terms. We note first that if a term begins with a number of "positive" contractions (contractions proper) we can simply perform the corresponding substitutions in E. If we want the result not in the form of the union of restricted classes but in the form of a constriction sum applied to the original E, we shall remember the corresponding contractions for each term and add them at the left side. Thus whenever we are concerned with simplification, the term begins with a "negative" contraction (restriction):

(A) $\backslash\backslash\ \delta_1\ \sigma_2\ \sigma_3\ \ldots\ \delta_p$

Second, we notice that restrictions, unlike contractions, are commutative. Therefore we can organize the process of transformation in the following way. At the beginning of the sequence being transformed we shall always have a group of restrictions which have been processed already:

$$\{\backslash\backslash\ \delta_1\ \backslash\backslash\ \delta_2\ \ldots\ \backslash\backslash\ \delta_m\}\ \sigma_{m+1}\ \ldots\ \sigma_p$$

If σ_{m+1} involves a variable which is not involved in the group of processed restrictions, then it is commutative. We either apply a substitution δ_{m+1} to E if σ_{m+1} is a contraction $//\ \delta_{m+1}$, or treat the restriction as if it were in the first place in the sequence: case (A). In particular, no action may be taken other than adding $\backslash\backslash\ \delta_{m+1}$ to the list of processed restrictions.

Suppose now that the replaced variable in σ_{m+1} is identical to the variable in one of the processed restrictions $\backslash\backslash\ \delta_x$. Two cases are to be considered:

(B) $\{\backslash\backslash\ \delta_1\ \ldots\ \backslash\backslash\ \delta_x\ \ldots\ \backslash\backslash\ \delta_m\}\ //\ \delta_{m+1}\ \ldots\ \sigma_p$

(C) $\{\backslash\backslash\ \delta_1\ \ldots\ \backslash\backslash\ \delta_x\ \ldots\ \backslash\backslash\ \delta_m\}\ \backslash\backslash\ \delta_{m+1}\ \ldots\ \sigma_p$

In both cases we can formulate rules which involve only δ_x and δ_{m+1}, but in case (B) we must remember that $//\ \delta_{m+1}$ does not commute with $\backslash\backslash\ \delta_x$ generally; therefore to transfer $//\ \delta_{m+1}$ to the beginning we must compare it to *all* $\backslash\backslash\ \delta_x$ involving the same variable. In case (C) it will not be an error just to add $\backslash\backslash\ \delta_{m+1}$ to the list, but again we must try all possible pairs $\backslash\backslash\ \delta_x\ \backslash\backslash\ \delta_{m+1}$ if we want a maximum of simplification.

There are three groups of rules to manipulate constrictions, which correspond to cases (A), (B), and (C). The rules of the first group are *elimination rules*; they are applicable to a single restriction, no matter whether there are other restrictions for the same variable or not. The rules of the second and third groups are correspondingly applicable to pairs *restriction-contraction* and *restriction-restriction*.

For our purpose it is necessary now to treat separately the cases of a finite specifier, i.e. of a variable of the form $s(P)_i$, and of the "any symbol" specifier, i.e. of an unspecified varibble s_i . Therefore, instead of three elementary contractions involving an s-variable, we will have six elementary contractions; this brings the full number of elementary contractions to eleven. In what follows we describe not only the rules of transformation, but the algorithm of their application (with some freedom of variety). We shall always make full lists of possible cases, and if no simplification is possible indicate this as "no action".

A rule of the form

$$\Sigma_1 = \Sigma_2 \cup \Sigma_3$$

should be understood as: with any E

$$E\ \Sigma_1 = E\ \Sigma_2 \cup E\ \Sigma_3$$

A. Elimination Rules

A.1 $\backslash\backslash\ s_i \rightarrow s(Q)_i$ no action

A.2 $\backslash\backslash\ s(P)_i \rightarrow s(Q)_i = //\ s(P)_i \rightarrow s(P \backslash Q)_i$

A.3 $\backslash\backslash\ s_i \rightarrow S$ no action

A.4 $\backslash\backslash\ s(P)_i \rightarrow S = //\ s(P)_i \rightarrow s(P \backslash \{S\})_i$

A.5 $\backslash\backslash\ s_i \rightarrow s_j$

A.6 $\backslash\backslash\ s(P)_i \rightarrow s(P)_j$ no action

A.7.L $\backslash\backslash\ e_i \rightarrow \Box = (//\ e_i \rightarrow s'_j\ e_i) \cup (//\ e_i \rightarrow (e'_j)\ e_i)$

A.7R $\backslash\backslash\ e_i \rightarrow \Box = (//\ e_i \rightarrow e_i\ s'_j) \cup (//\ e_i \rightarrow e_i\ (e'_j))$

A.8 $\backslash\backslash\ e_i \rightarrow s'_j\ e_i = (//\ e_i \rightarrow \Box) \cup (//\ e_i \rightarrow (e'_j)\ e_i)$

A.9 $\backslash\backslash\ e_i \rightarrow (e'_j)\ e_i = (//\ e_i \rightarrow \Box) \cup (//\ e_i \rightarrow s'_j\ e_i)$

A.10 $\backslash\backslash\ e_i \rightarrow e_i\ s'_j = (//\ e_i \rightarrow \Box) \cup (//\ e_i \rightarrow e_i\ (e'_j))$

A.11 $\backslash\backslash\ e_i \rightarrow e_i\ (e'_j) = (//\ e_i \rightarrow \Box) \cup (//\ e_i \rightarrow e_i\ s'_j)$

Elimination rules are applied as soon as a negative term is located. Either of the rules A.7L or A.7R may be used, and a clever algorithm may make a guess as to which choice will be more expedient. Also, it is possible to take no action at all, and we shall include this possibility in subsequent considerations.

So, after applying elimination rules we face a situation when only five out of eleven restrictions may take the place of δ_x in (B), and both δ_x and δ_{m+1} in (C). Now, for a shorter formulation of rules B and C, we again unite cases 5 and 6 (as listed in the Elimination Rules) into one case $s(\mathcal{P})_i \rightarrow s(\mathcal{P})_j$, and in addition consider case 3 as a special variation of case 1, by considering symbol S as $s'(\mathcal{Q})_j$ with $\mathcal{Q} = \{S\}$.

B. Restriction-Contraction Rules

B.1 $\backslash\backslash\ s_i \rightarrow s(\mathcal{Q})_i\ //\ s_i \rightarrow s(\mathcal{Q}')_i = //\ s_i \rightarrow s(\mathcal{Q}' \setminus \mathcal{Q})_i$

B.2 $\backslash\backslash\ s_i \rightarrow s(\mathcal{Q})_i\ //\ s_i \rightarrow s_j = //\ s_i \rightarrow s_j\ \backslash\backslash\ s_j \rightarrow s(\mathcal{Q})_j$

B.3 $\backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_j\ //\ s(\mathcal{P})_i \rightarrow s(\mathcal{Q})_i$

$= (//\ s(\mathcal{P})_i \rightarrow s(\mathcal{Q})_i\ \backslash\backslash\ s(\mathcal{P})_j \rightarrow s(\mathcal{Q})_j)$

$\cup\ (//\ s(\mathcal{P})_i \rightarrow s(\mathcal{Q})_i\ //\ s(\mathcal{P})_j \rightarrow s(\mathcal{Q})_j\ \backslash\backslash\ s(\mathcal{Q})_i \rightarrow s(\mathcal{Q})_j\ )$

B.4 $\backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_j\ //\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_k =$

if k = j *then* $\emptyset$ *else*

$//\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_k\ \backslash\backslash\ s(\mathcal{P})_k \rightarrow s(\mathcal{P})_j$

B.5 $\backslash\backslash e_i \rightarrow \Box\ //\ e_i \rightarrow \Box = \emptyset$

B.6-9 $\backslash\backslash e_i \rightarrow \Box\ //\ e_i \rightarrow \mathcal{E} = //\ e_i \rightarrow \mathcal{E}$

where $\mathcal{E}$ is one of: $s_j'e_i$, $(e_j')e_i$, e_is_j', $e_i(e_j')$.

C. Restriction-Restriction Rules.

C.1 $\backslash\backslash\ s_i \rightarrow s(\mathcal{Q})_i\ \backslash\backslash\ s_i \rightarrow s(\mathcal{Q}')_i = \backslash\backslash\ s_i \rightarrow s(\mathcal{Q} \cup \mathcal{Q}')_i$

C.2 $\backslash\backslash\ s_i \rightarrow s(\mathcal{Q})_i\ \backslash\backslash\ s_i \rightarrow s_j$ no action

C.3 $\backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_j\ \backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{Q})_i$

if $\mathcal{P}$ is "any symbol", *then* no action *else* use Rule A.2 for the second restriction; then use B.3.

C.4 $\backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_j\ \backslash\backslash\ s(\mathcal{P})_i \rightarrow s(\mathcal{P})_k$ no action ($k \neq j$ is implied)

C.5 $\backslash\backslash\ e_i \rightarrow \Box\ \backslash\backslash\ e_i \rightarrow \Box = \backslash\backslash\ e_i \rightarrow \Box$

C.6 $\backslash\backslash\ e_i \rightarrow \Box\ \backslash\backslash\ e_i \rightarrow s_j'e_i = //\ e_i \rightarrow (e_j')\ e_i$

C.7 $\backslash\backslash\ e_i \rightarrow \square\ \backslash\backslash\ e_i \rightarrow (e'_j)\ e_i = //\ e_i \rightarrow s'_j\ e_i$

C.8 $\backslash\backslash\ e_i \rightarrow \square\ \backslash\backslash\ e_i \rightarrow e_i\ s'_j = //\ e_i \rightarrow e_i\ (e'_j)$

C.9 $\backslash\backslash\ e_i \rightarrow \square\ \backslash\backslash\ e_i \rightarrow e_i\ (e'_j) = //\ e_i \rightarrow e_i\ s'_j$

Whenever in any of the rules an empty specifier appears in a "positive" contraction, the corresponding term in the union must be set empty.

There is one additional rule:

D. Rule of Symmetry.

D. $\backslash\backslash\ s(P)_i \rightarrow s(P)_j = \backslash\backslash\ s(P)_j \rightarrow s(P)_i$

This rule is used for restriction $\backslash\backslash\ \delta_x$, when constriction σ_{m+1} has in its left side variable $s(P)_j$.

Because of the eliminatrion rules, only the following four types of restrictions may enter the list of processed restrictions:

(1) $\backslash\backslash\ s_i \rightarrow s(Q)_i$

(2) $\backslash\backslash\ s_i \rightarrow s_j$

(3) $\backslash\backslash\ s(P)_i \rightarrow s(P)_j$

(4) $\backslash\backslash\ e_i \rightarrow \square$

where P and Q are *finite* (not "any symbol") specifiers. By using rules of groups B and C, and rule D, we ultimately represent each restricted subclass in the form

$$\mathcal{E}'_i\ \backslash\backslash\ \delta_1\ \backslash\backslash\ \delta_2\ \ldots\backslash\backslash\ \delta_q = \mathcal{E}_i\ //\ \delta^+_1\ \ldots\ //\ \delta^+_p\ \backslash\backslash\delta^-_1\backslash\backslash\ldots\backslash\backslash\delta^-_q$$

where each of the restrictions $\backslash\backslash\ \delta^-_i$ is one of the four types above. We notice in addition that for each unspecified s-variable there may be only one restriction (1) (because of Rule C.1), and a number of restrictions (2). For a specified s-variable there may be only some restrictions of type (3); for each e-variable there may or may not be one restriction (4) in the list. A restricted class so represented will be reterred to as s-*restricted*.

Example.

In the definition of a symmetric binary string:

.1 $\alpha \Rightarrow T$

.2 $\alpha\ s(10)_1\ e_2\ s(10)_2 \Rightarrow \alpha\ e_2$

.3 $\alpha\ e_1 \Rightarrow F$

what is the set X of expressions processed by the third sentence?

This set may be represented in the form (1) as

$$X = e_1 \backslash \square \backslash s(10)_1\ e_2\ s(10)_1$$

Projecting the left sides of the first and second sentences of e_1 , we get representation (2):

$$X = e_1 \backslash\backslash\ e_1 \rightarrow \square \backslash\backslash\ (e_1 \rightarrow s_3e_1)\ (s_3 \rightarrow s(10)_3)\ (e_1 \rightarrow e_1s_4) \cdot (s_4 \rightarrow s(10)_4)\ (s(10)_4 \rightarrow s(10)_3)$$

Using Rules RC and UR we get:

$$X = X^1 \cup X^2 \cup X^3 \cup X^4 \cup X^5$$

(because of the composition of five contractions).

Let us transform each term now:

$$X^1 = e_1 \backslash\backslash\ e_1 \rightarrow \square \backslash\backslash\ e_1 \rightarrow s_3e_1 = e_1 // e_1 \rightarrow (e_5)e_1 = (e_5)e_1$$

[Rule C.6]

$$X^2 = e_1 \backslash\backslash\ e_1 \rightarrow \square // e_1 \rightarrow s_3e_1 \backslash\backslash\ s_3 \rightarrow s(10)_3$$
$$= e_1 // e_1 \rightarrow s_3e_1 \backslash\backslash\ s_3 \rightarrow s(10)_3 = s_3e_1 \backslash\backslash\ s_3 \rightarrow s(10)_3$$

[Rule B.6]

$$X^3 = e_1 \backslash\backslash\ e_1 \rightarrow \square // e_1 \rightarrow s_3e_1 // s_3 \rightarrow s(10)_3 \backslash\backslash\ e_1 \rightarrow e_1s_4$$
$$= s(10)_3\ e_1 \backslash\backslash\ e_1 \rightarrow e_1s_4 = s(10)_3 \cup s(10)_3\ e_1(e_y)$$

[Rule A.10]

$$X^4 = e_1 \backslash\backslash\ e_1 \rightarrow \square // e_1 \rightarrow s_3e_1 // s_3 \rightarrow s(10)_3 // e_1 \rightarrow e_1s_4 \backslash\backslash\ s_4 \rightarrow s(10)_4$$
$$= s(10)_3\ e_1s_4 \backslash\backslash\ s_4 \rightarrow s(10)_4$$

$$X^5 = s(10)_3\ e_1\ s(10)_4 \backslash\backslash\ s(10)_4 \rightarrow s(10)_3$$

Thus the set X is the union of six s-restricted classes:

$$X = (e_5)e_1 \cup s_3e_1 \backslash\backslash\ s_3 \to s(10)_3 \cup s(10)_3 \cup s(10)_3\ e_1(e_6)$$
$$\cup\ s(10)_3e_1s_4 \backslash\backslash\ s_4 \to s(10)_4 \cup s(10)_3e_1\ s(10)_4 \backslash\backslash\ s(10)_4 \to s(10)_3$$

An s-restricted class may be empty even if this is not immediately obvious (we recall that when an empty specifier appears, the class is removed in the course of transformation, so in an s-restricted class no specifier may be empty). The following is an example:

$$s(10)_1s(10)_2s(10)_3 \backslash\backslash\ s(10)_1 \to s(10)_2 \backslash\backslash\ s(10)_1 \to s(10)_3 \backslash\backslash\, s(10)_2 \to s(10)_3$$

Our task now is to construct an algorithm which would determine whether a given s-restricted class is empty or not. We shall construct this algorithm as actually picking up one representative of the class, if such exists. We notice that only restrictions of the type

$$\backslash\backslash\ s(\mathcal{P})_i \to s(\mathcal{P})_j$$

may cause trouble in picking up a representative. Indeed, the restriction

$$\backslash\backslash\ s_i \to s(\mathcal{Q})_i$$

leaves us with still an infinite choice of possible symbols s_i. The same is true with respect to a restriction of the form

$$\backslash\backslash\ s_i \to s_j$$

(we can just take a new symbol for each new unspecified s-variable we encounter). And of course, there is no problem in picking up a nonempty expression.

If we have a system of inequalities for a number of s-variables with a finite specifier $\mathcal{P}$ consisting of p different symbols, we construct a graph G, the vertices of which are these s-variables, and two vertices i and j are connected by an edge if there is a restriction $\backslash\backslash\ s(\mathcal{P})_i \to s(\mathcal{P})_j$ or $\backslash\backslash\ s(\mathcal{P})_j \to s(\mathcal{P})_i$. Then we compute (see, e.g., [27]) the *chromatic number* $\chi(G)$, that is the minimum number of colors needed to color verticies in such a manner that no adjacent vertices have the same color. If $X(G) > p$, the restricted class is empty, otherwise we can pick up a represent·ative, treating symbols from P as colors.

The result of our consideration may be formulated as

Theorem 4.3. For each vertex in the graph of states of a Refal machine, the quasiinput set can be represented as a union of nonempty s-restricted classes.

Now we know how to *clean* the graph of states; we remove all vertices to which empty quasiinput sets correspond; we also remove dynamic arcs leading to these vertices. This gives us

Theorem 4.4. An algorithm exists which makes any graph of states clean.

4.4. Compilation Strategy

When we have constructed a complete 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 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 from entering a graph of states. The general scheme of constructing a graph of states is as follows. Starting with the initial configuration, we execute driving, and every time that we receive an active configuration we decide whether to continue driving or to express the configuration through some explored basic configurations and stop driving. More specifically, the *strategy of compilation* may be defined by giving answers to the following questions:

(1) How do we choose the subexpression to be driven? Two most natural strategies would be, first, following the definition of the *leading* concretization sign, and second, the strategy "from without within".

(2) When do we start trying to stop driving? For example, we may take as a rule never to stop on a *transitory* vertex, no matter whether or not it is a basic configuration.

(3) When and how do we decompose a configuration?

(4) What should be the initial list of basic configurations, or the initial criterion for a configuration to be basic?

(5) When and how do we expand the list of basic configurations, or to change the criterion?

With a compilation strategy given, the first question to answer is: will this strategy necessarily lead to the construction of a finite complete graph?A question still more fundamental: are there any strategies at all which always lead to a successful end (complete graph)?

The answer to the last question is positive. There is a compilation strategy, which finds itself at the extreme interpretation end of the interpretation-compilation axis. It may be called *generalization to functions*. With this strategy the following answers are given to the five above mentioned questions:

(1) The range of the leading concretization sign is chosen.

(2) Try to stop at every step.

(3) Always decompose a configuration which has more than one k-sign, separating the leading subexpression.

(4) Basic configurations are configurations of the form:

$$k\ F\ e_x \perp$$

with any determiner F.

(5) This criterion does not change.

Essentially, this strategy leaves the program in exactly the same form as it has been written in Refal. The graph of states decomposes into subgraphs corresponding to functions, and no optimization occurs. E.g., if we discover a call of function F^1 with a specific completely defined argument:

$$k\ F^1\ A\ B\ C\ D \perp$$

then instead of computing and substituting the result of concretization, we will have to generalize to the configuration $k\ F^1\ e_1 \perp$ and construct the complete graph of states for this configuration, that is reproduce the definition of F^1.

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 the configurations chosen as basic in constructing the graph of states.

The strategy of generalization to functions can be considerably improved by excluding transitory configurations and including into the basic configurations the *formats* of functions. We shall illustrate this strategy (*generalization to formatted functions*) by the following example. Let function F^1 with the format

$$\text{(GFF)} \qquad k\ F^1\ (e_1)\ (e_2)\ e_3 \perp$$

be defined by the sentences:

$$k\ F^1\ (e_1)(e_2)\ s_3 e_4 \Rightarrow k\ F^1\ (e_1)\ (e_2 s_3)\ e_4 \perp$$

$$k\ F^1\ (e_1)\ (e_2) \Rightarrow (e_1)\ (e_2)$$

Using the strategy of generalization to formatted functions we take expression (GFF) as a basic configuration. This will give us the graph of states represented in Figure 9a. Let us compare it with the graph in Figure 9b, which is the result of generalization to unformatted functions for the same Refal program.

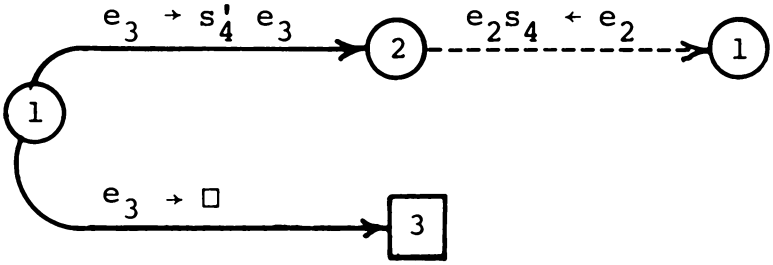


Figure 9a

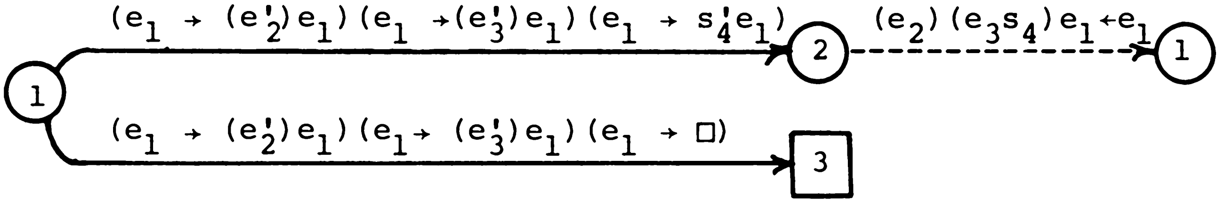

$(e_1 \to (e'_2)e_1)(e_1 \to (e'_3)e_1)(e_1 \to s'_4 e_1)$
$(e_2)(e_3 s_4)e_1 \leftarrow e_1$
1
2
1
$(e_1 \to (e'_2)e_1)(e_1 \to (e'_3)e_1)(e_1 \to \square)$
3

Figure 9b

According to Figure 9b, the argument e_1 should be split into three subarguments e_1 , e_2 and e_3 , whereafter symbol variable s'_4 is separated (if possible) from e_1. After exchanging symbol s_4 , the three subarguments are once again united into one argument — all this being repeated at each step of the Refal machine. According to the version of Figure 9a, the start configuration has three arguments. At each step, a symbol s'_4 is separated from the third argument and passed to the second. The first argument is not involved at all. Obviously, the graph in Figure 9a, when mapped on the object machine (a real computer), will provide an essentially higher efficiency than the other graph: there is no need to decompose and then recompose the argument with the help of parentheses during each cycle of the loop. Our second strategy is more compilative than the first, because the format parentheses are included in the basic configuration, making it more specific. The parentheses are now absent from the object program; they have been dealt with in the process of compilation. Seen from the other side, the first strategy is more interpretive, because the argument of the function F^1 is interpreted at run time as an expression of the form $(e_1)(e_2)e_3$.

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 let us go back to section 4.1, where the language /L/ was defined and the start configuration

(1) k /L/ CROSS(CAT); ADD(DOG) (e_x) ⊥

given.

If the set of basic configurations is declared empty, which means the maximum depth of compilation (and in this case leads to a finite graph because a solution without loops exists), we receive the following graph of states (Figure 10):

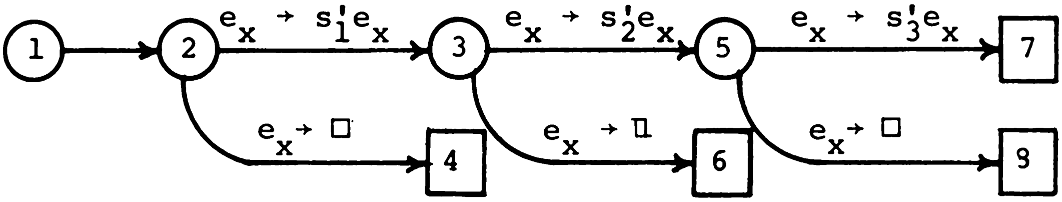


Figure 10

where the configurations are:

(2) k /L/ ADD(DOG) (k /Ll/ CROSS(CAT) (e_x) ⊥) ⊥

(3) k /L/ ADD(DOG) (C s_1 k /Ll/ CROSS(AT) (e_x) ⊥) ⊥

(4) CATDOG

(5) k /L/ ADD(DOG) (C s_1 A s_2 k /Ll/ CROSS(T) (e_x) ⊥) ⊥

(6) C s_1ATDOG

(7) Cs_1As_2T$s_3$$e_x$DOG

(8) Cs_1As_2TDOG

When mapped on the object machine, this graph will become an efficient program. But imagine 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 tradeoff between space and time parameters — to make the program more compact at the expense of retaining a level of interpretation. We declare as basic the configuration

(9) k /L1/ CROSS$(e_y)(e_x) \perp$

The following graph will be constructed as the result of the compilation process (Figure 11):

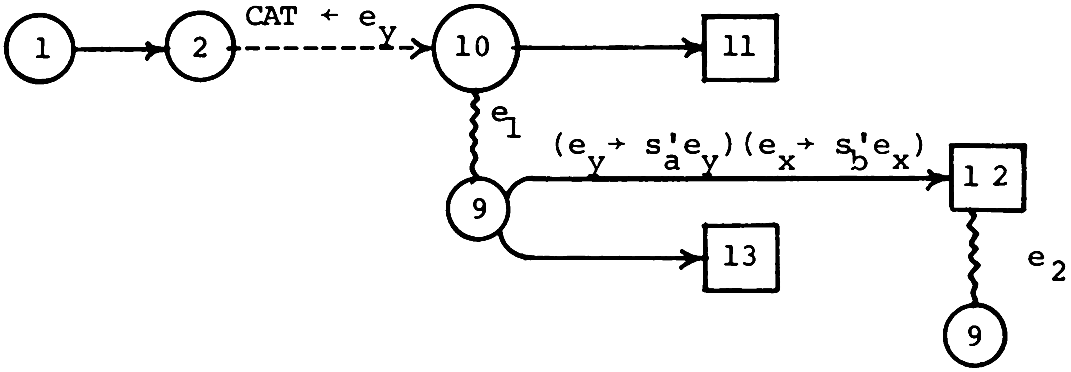


Figure 11

(10) k /L/ ADD(DOG) $(e_1) \perp$

(11) e_1 DOG

(12) $s_a s_b e_2$

(13) $e_y e_x$

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 — the procedure of "crossing", which of course could have a longer word than "CAT" as the first argument — is interpreted.

The notion of compilation strategy provides us with a key to the notion of a *normal form* of a Refal program. Suppose we have a graph of states and consider various ways to map it onto the object Refal machine. Differences in Refal programs stemming from different ways of mapping are clearly nonessential; a question of form. So let us fix a definite way of mapping — say, that described in Section 4.2. Now we choose a compilation strategy. If it is *generalization to functions*, then compilation of the original Refal program for any function with a subsequent mapping will give us a new Refal program which is a representation of the original in a *standard normal form*. When we change the compilation strategy, we receive functionally equivalent Refal programs which may be referred to as *normal forms*, each normal form being defined by the corresponding compilation strategy. Differences between various normal forms of the same function may be huge and are "essential", not "formal"; they reflect the differences in the strategies they have resulted from.

4.5. Perfect Graphs.

A *walk* in a graph of states is a sequence of alternate vertices and arcs $V_1A_1V_2A_2 \dots V_{k-1}A_{k-1}A_k$ which "might be" followed (passed) by the Refal machine with some definite values of the input variables (i.e. the Refal machine in a definite *exact input state*). When we say "might be" here, we mean that the actual existence of an exact input state which forces the Refal machine to make this walk is not presupposed; a walk exists if certain rules are observed in its construction. These rules are as follows:

(1) If only horizontal arcs are outgoing from a vertex V_i in the walk, then any of them can be taken as A_i. For a concise representation of a walk, as in the case of a path, it is convenient to write out only the numbers of vertices and separate them by special signs indicating the nature of the connecting arc: it will be a comma "," for a dynamic arc and an equality

sign "=" for a representation arc. (If needed, the contraction on the arc may be specified in parentheses before the comma.)

(2) If there is a vertical arc outgoing from V_i, it must be taken as A_i when we first encounter V_i; this downward passage of a vertical arc will be denoted in our concise notation by a left bracket "[". When we come to a passive terminal configuration V_j, we look for the latest unpaired left bracket and return to the vertex preceding this bracket. This upward shift is depicted by a right parentheses "]" after V_j, which becomes paired, of course, with the last left bracket. A pair of corresponding brackets will be referred to as a *functional loop*.

(3) After a right bracket closing a function loop, a horizontal arc must be taken if there are any.

(4) When we come to a passive terminal vertex, without a composition arc or after closing the functional loop, and there are no unpaired left brackets, the walk cannot continue. It is concluded.

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 exact input state either a concluded or an infinite walk corresponds: the one taken by the Refal machine from this initial state. To each walk (*finite*: a walk is finite if the opposite is not stated) an input set corresponds, which comprises all exact input states 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 will be called *perfect*.

The graph in Figure 4 is perfect. We can easily find a corresponding input set for each possible walk in it, and this set will not be empty. For instance, for the walk 1, 4 = 14, 16 = 14, 17 the restricted input class

$$/PLUS/\ s_3\ e_1 \backslash\backslash\ s_3 \rightarrow /PLUS/$$

corresponds to it, and for the concluded walk 1, 4 = 14, 17 = 14, 17 = 14, 16 = 14, 15 the corresponding input set is

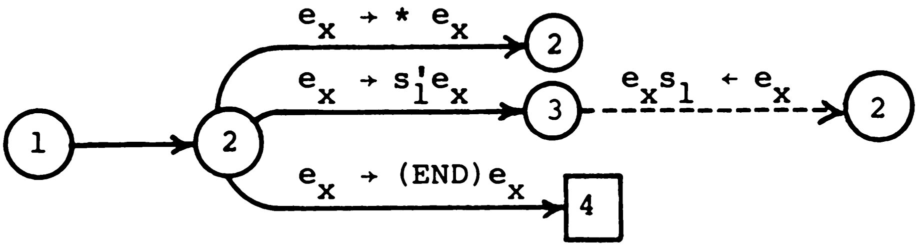
$$s_3\ s_4\ /PLUS/ \backslash\backslash\ s_3 \rightarrow /PLUS/ \backslash\backslash\ s_4 \rightarrow /PLUS/$$

If there are no redundant variables in a graph of states, or they are never subject to contractions (which is the case for Figure 4), and if the input variables have the same range of values on both ends of each transformation (submission) arc (which again is the case for Figure 4), then a clean graph of states will be also perfect.

Here is an example where a redundant (transformation) variable *is* subject to contraction, but the graph of states,is, nevertheless, perfect. We choose to define the procedure of deleting asterisks from a string of symbols in this bizarre way:

$$\begin{array}{lll}
\alpha\ e_x & \Rightarrow & \alpha^1 e_x(END) \\
\alpha^1 {*} e_x & \Rightarrow & \alpha^1 e_x \\
\alpha^1 s_1 e_x & \Rightarrow & \alpha^1 e_x s_1 \\
\alpha^1 (END) e_x & \Rightarrow & e_x
\end{array}$$

The following graph of states corresponds to this (Figure 12), which as can easily be seen is perfect:



$e_x \rightarrow * \ e_x$
$e_x \rightarrow s_1' e_x$
$e_x s_1 \leftarrow e_x$
$e_x \rightarrow (END) e_x$
1
2
3
2
2
4

Figure 12

A perfect graph (or the program in Refal represented by this graph — we will not distinguish these) cannot be improved by a compilation process. Compilation is consequential when there is *a margin of generality* in the original program: the definitions of functions are more general than is really needed, and walks exist in the graph of states which in fact cannot be actualized under any input assumptions. A perfect graph of states has no margin of generality, and all the tests implied by the dynamic arcs in such a graph must be actually performed at every step of the computation process. This does not mean that there may be no functionally equivalent graph which would work more efficiently than a perfect graph, i.e. our term "perfect" does not mean that it is perfect in any sense of the word. E.g., the following function, which only scans its argument without doing anything:

$$\begin{aligned} &\beta\ s_1 e_2 && \Rightarrow\ s_1 \beta e_2 \\ &\beta\ (e_1)\ e_2 && \Rightarrow\ (\beta e_1)\ \beta e_2 \\ &\beta && \Rightarrow \end{aligned}$$

has a perfect graph of states, but it is functionally equivalent to:

$$\beta\ e_1 \quad \Rightarrow \quad e_1$$

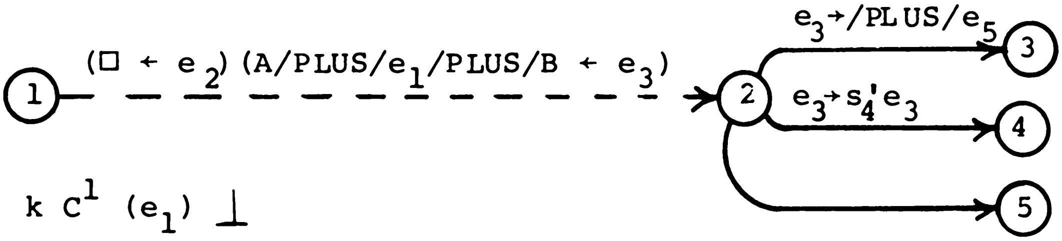
This example shows us the limits of the compilation process: compilation is essentially *computation in a general form*, but it does not include the application of *the principle of mathematical induction*. This is why the transformation of the function β as indicated above is beyond our capability at the moment. The inclusion of induction into the system of formal transformations of Refal programs will be done later in Section 4.6.

So, a perfect graph is a graph which cannot be improved by a straightforward compilation process. But even by a straightforward compilation process we can achieve a very impressive level of optimization — to be more exact, we can eliminate most typical efficiency losses resulting from automated, straightforward construction of algorithms from some "building blocks". We demonstrate it by giving three types of optimizations as examples.

The first and most obvious type of optimization is, of course, executing at compile time all calculations which are possible to do without knowing the input data. An example was given in Section 4.2, where the initial function

$$k\ C^1\ (e_1) \Rightarrow \alpha\ A\ /PLUS/\ e_1\ /PLUS\ B \perp$$

where function α was defined on page 77, was transformed into the efficient definition represented on page 85, which has a perfect graph of states. The graph of the initial definition is not, obviously, perfect: it contains an element, reproduced in Figure 13, where only the walk 1, 2, 4 is feasible, but not the walk 1, 2, 3, nor the walk 1, 2, 5. Note that the *paths* 1,2,3 and 1,2,5 are feasible, so that the graph is clean.



(1) $k\ C^1\ (e_1) \perp$

(2) $\beta\,(e_2)e_3 \perp$

Figure 13

The second type of optimization, "loop cleansing", will be illustrated in an example discussed by E. W. Dijkstra (see [28], p. 23-24). Consider the following two programs:

```
              if B2 then
              begin while B1 do S1 end
(1)                else
              begin while B1 do S2 end
```

and

```
              while B1 do
(2)           begin if B2 then S1 else S2 end
```

Here B1 and B2 are boolean expressions, and S1 and S2 are statements. It is supposed furthermore that B2 is constant, i.e. unaffected by the execution of either S1 or S2, and there are no side effects of the evaluation of the boolean expressions. With these presumptions, programs (1) and (2) are equivalent. Comparing them, E. W. Dijkstra writes: "I can establish the equivalence of the output of the computations, but I cannot regard them as equivalent in any other useful sense. I had to force myself to the conclusion that (1) and (2) are 'hard to compare'. Originally this conclusion annoyed me very much."

The notion of the perfect graph of states makes it easy to compare these programs, for the graph of states of the first program is perfect, while that of the second is not.

To see this clearly, we will translate ALGOL-60, in which the programs are written, into Refal, i.e. we will map the Algol machine onto the Refal machine. The state of the Algol machine is determined by the position of the control point and the values of the variables. Different positions of the control point correspond to different configurations of the Refal machine. ALGOL-60 is an instruction language. If we were to define its semantics in Refal, the simplest way to do so would be by using the configuration

$$k\ /ALGOL/\ (E_1)\ E_2 \perp$$

where E_1 and E_2 together always make a program in ALGOL, and the right parenthesis serves as the control point, the currently executed instruction being placed immediately after it. (Compare the language /L/ in Section 4.1). So, the start configuration in the case of program (1) will be:

(1) k /ALGOL/ ( ) *if* B2 *then*... etc. $\perp$

After the branching according to B2 has been executed, the configuration will be either

(2) k /ALGOL/ (*if* B2 *then*) *begin while* B1 ... etc. $\perp$

or

(3) k /ALGOL/ (*if* B2 *then begin*...*end else*) *begin while* ...etc. $\perp$

and so forth.

Let us consider the variables now. We must separate the value of the boolean expression B2 from the others, because it is independent of the other variables and remains constant. Let us denote it by e_b. The other variables will be represented by a variable e_x, which may consist of any number of subarguments. The expression B1 depends on e_x. The statements S1 and S2 are supposed to change the value of e_x.

Now program (1) can easily be translated into the following program for the starting configuration $k\,C^1\,(e_b)\,(e_x)\perp$:

$$\begin{aligned}
&k\,C^1\,(T)\,(e_x) &&\Rightarrow\; k\,C^2\,(e_x)\perp\\
&k\,C^1\,(F)\,(e_x) &&\Rightarrow\; k\,C^3\,(e_x)\perp\\
&k\,C^2\,(e_x) &&\Rightarrow\; k\,C^4\,(k\,B^1\,(e_x)\perp)\,(e_x)\perp\\
&k\,C^4\,(T)\,(e_x) &&\Rightarrow\; k\,C^2(kS^1(e\ )\perp)\perp\\
&k\,C^4\,(F)\,(e_x) &&\Rightarrow\; e_x\\
&k\,C^3\,(e_x) &&\Rightarrow\; k\,C^6\,(k\,B^1(e_x)\perp)\,(e_x)\perp\\
&k\,C^6\,(T)\,(e_x) &&\Rightarrow\; k\,C^3\,(k\,S^2(e_x)\perp)\perp\\
&k\,C^6\,(F)\,(e_x) &&\Rightarrow\; e_x
\end{aligned}$$

Program (2) will be translated into:

$$\begin{aligned}
&k\,C^1\,(e_b)(e_x) &&\Rightarrow\; k\,C^2(kB^1(e_x)\perp)(e_b)(e_x)\perp\\
&k\,C^2(T)(e_b)(e_x) &&\Rightarrow\; k\,C^3(e_b)(e_x)\perp\\
&k\,C^2(F)(e_b)(e_x) &&\Rightarrow\; e_x\\
&k\,C^3(T)(e_x) &&\Rightarrow\; k\,C^1(T)(kS^1(e_x)\perp)\perp\\
&k\,C^3(F)(e_x) &&\Rightarrow\; k\,C^1(F)(kS^2(e_x)\perp)\perp
\end{aligned}$$

The graphs of states for programs (1) and (2) are represented in Figures 14a and 14b respectively. They are not complete, for configurations B^1, S^1 and S^2 are not explored. But we can apply to such graphs the concepts *perfect* and *imperfect* in the sense: an incomplete graph is perfect, if there are *some* definitions of the unexplored configurations for which the completed graph proves perfect; it is imperfect

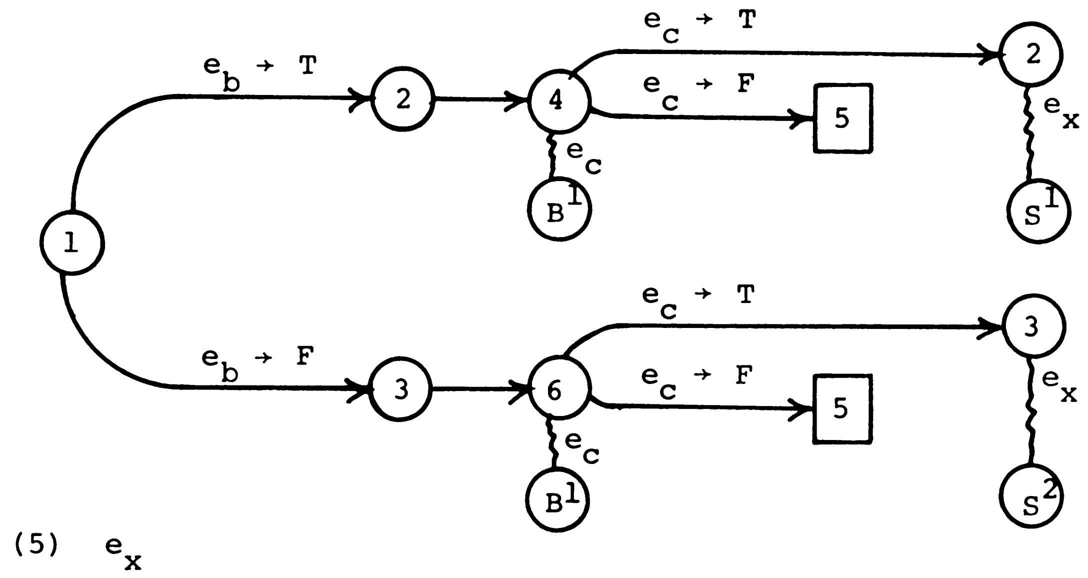
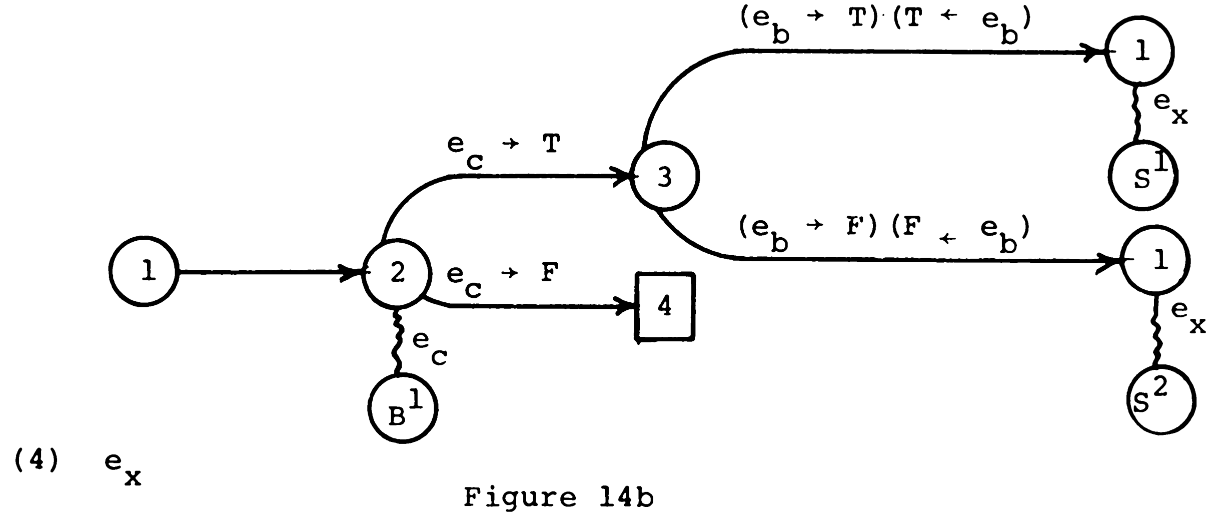


Figure 14a



1
2
e_c
B^1
$e_c \to T$
$e_c \to F$
3
4
$(e_b \to T)(T \leftarrow e_b)$
$(e_b \to F)(F \leftarrow e_b)$
1
1
e_x
S^1
S^2
(4) e_x

Figure 14b

if for *any* definitions the completed graph will be imperfect. In this sense the graph in Figure 14b is imperfect: the walk

$1,\ 2[B^1],\ 3(e_b \to T),\ 1[S^1],\ 2[B^1],\ 3(e_b \to F),\ 1[S^2]$

is not feasible, as are all the walks (an infinite set of them) which include both dynamic arcs originating from C^3.

If we are given program (2) as the definition of the algorithm, we can improve it and make the graph of states perfect by using the compilation process. Almost any compilation strategy which is deeper (farther from interpretation) than the generalization to (formatted) functions will do the job. It comes very naturally, but it is interesting to note, that the resulting graph of states, although perfect, will be different from Figure 14a. It is represented in Figure 14c and is obtained in the following way.

We start with configuration C^1, and until we have come to configurations C^3 and C^4 we are just copying Figure 14b, because there is actually no other choice. Driving C^3 we get two configurations:

$$kC^1(T)(kS^1(e_x)\bot)\bot$$

and

$$kC^1(F)(kS^2(e_x)\bot)\bot$$

We have to decompose them, separating configurations S^1 and S^2, because they are not specified and cannot be explored. Therefore we have configurations:

(5) $$kC^1(T)(e_x)\bot$$

(6) $$kC^1(F)(e_x)\bot$$

If we generalize them to configuration C^1, we will get Figure 14b. But why should we? These configurations result from substututing some specifid values into an already existing configuration C^1. To keep and continue to explore such configurations *at their first appearance,* is a safe strategy, because there may be only a finite number of ways to select variables for substitution.

Driving C^5 we get, after an inevitable decomposition, configuration

(7) $$kC^2(e_c)(T)(e_x)\bot$$

It produces a branching with one arc leading to a passive configuration, and the other to the configuration

$$kC^3(T)(e_x)\bot$$

which is transitory and therefore will not be considered as a possible stop. Then we come to C^5 once more and this time of course stop the process. Configuration C^6 is treated correspondingly.

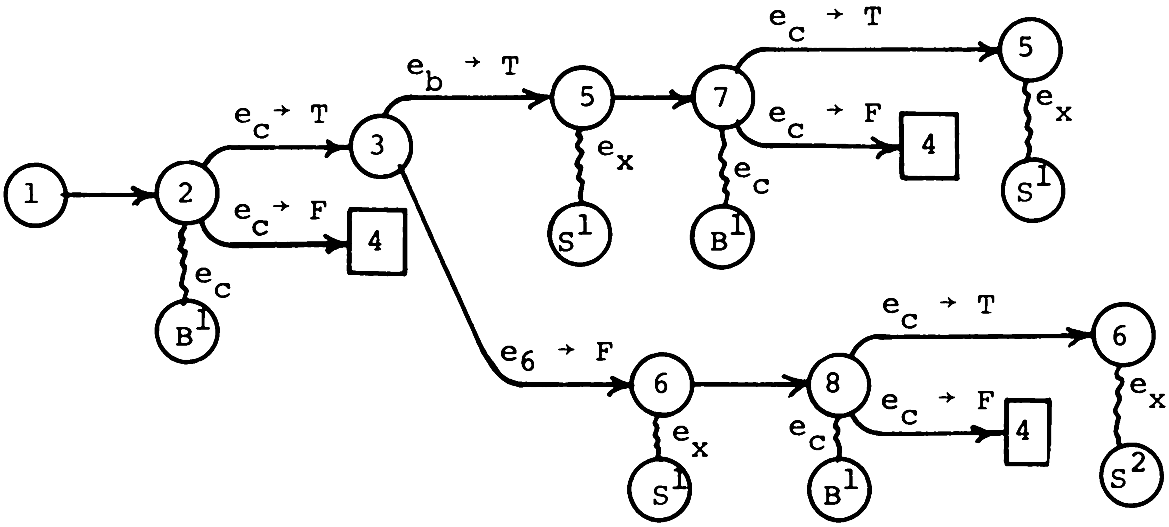


Figure 14c

The program we have come to (Figure 14c) is efficient: there are no unneeded tests of boolean variable B^2 in the loop. It is a bit more compilative (and a bit larger) than the equally efficient program (1) (Figure 14a). But this bit of compilation does not come without a benefit: if boolean variable B1 is *false* from the very beginning, our program, unlike program (1), will not even ask for the value of B2. This may be very valuable in further optimization.

As the third example of optimization let us consider the following definitions:

$$kF^aAe_1 \Rightarrow B\ kF^a e_1 \perp$$
$$kF^a s_1 e_2 \Rightarrow s_1\ kF^a\ e_2 \perp$$
$$kF^a \Rightarrow$$
$$kF^b Be_1 \Rightarrow C\ kF^b\ e_1 \perp$$
$$kF^b s_1 e_2 \Rightarrow s_1\ k\ F^b\ e_2 \perp$$
$$kF^b \Rightarrow$$

And let the initial configuration be

(1) $kF^{b}\ kF^{a}\ e_{1} \perp \perp$

The corresponding graph of states is represented in Figure 15.

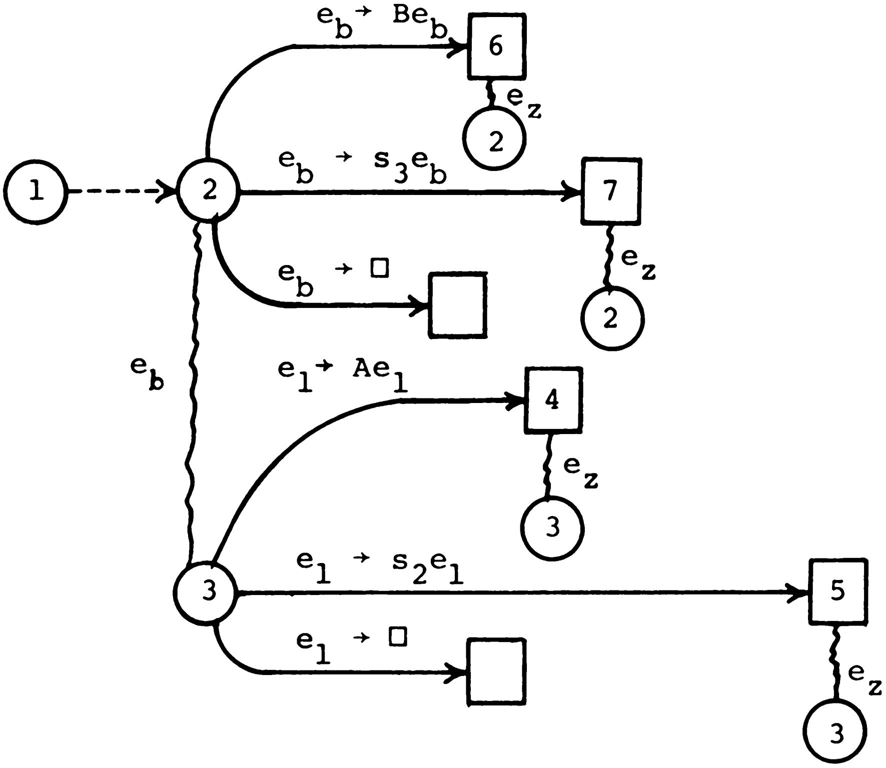


Figure 15

The configurations in the graph are:

(2) $k\ F^{b}\ e_{b} \perp$

(3) $k\ F^{a}\ e_{1} \perp$

(4) $B\ e_{z}$

(5) $s_{2}\ e_{z}$

(6) $C\ e_{z}$

(7) $s_{3}e_{z}$

This graph of states is far from being perfect, in any sense of the word. For example, the following set of walks is unfeasible:

$$1=2[3,4[\ 3W_3\ ]],7[\ 2W_2\ ]$$

where $3W_3$ and $2W_2$ are any walks staring with vertcies 3 and 2, correspondingly. Still worse, any walk of the form

$$1=2[3W_3]W_2$$

where the number of arcs in the walks $3W_3$ and $2W_2$ are not equal, is unfeasible. This is the reflection of the organization of the procedure as a double passage of the argument.

Let us however apply the compilation process with the strategy of driving *from without within* On the first step, the call of F^a will be driven, and we receive three new configurations:

(2) $kF^b\ B\ kF^a\ e_1 \perp \perp$

(3) $kF^b\ s_2\ kF^a e_1 \perp \perp$

(4) $kF^b \perp$

Now the external call can be driven in all the three configurations, and in the result we come to the starting configuration (1). Thus we see that the graph is complete and stop the process. The graph is represented in Figure 16, and the passive configurations are:

(5) $C\ e_z$

(6) $s_2 e_z$

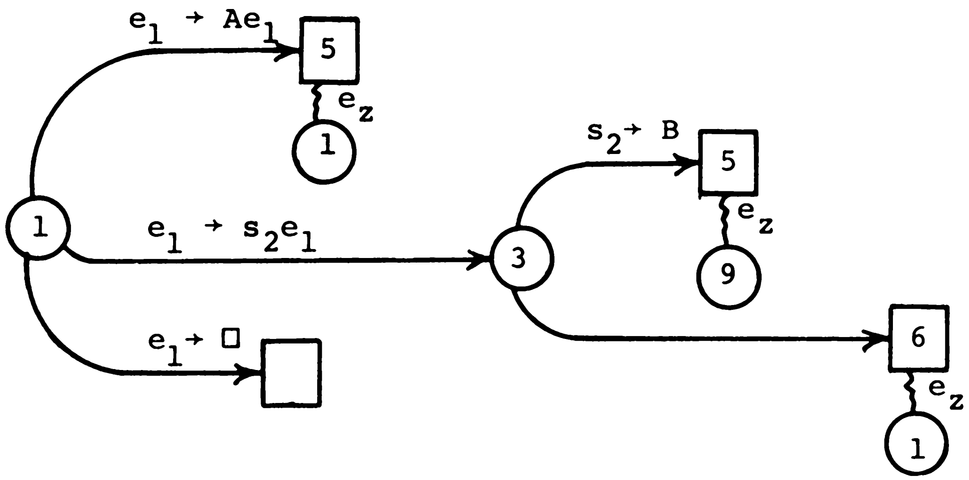


Figure 16

This graph of states is perfect. It can be further simplified by excluding the vertex 3. The corresponding program in Refal is

$$\begin{array}{lll}
kC^1(Ae_1) & \Rightarrow & C\ kC^1\ (e_1) \perp \\
kC^1(Be_1) & \Rightarrow & C\ kC^1\ (e_1) \perp \\
kC^1(s_2e_1) & \Rightarrow & s_2\ kC^1(e_1) \perp \\
kC^1(\ ) & \Rightarrow &
\end{array}$$

So, in the result of compilation we have transformed a double-pass procedure into a one-pass procedure: an essential optimization and an important one.

4.6. Generalization and Induction

A *generalization* of a set of expressions S is any expression G such that for any expression E, if $E \in S$, then $E \subseteq G$. If G is an L-expression, we call it an L-*generalization*. The same terms will be used with respect to *classes* (we recall that when it is a question of syntactic recognition, pattern expressions and classes may not be distinguished); one may also consider generalization of *restricted* classes, and allow generalizations to be restricted classes too. Speaking of restricted L-classes (or of pattern L-expressions with restrictions) we shall assume that they are s-*restricted*. (see Section 4.3 and Theorem 4.3). We shall be interested mostly in L-generalizations.

An L-generalization L will be referred to as *tight*, if there exists no L-generalization L' such that $L' \subset L$. The following example shows that a tight L-generalization is not unique. Let

$$\{XXX,\ XX\}$$

be a set of two (object) expressions (in examples like this, the comma will be used as separator, and therefore not allowed to enter expressions). The following three expressions are tight

L-generalizations:

$$e_1XX$$
$$Xe_1X$$
$$XXe_1$$

Indeed, consider any of the three expressions. If it is not tight, then a tighter generalization exists, which can be produced by a contraction. The variable e_1 may be contracted either into an empty expression, or into an expression which has at least one term. In the first case, the expression XXX from the original set will not be covered, in the second case the expression XX will not be covered. Therefore all three generalizations are tight.

Moreover, the operation of taking *any* possible tight L-generalization of two expressions is not associative in the following sense. Let a tight L-generalization L of two expressions E_1 and E_2 be found, and let L' be a tight L-generalization of L and E_3. Then, L' will not necessarily be a tight generalization of the set $\{E_1,E_2,E_3\}$. To give an example, again let E_1 be XXX and E_2 be XX. Let E_3 be X(Y)X. We take e_1XX as the generalization L. Generalizing L and E_3 we obtain e_1X, which is not a tight generalization for the set $\{E_1,E_2,E_3\}$. The tight generalization will be Xe_1X. Ergo: when we need a tight L-generalization of a set of expressions, we should consider them all together.

Notwithstanding these unpleasant properties of tight L-generalizations, they are extremely useful for the theory of compilation. The following property of L-generalization is crucial for certain applications:

Theorem 4.5. In a universe with a finite number of symbols, any class has only a finite number of L-generalizations.

To prove it we will first consider L-generalizations of terms (i.e. pattern expressions which are terms and represent terms, e.g., X, s_1 , etc.). A generalization of one or several terms which is a term itself will be called a t-*generalization*.

If the number of different symbols is finite, then we can write every s-variable as a specified s-variable, and we do not need restrictions which forbid a symbol variable to take certain values (we just exclude these values from the specifier). Consequently, the following lemma covers all possible cases of t-generalizations:

Lemma 1. The following propositions are true:

(1) Any t-generalization of a symbol S is either the symbol itself, or has the form $s(P)_i$, where P includes S.

(2) Any t-generalization of a symbol variable $s(P)_i$ has the form $s(Q)_j$ where $P \subseteq Q$.

(3) Any t-generalization of a class of the form (E) is a class of the form (E'), where E' is an L-generalization of E.

Three more lemmas are obviously true:

Lemma 2. An L-generalization of an empty expression is either empty, or an unrestricted (in the sense of s-restrictions) e-variable.

Lemma 3. If an expression has the form $T\ E$ where T is a term different from an e-variable, then its L-generalization either starts with an e-variable, or has the form $T'E'$, where T' is a t-generalization of T, and E' is an L-generalization of E. If an expression has the form $E\ T$, where T is a term different from an e-variable, then its L-generalization either ends with an e-variable, or has the form $E'T'$, where T' is a t-generalization of T, and E' is an L-generalization of E.

Lemma 4. If an expression starts with an e-variable, its L-generalization starts with an e-variable; if an expression ends with an e-variable, its L-generalization ends with an e-variable; if an expression starts and ends with an e-variable, its only L-generalization is e_1.

Using these lemmas we can algorithmically construct all possible L-generalizations of a given class. The process will consist of a finite number of steps, and at each step we will have a finite number of choices. Instead of giving a formal definition of the algorithm, we shall consider an example of its work.

We denote by $<E>$ the set of all possible L-generalizations of an expression E, and by $<T>^t$ the set of all t-generalizations of a term T. The set of all those L-generalizations of an expression E which start with an e-variable will be denoted by $<eE>$; and $<Ee>$ will mean the set of those L-generalizations which end with an e-variable.

Let us construct the L-generalizations of the expression Xe_1Y in the universe with three symbols: X, Y and Z.

We start with using the first part of Lemma 3, i.e. processing our expression from left to right (it could be the other way around). As a result of the first step we have two possible sets of L-generalizations:

(1) $$<X>^t <e_1Y>$$

(2) $$<eXe_1Y>$$

The number of possible t-generalizations is always finite. In our case

$$<X>^t = \{X, s(XY)_2, s(XZ)_2, s(XYZ)_2\}$$

Exploring the possibility (1), we use Lemma 3 for $<e_1Y>$. The first possibility is

(1.1) $$<X>^t<e_1><Y>^t$$

i.e.

$$<X>^t e_1<Y>^t$$

where

$$<Y>^t = \{Y, s(XY)_3, s(YZ)_3, s(XYZ)_3\}$$

We have obtained a set consisting of 16 generalizations, such as

$$Xe_1Y$$

$$Xe_1s(XY)_3$$

$$s(XYZ)_2e_1s(YZ)_3$$

etc.

If we choose the second possibility in the exploration of (1), we get:

(1.2) $$<X>^t <e_1Ye>$$

which by virtue of Lemma 4 is

$$<X>^t e_1$$

This gives us four more generalizations.

Continuing in the same manner the exploration of the possibility (2), we receive two more sets of generalizations:

(2.1) $$<eXe_1><Y>^t = e_1<Y>^t$$

(2.2) $$<eXe_1Ye> = e_1$$

Generating variables in the process of generalization, we first make all of them different, which gives us the widest classes with respect to possible values of s-variables. Then we consider all possible pairs of s-variables and produce new classes by contraction, if corresponding pairs in the original expression consisted of identical symbols or were variables with identical indexes. We can expand this procedure to include e-variables also. We will then receive generalizations in which some e-variables may occasionally coincide, but which in all other respects are L-expressions. We shall call such expressions LE-*generalizations.*

We have considered generalization of unrestricted classes. If there are restrictions, we can cancel any combination of them. We also can introduce new restrictions if it is in agreement with the original expression. In our example, we can add the restriction $\backslash\backslash\ s_2 \rightarrow s_3$, since their projections in the original expression X and Y were different. According to the definition of s-restricted expressions, there are two types of restrictions that can be used:

$$\backslash\backslash\ s_i \rightarrow s_j$$
$$\backslash\backslash\ s_i \rightarrow \square$$

Since the number of e-variables and the number of pairs of s-variables are finite, manipulation with restrictions can give only a finite number of additional generalizations.

It should be noted that if we did not restrict generalizations with L-expressions, every expression would allow an infinite number of generalizations. Indeed let $\mathcal{E}$ be an expression which does not contain variable e_x. Then

$$\mathcal{E}e_x$$
$$\mathcal{E}e_xe_x$$
$$\mathcal{E}e_xe_xe_x$$

and so on ad infinitum, will be different generalizations of $\mathcal{E}$.

From the finiteness of the number of L-generalizations it follows

Theorem 4.6. There exists an algorithm which for every set of s-restricted classes gives the set of its tight L-generalizations.

The algorithm, if we ignore restirctions, is as fast as the GPA. Essentially it differs from the above algorithm only in that we redefine t-generalization to become a unique operation and make some pruning of the production tree. The following rules define t-generalization on a pair of expressions when possible:

(1) $$<S_1,S_2>^t = s(S_1S_2)_i$$

(2) $$<S_1,s(P)_i>^t = s(\{S_1\} \cup P)_i$$

(3) $$<s(P)_i,s(Q)_j>^t = s(P \cup Q)_i$$

(4) $$<(E_1),(E_2)>^t = (<E_1,E_2>)$$

Associative extension defines t-generalization on a set of any number of expressions. When t-generalization becomes impossible, we either change the direction of motion, or (if we already are moving from right to left) generate an e-variable and finish the process.

The following theorem may be useful:

Theorem 4 .7. If all of the expressions to be generalized have an e-variable on the main level of parenthesis structure, then their tight L-generalization is, on the main level, unique.

The proof of this theorem is left to the reader.

Using L-generalization we can formulate simple but powerful strategies of compilation. This is the simplest one (the points below refer to the five questions about a strategy listed in Section 4 .4).

(1) The from without within strategy is used to determine the subexpression to be driven.

(2) We try to stop driving at a given vertex of the graph of states only if a contraction is necessary, and only if there has already been, on the path to the current configuration C, another configuration C' with the same determiner. If $C \subseteq C'$, we loop C on C', otherwise we take a tight L-generalization $G = <C,C'>$, come back to the vertex C', and redo the graph generalizing C' to G.

(3) When we loop in accordance with p. (2), and configuration C' is a part of a larger configuration, we perform a decomposition, separating C' as a vertex in the graph of states.

(4) There is no necessity to declare any configurations as basic *a priori*. Should some be so declared, corresponding provisions must be added to pp. (2) and (3).

(5) The list of basic configurations is not expanded during the compilation process. When it is finished, one may make the list of basic configurations *a posteriori*.

It is easy to see that this strategy always leads to a finite complete graph of states in a finite number of steps. Indeed, no path in the graph may contain more than two configurations with the same determiner. On the other hand, only those symbols may appear in the compilation process and must be taken into account which appear in the original Refal program. Therefore our universe has a finite number of difffrent symbols, and we cannot generalize one and the same configuration more than a finite number of times because of Theorem 4 .5.

As an example of this strategy, consider the compilation process for the function

$$\begin{array}{lcl}
kFe_1 & \Rightarrow & kF^1(\ )e_1 \perp \\
kF^1(e_2)+e_1 & \Rightarrow & kF^1(e_2 -)\ e_1 \perp \\
kF^1(e_2)s_xe_1 & \Rightarrow & kF^1(e_2s_x)\ e_1 \perp \\
kF^1(e_2) & \Rightarrow & e_2
\end{array}$$

Our starting configuration is $kFe_1\perp$. Immediately, it turns into $kF^1(\ )e_1 \perp$. It is the first appearance of the determiner F^1, therefore we continue driving. Next step we obtain configuration $kF^1(-)e_1 \perp$ on the first branch. Now we have to generalize. The only tight L-generalization for these two configurations is $kF^1(e_2)e_1 \perp$. This is how we automatically find the correct format and easily complete the compilation. It of course gives the original definition because it is perfect.

The strategy we described ("L-generalization strategy") tries to make only those generalizations which are necessary in order to convolute an infinite graph of exact states into a finite graph of generalized states. It tries to avoid excessive overgeneralization. Of course it uses rather simple and crude means for that purpose, the technique of L-generalization. More subtle methods of generalization will lead to more perfect (less interpretive) programs. But we have reason to expect that even this technique will be quite adequate for a vast number of applications. One of the features of the compilation strategy described is that different calls of the same function appearing in the different branches of the graph of states may be generalized to different configurations. Thus we can use the same function with a very specialized argument, and with a general argument in the same program without being afraid of undesirable interference. It should also be noted that although the number of possible L-generalizations of a given configuration C may be, according to Theorem 4.5, very large, the number of generalization steps in the compilation process is very unlikely to be large, and is always incomparably less than the full number of L-generalizations. For in each step we produce a proper generalization of the preceding configuration, so that the maximum number of steps is the number of members in the longest sequence of the form:

$$C \subset G \subset G' \subset G'' \subset \ldots \subset e_x$$

and not the number of possible generalizations of C.

L-generalization may also be used for a direct (i.e. without a compilation process) transformation of a function by *induction*. This method is a thorough imitation of human reasoning by incomplete induction followed by a strict proof by mathematical induction.

Consider a function definition

$$\phi\ L^1 \Rightarrow R^1$$
$$\phi\ L^2 \Rightarrow R^2$$
$$\ldots$$

Let us compute the function ϕ with several specific arguments: A_1 , A_2 , etc. For each of these arguments we construct an argument-value form:

$$(A_1) = Z_1$$
$$(A_2) = Z_2$$
$$\ldots$$

etc.

where Z_i is the concretization of $\phi\ A_i \perp$ for all i.

Now we LE-generalize these forms, and this will be our tentative form for the function ϕ. We also could start with a generalization based on two values, then add one more and compare the new tentative form with the preceding one, repeating this procedure until we get a stable tentative form. If it is not trivial, (i.e. not just $(e_x) = e_1$), we try to transform the definition of function S using the discovered form. In the right side of the tentative form we turn all free variables which do not appear in the left side into new functions of the variables found in the left side. This is our hypothesis.

Suppose, e.g., that the tentative form is

$$(e_x) = L^{\phi}$$

with the right side (which of course is an L-expression):

$$(L^{\phi}) \qquad \text{ABC } e_1 \; s_2 \; (e_3)$$

The hypothesis is:

$$\phi \; e_x \perp \; = \text{ABC } \psi^1 \; e_x \perp \psi^2 \; e_x \perp (\psi^3 \; e_x \perp)$$

In the general form the hypothesis is:

$$\phi \; e_x \perp = L^{\phi} \; // \; \{(V_i \rightarrow \psi^i \; e_x \perp)\}_i$$

Now we take the first sentence defining ϕ and substitute the hypothetical expression for all ϕ-calls in the right side R^1. If we denote ϕ-calls entering R^1 by $\phi \; E^j \perp$ (for a number of j), then we get a new right side

$$\bar{R}^1 = R^1 \; // \; \{(\phi \; E^j \perp \rightarrow L^{\phi} \; // \; \{(V_i \rightarrow \psi^i \; E^j \perp)\}_i \;)\}_j$$

If, e.g., the first sentence is

$$\phi \; 1 \; e_x \Rightarrow \text{ABC } e_x \; 1 \; (\; \phi \; e_x \perp)$$

then $\bar{R}^1$ will be

$$(\bar{R}^1) \qquad \text{ABC } e_x \; 1 \; (\; \text{ABC } \psi^1 \; e_x \perp \psi^2 \; e_x \perp (\; \psi^3 \; e_x \perp) \;)$$

Now we try to recognize $\bar{R}^1$ as L^{ϕ}. If we succeed (without any contractions or restrictions), then the first sentence is in agreement with the hypothesis. As a result of the recognition process we receive a list of values assigned to all of the variables V_i. They will generally include function calls $\psi^i\ E^j \perp$ (and possibly calls of other functions). We denote this as

$$V_i := v_i(\{\psi^i\ E^j \perp\}_{ij})$$

Since $\bar{R}^1$ is the result of one-step concretization of $\phi\ L^1 \perp$, we get the following sentences for each of the functions ψ^i:

$$\psi^i\ L^1 \rightarrow v_i(\{\psi^i E^j \perp\}_{ij})$$

In our example:

$$\psi^1\ 1\ e_x = e_x$$
$$\psi^2\ 1\ e_x = 1$$
$$\psi^3\ 1\ e_x = \text{ABC}\ \psi^1\ e_x \perp \psi^2\ e_x \perp (\ \psi^3\ e_x \perp\ )$$

If all the sentences defining function ϕ allow this transformation, we will receive a new definition of the function ϕ using auxiliary functions ψ^i. Note that in each of the sentences defining the new functions ψ^i, the left side is identical to the left side of the corresponding sentence in the definition of ϕ.

As a special case, the tentative form may not contain any free variables at all. Then no auxiliary functions will be introduced and justification of the transfortation will consist consist of checking the identity of $\bar{R}^i$ to L^ϕ for all sentences defining ϕ. Consider, for example, this function:

$$\phi \Rightarrow T$$

$$\phi\ s_1\ e_2 \Rightarrow \phi\ e_2 \perp$$

We perform driving and receive values for $\phi \perp$ and for $\phi\ s_1 \perp$, which both are T. We take their tight L-generalization, which is also T. Thus L^ϕ is T. For the first sentence, $\bar{R}^1$ is obviously, T, and identical to L^ϕ. For the second sentence, we find after making the necessary substitution, that $\bar{R}^2$ is also T, i.e. also identical to L^ϕ. Therefore, we transform the definition of ϕ into

$$\phi\ e_1 \Rightarrow T\ .$$

4.7. Mapping on the Computer

The most general principle of mapping of the Refal machine on a target machine (computer) is: to each configuration of the Refal machine there corresponds a control point in the program for the computer, and to each variable in the graph of states of the Refal machine there corresponds a variable in the computer program together with an information field in the computer memory and an access method to use this information.

The linkage between the Refal machine and the computer is established by the concept of *mapped variable*. If V_i is a variable, then the mapped variable is

$$V_i \ \mathit{in} \ M$$

where M is a *mapping* of this variable, i.e. an object which encodes all directions necessary to have access to the value of the variable. The concrete form of M depends, of course, on convention. The access method used must be encoded, and some specific information must be provided, such as numerical or symbolic word addresses etc. A configuration, in which all variables are mapped, will be referred to as a *mapped configuration*. E.g., the mapped configuration

$$C^6(s_1 \ \mathit{in} \ \mathrm{A23})(e_2 \ \mathit{in} \ \mathrm{A24,\ A25})$$

may signify that when control in the computer is at the point corresponding to configuration C^6 of the graph of states of the Refal machine, the value of the variable s_1 is stored in the word A23 (symbolic address), and the value of e_2 is stored in the field beginning at A24 and ending at A25.

To start turning a graph of states into a program for a computer we must somehow map the input configuration. In fact, the mapping of the input configuration should be an integral part of the exact formulation of the job. To define the final program uniquely, one must specify not only the input (initial) configuration, but also the way the input data is stored in the computer. The same is true for the output configuration:

the program will vary depending on the representation of the output data we choose.

A compilation task must include a program *P* in Refal, and two mapped configurations: the input configuration C^{in} and the output configuration C^{out}. We will join these two configurations into *the i/o quasisentence:*

$$C^{in} \Rightarrow C^{out}$$

E.g., the i/o quasisentence for the problem considered in Section 4.1 might be

k /L/ GROSS(CAT); ADD(DOG) (e_x *in* R1, R2) ⇒ e_y *in* R1, R3

which means that the input string e_x should be taken from the field with the boundaries stored in registers R1 and R2, while the boundaries of the output string should be stored in R1 and R3.

The i/o quasisentence may be a useful tool even if variables are unmapped. Like a normal sentence, it begins with a concretization sign on the left side paired with the sign ⇒ which separates the two sides, and the right side is the product of the concretization of the left side. But unlike the case of a normal sentence, the variables in the right side of a quasisentence are different from those in the left side and should have differing indexes. We shall make one exception, though. We shall use a variable in the right side with the same index as a variable in the left side, if we know for sure that the value of the right side variable will always be the same as the value of the left-side variable. Another distinction of the quasisentence is that the left side may contain nested concretization brackets. A quasisentence may also be likened to a contraction in that the former, as well as the latter, defines the variables entering the right side through the variables of the left side: a definition of *computed* variables.

The graph of states may have more than one passive configuration by which the concretization process may end. In this case we take an L-generalization of these configurations as the output configuration. Hence it may be necessary to add to the graph of states some transformation arcs leading to the output configuration. The arcs will bear assignments resulting from the recognition of a given finishing configuration as the output configuration. The recognition will always be possible because the output configuration is a generalization, and will always lead to unique assignments because it is an L-expression.

Transformation of a Refal program (or a graph of states of the Refal machine) into a program for the target machine, i.e. the mapping process in the general sense, includes procedures of two types: mapping proper, which refers to variables and configurations, and translation of arcs and other subgraphs of the graph of states (when all of the variables are mapped) into instructions for the target machine. The main unit in the translation process is a *translation statement*. It consists of two parts separated by a horizontal bar, the top part being an element of the graph of states, and the bottom part being its computer equivalent (translation). The final result of the mapping process, as well as intermediate results and some prerequisites, are translation statements.

For a subgraph with mapped input and output configurations the translation sentence is:

$$\frac{\text{<i/o quasisentence>}}{\text{<corresponding computer program>}}$$

If we make a mapping simultaneously with the compilation of the graph of states of the Refal machine, the configurations of i/o quasisentences will be expressed in terms of the original Refal program. If we first compile a graph of states and when this part of the job is finished proceed to map, then configurations in quasisentences will appear in a standard notation (normal form). Translation of external function calls is also performed with the aid of corresponding translation statements. Suppose, e.g., that the target machine has an instruction for addition which in the assembler language (serving as the target language) is written in he form:

$$\text{ADD, A1, A2} \rightarrow \text{A3}$$

(a three-addressed machine). Let us use + as the determiner of the external function perofrming the operation designated by ADD, and let the format be

$$k + (N^1)\ (N^2\ )\ \perp$$

where N^1 and N^2 are the numbers to be added. (We ignore the type of numbers they are and their representatnon, although the user of course should know this). The translation statement which would allow us to use this external function might be:

$$\frac{k + (e_1\ \textit{in}\ \text{A1})\ (e_2\ \textit{in}\ \text{A2}) \Rightarrow e_3\ \textit{in}\ \text{A3}}{\text{ADD, A1, A2} \rightarrow \text{A3}}$$

The essence of the mapping process is: starting with the configurations already mapped, move along the arcs and then map yet unmapped configurations in such a way so as to avoid unnecessary moves of the information in the computer. Contractions are translated into conditional statements and definitions of new variables, whereas assignments are translated into assignments. Decompositions will become procedure calls.

The full state of the Refal machine is not represented, generally, by a configuration, but by a *vertical segment*, i.e. a composition of a number of configurations (a stack of function calls). The configurations (vertices, to be precise) of a graph of states fall into *nonrecurrent*, which do not appear in the vertical segments generated by them, and *recurrent*, which do appear there. Recurrent configurations, in turn, fall into *static* and *dynamic* ones. A configuration is static if it appears only on the top of any of the vertical segments generated by it. Otherwise it is dynamic. Static recurrent configurations correspond to iteration, and should be programmed as nonrecursive procedures. Dynamic configurations correspond to "recursion proper" and must be programmed as recursive procedures. Analyzing the graph of states, it is easy to break down all active configurations into nonrecurrent, static and dynamic.

The graph of states is a sort of flow chart of the future program, written in a special language. This language is rather abstract and doesn't specify some important features of the computer program (the mapping of configurations), so we have some freedom of action for program optimization. Mapping is essentially code generation, with a graph of states as the source program. Methods and techniques of efficient code generation developed by different authors using different source languages can and should be used for mapping.

CHAPTER 5. METASYSTEM TRANSITION

5.1. Metasystem Levels.

Consider some functions defined in a certain *object space*. Consider different procedures of (equivalency) transformation of these functions. To speak of transformations we have some representation of functional definitions. In Refal, functions are defined by sequences of *sentences*. While we do transformations manually, we deal with sentences as objects of a different nature than *object expressions* — the elements of the object space of the functions being transformed. To computerize functional transformations we introduce new functions which deal with objects representing functional definitions. We call these functions *metafunctions* with respect to the original functions. Construction of such functions is a *metasystem transition*. A metasystem transition may be repeated unlimitedly. The original functions defined in the original object space will be referred to as functions of the ground (zero) *metasystem level*. Functions applied to transform (or generate) these functions 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.

The principal idea of the present work is to construct a formal system, in which metasystem transition is one of the formalized operations. Refal has been conceived as the language of this system. Since functions in Refal may be defined only on object expressions, the representation of functional definitions to be used in a metasystem transition must transform sentences (and their parts: free variables, pattern expressions, function calls) into object expressions.

This representation, *Metacode* A was defined in Section 1.3. (We shall just say "metacode", for there won't be any other metacodes used.)

The metacode is designed in such a way that a metasystem transition does not expand the full set of symbols used (otherwise it couldn't be performed unlimitedly by the Refal machine, which doesn't generate new symbols). But it is convenient to use special symbols for the images of free variables, as a part of *shorthand notation*.

On the ground metasystem level, we have such free variables as e_1 , s_a , etc., which represent sets of object expressions. On the first metasystem level these variables will be represented as object expressions *E1, *SA, etc. In shorthand notation we will write a combination of three object signs *E1 as E_1 ; combination *SA will become S_a , etc. These symbols will be called *nonterminal symbols* (or just *nonterminals*) of the first order. On the first metasystem level, nonterminals E_1 etc. (as well as sign combinations *E1 etc.) are dealt with as normal object symbols and expressions, but when we are coming back to the ground level, we interpret them again as sets.

Sure enough, the first metasystem level has its own free variables, which have the usual form e_1 , s_a , s_1, etc. When we make a metasystem transition to the second level, they turn into first-order nonterminals E_1 (i.e. *E1) etc. First-order nonterminals of the first metasystem level turn on the second level into combinations *VE1, *VSA, etc., which will be called nonterminal symbols of the second order and represented in shorthand notation as E_1^2 , S_a^2 , etc. Generally, a nonterminal E_i^n turns into E_i^{n+1} in a metasystem transition. The absence of a superscript means the superscript 1.

The author believes that formalization and computerization of the metasystem transition will have far-reaching consequences because repeated metasystem transitions is the essence of evolutionary processes and, in particular, it is a powerful instrument of creative human thinking (see [2]). To solve a problem we first try to use some standard system of operations, rules, etc. If we fail, we start to analyze *why* we failed, and for this purpose we examine the *process of applying* our rules and operations. This is a metasystem transition. We try to 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 as the instruments to succeed in solving 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 try to 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 an analysis of its use and creation of instruments to produce instruments may be repeated again and again, and 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.

We may construct a high tower of metasystem levels, but our ultimate goals will stay on the earth, expressed in terms of the ground metasystem level (the proof and use of all of the mathematics is ultimately with such tangible things as numbers and geometrical figures). As a complement to climbing up the metasystem stairway there must be some *reduction* of higher-level constructs to lower-level constructs. The specific rule of reduction will in each specific case be defined by the specific goal we pursued in making the metasystem transition. General laws and rules may also exist. This is one. Consider classes of expressions on the first metasystem level. The pattern expression which represents a class may include nonterminals, e.g.

(1) $$E_1\ s_1\ (A\ e_2)$$

This is a class which includes such object expressions as

(1.1) E_1 A (A)

(1.2) E_1 B (A BC+)

(1.3) E_1 B (A E_6 E_1 S_2 END)

etc. Each of these object expressions of the first metasystem level can be interpreted as a class of object expressions of the ground level by applying the inverse metacode transformation. E.g. the first of these will be translated into

(1.1') e_1 A (A)

and so on. Now the question arises: what set of object expressions of the ground level corresponds to a class (1) on the first level? In other words, what is the union of all classes (1.1), (1.2), (1.3), etc.?

The answer, as one can easily see, is very simple and may be expressed by the following *reduction rule*: a class of classes is a class. To turn a class of the first metasystem level into the corresponding class of the ground level we only have to turn nonterminals of the first order into free variables with new indexes.

5.2. Graph of States as a Production System.

On the first metasystem level the variables in the graph of states pass into nonterminal symbols. Contractions may be interpreted now as production rules for nonterminals. Assignments may also be interpretedas production rules but they are read from right to left and obey the corresponding law of composition (see Section 4.2, page 83). We introduce now a new type of nonterminal symbol in order to represent all information about an arc in the form of production rules. To each active nonterminal vertex V_i characterized by configuration C^i we put into correspondence a nonterminal symbol K_i , which should eventually produce the set of all possible concretizations of C^i. The transition from an active vertex V_i to another (active or passive) vertex by a dynamic arc is a contraction for the nonterminal symbol K_i, because it generally limits the set attached to (eventually produced by) K_i by changing K_i into an expression which either is more specific already, or, at least, is closer to the completion of concretization.

Example 1

Consider as an example (not for the first time already!) the function which turns pluses into minuses, with the graph of states in Figure 17.

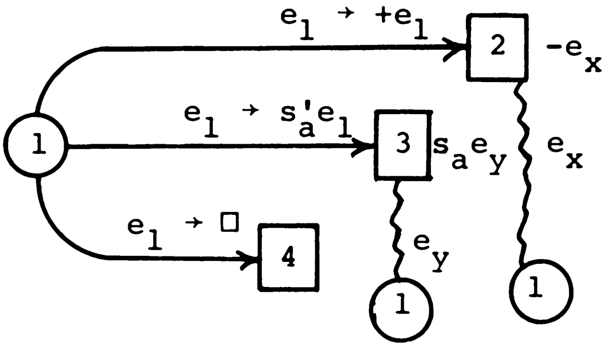


Figure 17

The first horizontal arc here will be represented by two successive productions:

$$(E_1 \to +E_1)\ (K_1 \to -E_x)$$

The vertical line from C^2 to C^1 will become

$$(K_1 \leftarrow E_x)$$

We can combine (informally, for the time being) these two production sequences into one:

(1) $$(E_1 \to +E_1)\ (K_1 \to -K_1)$$

Analogously, the second and third horizontal arcs will generate sequences:

(2) $$(E_1 \to S'_aE_1)(K_1 \to S_aK_1)$$

(3) $$(E_1 \to □)\ \ (K_1 \to □)$$

(Nonterminal symbol S_a in the production for E_1 in (2) is primed, as was the variable s_a in the corresponding contraction. Soon we shall see why this is important and how it is used.)

Production sequences (1), (2) and (3) are not independent, because the arcs in the graph of states were not independent: the quasiinput set that corresponds to each arc depends on the preceding arcs. But we can transform the graph of states to make all arcs independent by adding necessary restrictions to the contractions on the arcs. As we saw in Section 4.3, these restrictions generally have the form of a *constriction sum*:

(4) $$\Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$$

where *constriction terms* Σ_i for each i = 1,2,...,n may be reduced to the form:

$$\Sigma_i = // \; \delta^+_{i1} \; // \; \delta^+_{i2} \; \ldots \; // \; \delta^+_{iq_i} \; \backslash\backslash \; \delta^-_{i1} \; \ldots \; \backslash\backslash \; \delta^-_{ir_i}$$

with all contractions preceding restrictions. Moreover, restrictions $\backslash\backslash\ \delta^-_{ij}$ may belong only to one of the four types described in Section 4.3, and correspond to a set known to be nonempty (Theorem 4.3).Both q_i and r_i may be zero (although not simultaneously).

If sum (4) has more than one term, we shall introduce an additional dynamic arc for each term; each additional arc will lead to a new (additional) vertex. Each of the additional vertices will have to be explored, thus the graph of states may become essentially larger as a result of this transformation.

Restrictions on free variables will pass into restrictions on nonterminals, which we shall write immediately after the corresponding contraction. In our example the second arc (or path, to be exact) will change into:

(2') $$(E_1 \rightarrow S'_a E_1) \; (\backslash\backslash \; S_a \rightarrow + \;) \; (K_1 \rightarrow S_a K_1)$$

Using primary production sequences corresponding to the paths of the graph of states we may build sequences corresponding to walks by mere concatenation of primary sequences. E.g., to the walk

$$1,2[1,3[1,3[1,4]]]$$

the following production sequences corresponds:

$$(5)\quad \begin{array}{l} (E_1 \to +E_1)\ (K_1 \to -K_1)\ (E_1 \to S'_aE_1)\ (\backslash\backslash S_a \to +)\ (K_1 \to S_aK_1) \\ (E_1 \to S'_bE_1)(\backslash\backslash S_b \to +)\ (K_1 \to S_bK_1)\ (E_1 \to \square\ )\ (K_1 \to \square) \end{array}$$

Essentially, this is a driving operation, hence the primed nonterminals behave like primed variables in driving: they generate a nonterminal with a new index, which should be substituted for all (primed and unprimed) entries of the original nonterminal in the primary production sequence (path).

If we perform, step by step, all the substitutions called for in the production sequence (5), we shall see that it is equivalent to the following production sequence:

$$(6)\quad (E_1 \to +S_aS_b)\ (\backslash\backslash S_a \to +)\ (\backslash\backslash S_b \to +)\ (K_1 \to -S_aS_b)$$

which clearly provides the definition of our original function for a special class of arguments.

From now on, we shall call production sequences like (5) and (6), corresponding to walks, just *walks*. Equivalency transformation as of (5) into (6) will be referred to as *normalization*. Unlike our previous notation for walks, which served only to refer to a graph of states reproduced elsewhere, our new walks are selfsufficient objects, which are constructed and transformed by certain rules, and have an interpretation.

Our target now is to build *productions for walks* so as to examine the set of all walks. For this purpose we introduce one more type of nonterminal, nonterminal symbols of second order, which we shall denote as W_i (instead of the standard E_i^2)

in order to stress that they generate sets of walks. To distinguish productions of the second level (W-*productions*) from productions of the first level, we shall use the double arrow ⇒ in recording these productions.

To each active nonterminal vertex V_i we put into correspondence the nonterminal symbol W_i , which eventually generates all walks beginning at vertex V_i. Now we can rewrite the whole graph of states as a production system of second order, where to each horizontal arc exactly one W-production corresponds. For the graph of states in Figure 17 the production system will be:

(7.1) $W_1 \Rightarrow (E_1 \rightarrow +E_1)(K_1 \rightarrow -K_1)W_1$

(7.2) $W_1 \Rightarrow (E_1 \rightarrow S'_a E_1)(\backslash\backslash S_a \rightarrow +)(K_1 \rightarrow S_a K_1)W_1$

(7.3) $W_1 \Rightarrow (E_1 \rightarrow \square)(K_1 \rightarrow \square)$

Production rules (7.1) and (7.2), corresponding to the paths which lead to active vertices, end with nontermimal W-symbols requiring continuation of the walk. Production (7.3) corresponds to a path leading to a passive vertex and does not have in the right side W-symbols; it consumates the process of the production of a walk.

Production system (7) is a classical context-free grammar, generating the set of all possible (in the given graph of states) walks. The graph of states in the original form is regarded as a function definition. To use it we assign some values to the input variables, and generate in the process of computation a unique walk, because at each branch we must take a unique dynamic arc. Now we regard the graph of states as a production system, and at each branch we make an arbitrary choice as to which arc to take. Thus we can produce any walk. Normalizing this walk we obtain:

(1) the input set for this walk defined by contractions for the input variables;

(2) the *output set*, i.e. the set of all possible results of concretization of the starting configuration;

(3) the mapping of the input set on the output set, (2) and (3) being defined through the contraction for the

starting K-symbol.

Therefore we receive a nonrecursive *one-step* subfunction of the original function. It is identical to the original function on the input set and undefined outside.

This is the subfunction for those arguments which take the chosen walk in the process of computation. Its definition by the normalized walk is essentially the same as in a Refal sentence with no concretization signs in the right side (the only difference is that restrictions are indicated explicitly).

Consider now the case when there are composition loops in the graph of states. To the subgraph represented in Figure 18 the following W-production will correspond:

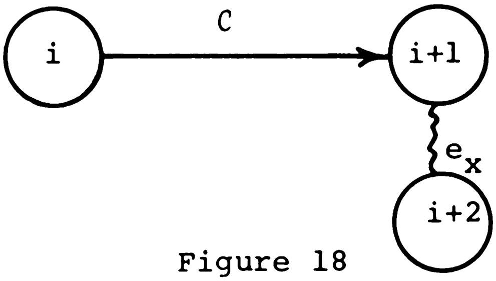


Figure 18

$$W_i \Rightarrow C(K_{i+2} \leftarrow E_x)\ [W_{i+2}]\ (K_i \rightarrow K_{i+1})\ W_{i+1}$$

Here C is the record of constrictions borne by the dynamic arc i, i+1. Should this arc be transformational, then assignments will take the place of C.

Thus a composition loop adds the assignment of its starting K-symbol to the computed variable, followed by a bracketed W-symbol for the walk representing concretizations of the inner configuration. The bracketed nonterminal will produce walks, therefore every walk finally produced by W_i will not be a simple sequence of first-level productions, but an *expression* (a *tree*) of productions. This should be reflected in the rules of normalization. We use brackets (instead of parentheses) to structure these expressions for the sake of convenience only. From the formal point of view, brackets are the same as parentheses.

Example 2

As a further example let us take the function of double scanning considered (and transformed) in Section 4.5 (page 120):

$$\begin{array}{lll}
\alpha\ e_1 & \Rightarrow & kF^b\ kF^a\ e_1 \perp \perp \\
kF^a A e_1 & \Rightarrow & B\ kF^a\ e_1 \perp \\
kF^a s_1 e_2 & \Rightarrow & s_1\ kF^a\ e_2 \perp \\
kF^a & \Rightarrow & \\
kF^b B e_1 & \Rightarrow & C\ kF^b\ e_1 \perp \\
kF^b s_1 e_2 & \Rightarrow & s_1\ kF^b\ e_2\ \perp \\
kF^b & \Rightarrow &
\end{array}$$

The graph of states for this function is represented in Figure 15 (page 121). The production system for it is:

(8.1) $W_1 \Rightarrow (K_3 \leftarrow E_b)\ [W_3]\ (K_1 \rightarrow K_2)\ W_2$

(8.2) $W_2 \Rightarrow (E_b \rightarrow BE_b)\ (K_2 \rightarrow CK_2)\ W_2$

(8.3) $W_2 \Rightarrow (E_b \rightarrow S'_3 E_b)\ (\backslash\backslash S_3 \rightarrow B)\ (K_2 \rightarrow S_3 K_2)\ W_2$

(8.4) $W_2 \Rightarrow (E_b \rightarrow \Box)\ (K_2 \rightarrow \Box)$

(8.5) $W_3 \Rightarrow (E_1 \rightarrow AE_1)\ (K_3 \rightarrow BK_3)\ W_3$

(8.6) $W_3 \Rightarrow (E_1 \rightarrow S'_2 E_1)\ (\backslash\backslash S_2 \rightarrow A)\ (K_3 \rightarrow S_2 K_3)\ W_3$

(8.7) $W_3 \Rightarrow (E_1 \rightarrow \Box)\ (K_3 \rightarrow \Box)$

To establish the rules of the walk normalization we must give a more systematic treatment of walks.

Walks are composed of first-order productions and restrictions, and brackets. In the process of a walk prcduction we are dealing also with *nonterminal walks*, which in addition contain second order nonterminals W_i. First order productions are bidirectional substitutions for first order nonterminals E_i , S_i , K_i . A bidirectional substitution consists of the *antecedent*, which is a nonterminal, and the *consequent*, which is an expression. The antecedent may be either on the left, or on the right side of the substitution; it is separated from the consequent by the arrow → or ← , its direction being from the antecedent to the consequent.

There are three types of bidirectional substitution: contractions, assignments and *replacements*. We have familiarized ourselves with contractions, restrictions, and assignments pretty well already. Replacements have the form of a contraction for a K-symbol, but their meaning is different. A replacement does not define the variables entering the consequent, as a contraction for an E-symbol does. Neither does it restrict the input set, as contractions for both E- and S-symbols, and restrictions do. Like assignments, replacements are used to attach a value to the antecedent, but this is a value (replacement) for a nonterminal symbol *already defined*. Moreover, a replacement will be applied to *exactly one* entry of the antecedent nonterminal, not to *all* of the entries identical to the antecedent, as is the case for assignments.

We call a walk *terminal*, when we want to stress that it does not contain W-symbols (is not *nonterminal*). A terminal walk may be interpreted as a function definition. The process of interpretation is the reproduction of the steps the Refal machine will make when concretization takes place as indicated in the walk.

By metasystem reduction, we interpret nonterminal S- and E-symbols as corresponding free variables in the graph of states. Nonterminal K-symbols are interpreted as configurations of the Refal machine. The full state of the Refal machine is given when a vertical segment is given, as well as the values of all the variables in all configurations of the segment. But we have only one walk, and we have no way to derive from it the full list of all configurations in the graph of states. Still worse, even for those configurations which appear in the walk, coded by K-symbols, we have no way of knowing what their full lists of arguments are. This becomes dramatically evident when an argument is present in the definition of a function and all auxiliary functions, on which nothing actually depends. Even if we know all possible walks in the graph of states, we shall never suspect that this argument is present in all configurations. A more realistic case is when an argument does have an impact on the process of concretization, but not along the walk we are examining.

In the course of interpretation, we will maintain a *stack of variables* as a model of the full state of the Refal machine. The stack of variables will be a sequence of *fields*, each field being in correspondence with a configuration in the vertical segment representing the full state. A field is a sequence of assignemtns, in which the consequents are object èxpressions. To determine the value of a variable we, as is usual with stacks, find the last assignment for this variable, ignoring the boundaries between fields. The procedure of interpretation is such that a variable appears in the stack either when it is used as an input variable, or when it is defined as a generalization or computed variable. Before it appears in one of these two capacities (if at all), the variable is ignored in the process of interpretation, because it does not influence the evolution of the configuration in the Refal machine.

Besides the stack of variables, we shall keep track of *the replacement rule*, which is initially empty (nonexistent), appears at some stage of interpretation, undergoes transformations, and eventually provides the final result of interpretation: a replacement

$$K_i \rightarrow \mathcal{E}$$

where K_i is the starting configuration, and $\mathcal{E}$ is the result of its concretization. It is convenient to represent the stack of variables as a column of assignments, with horizontal bars separating fields, and with the replacement rule positioned at the top (beginning) of the stack. In the following, the replacement rule will be considered as a part of the stack.

In the process of interpretation we scan the walk from left to right performing the following. (At the beginning the stack is empty.)

(1) A contraction. If the antecedent is not defined we declare it an input variable and obtain its value through an input procedure. If the antecedent is defined, i.e. there is an assignment for it in the stack of variables, we take its value from the stack. In both cases, we then apply the contraction to the value of the antecedent variable. If there are variables in the right side (consequent) which are not defined or redefined in the contraction, we either take their values from the stack of variables, or, if there is no corresponding line, declare them input variables and use the input procedure. (We recall that an e-variable appearing in the right side of a contraction is always (re)defined. A primed s-variable is defined, a nonprimed one is not defined.) If the contraction succeeds, we update the stack of values (in particular, new lines may be added, if new variables are defined in contraction). If it fails, we stop the interpretation with the conclusion that the *exact input state* (see Section 4.5, page 111) is outside the input set for the interpreted walk.

(2) A restriction. We check the condition, using the values of the variables from the stack. If it is not satisfied, we stop the process. A variable on either side of the restriction, for which there is no line in the stack, is treated as an input variable.

(3) An assignment. We change the last line for the antecedent variable in the stack if it is there, otherwise we add a new line. The values of the variables in the consequent (left side) are taken from the stack or through the input procedure, as described in (1).

(4) A replacement. We replace the values of the variables in the consequent (right side) by their values. Then we examine all the consequents in the stack of variables (including the replacement rule) and find the last nonterminal K-symbol with the same index as at the antecedent of the current replacement.

Then we replace it with the consquent of the current replacement. 3y the last K-symbol we mean the last *in time*. To find it easily we shall mark each K-symbol appearing in the consequents in the stack with its sequential number written as a superscript. If there is no K-symbol with the same index to be found in the stack, we write the whole current replacement as the replacement rule in the stack.

(5) A left bracket. At the end of the stack, we add a bar separating fields.

(6) A right bracket. We eliminate the last field in the stack.

(7) The end of the walk. We eliminate the assignments and leave only the replacement rule in the stack. This is the final product.

Let us come back to Example 1 for an illustration. Consider the walk (5). Take +XY as the input value for e_1. Let us follow the process of interpretation.

At the first step we execute the contraction $(E_1 \to +E_1)$. Since the stack of values is still empty, we declare E_1 in the left side as an input variable, get the value +XY for it (input procedure), and start the stack of values by writing in the line

$$+XY \leftarrow E_1$$

Immediately, we apply the contraction, which is successful and changes this line into

$$XY \leftarrow E_1$$

Now we execute the second production — the replacement $(K_1 \to -K_1)$. There is no replacement rule in the stack yet, therefore we put this replacement in the stack, which becomes

$$\frac{K_1 \to -K_1^1}{XY \leftarrow E_1}$$

Executing the third production, we once more change the value of E_1 and define a new variable S_a. The stack becomes:

$$\begin{array}{c} K_1 \to -K_1^1 \\ \hline Y \leftarrow E_1 \\ X \leftarrow S_a \end{array}$$

Next we check the restriction (\\ $S_a \to +$), which holds because the value of S_a differs from + . Executing the replacement ($K_1 \to S_aK_1$), we first turn the variable S_a in the right side to its value found in the stack: ($K_1 \to XK_1$). Now we find the last (and only) symbol K_1 in the *consequents* of the stack, which is K_1^1 , and replace it by XK_1. The stack becomes:

$$\begin{array}{c} K_1 \to -XK_1^2 \\ \hline Y \leftarrow E_1 \\ X \leftarrow S_a \end{array}$$

Proceeding in this way we ultimately get the result:

$$K_1 \to -XY$$

Consider now a more complicated example, where a function invokes itself with an argument which includes a call of the same function.

<u>Example 3</u>

$$\phi\ A\ e_1 \Rightarrow B\ \phi\ e_1\ \phi\ e_1 \perp \perp$$

$$\phi\ e_1 \Rightarrow e_1$$

We take AA for the argument e_1 of function ϕ. The following sequence of view-fields results from concretization:

$$\phi\ A\ A \perp$$

$$3\ \phi\ A\ \phi\ A \perp \perp$$

$$B\ \phi\ A\ B\ \phi\ \phi \perp \perp \perp$$

$$B\ \phi\ A\ B\ \ \ \phi \perp \perp$$

$$B\ \phi\ A\ B \perp$$

$$B\ B\ \phi\ B\ \phi\ B \perp \perp$$

$$B\ B\ \phi\ B\ B \perp$$

$$B\ B\ B\ B$$

The graph of states for the function ϕ is presented in Figure 18.

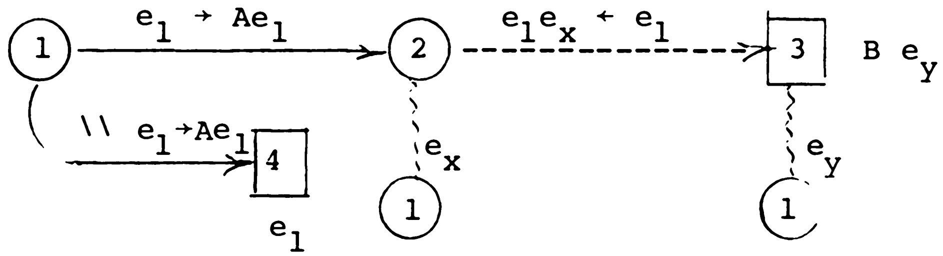


Figure 18

The corresponding production system is:

$$W_1 \Rightarrow (E_1 \rightarrow AE_1)\ (K_1 \leftarrow E_x)\ [W_1]\ (K_1 \rightarrow K_2)\ W_2$$

$$W_1 \Rightarrow (\backslash\backslash\, E_1 \rightarrow AE_1)\ (K_1 \rightarrow E_1)$$

$$W_2 \Rightarrow (E_1E_x \leftarrow E_1)\ (K_2 \rightarrow BK_1)\ W_1$$

One can see that the nonrecurrent nonterminal symbol W_2 can be eliminated (we do it informally at this time), which transforms the production system into:

$$W_1 \Rightarrow (E_1 \rightarrow AE_1)\ (K_1 \leftarrow E_x)\ [W_1]\ (K_1 \rightarrow BK_1)\ (E_1E_x \leftarrow E_1)\ W_1$$

$$W_1 \Rightarrow (\backslash\backslash\, E_1 \rightarrow AE_1)\ (K_1 \rightarrow E_1)$$

The walk which will be taken by the Refal machine with the initial view-field ϕ A A $\perp$, consists of twenty productions Below we reproduce the process of interpretation of this walk, giving the stack of variables at each stage (*after* performing the substitution).

1	2	3	4	5
$E_1 \rightarrow AE_1$	$K_1 \leftarrow E_x$	$[E_1 \rightarrow AE_1$	$K_1 \leftarrow E_x$	$[\backslash\backslash E_1 \rightarrow AE_1$
$A \leftarrow E_1$	$A \leftarrow E_1$ $K_1^1 \leftarrow E_x$	$A \leftarrow E_1$ $\underline{K_1^1 \leftarrow E_x}$ $\square \leftarrow E_1$	$A \leftarrow E_1$ $\underline{K_1^1 \leftarrow E_x}$ $\square \leftarrow E_1$ $K_1^2 \leftarrow E_x$	$A \leftarrow E_1$ $\underline{K_1^1 \leftarrow E_x}$ $\square \leftarrow E_1$ $\underline{K_1^2 \leftarrow E_x}$
6	7	8	9	10
$K_1 \rightarrow E_1]$	$K_1 \rightarrow BK_1$	$E_1E_x \leftarrow E_1$	$\backslash\backslash E_1 \rightarrow AE_1$	$K_1 \rightarrow E_1]$
$A \leftarrow E_1$ $\underline{K_1^1 \leftarrow E_x}$ $\square \leftarrow E_1$ $\square \leftarrow E_x$	$A \leftarrow E_1$ $\underline{BK_1^3 \leftarrow E_x}$ $\square \leftarrow E_1$ $\square \leftarrow E_x$	$A \leftarrow E_1$ $\underline{BK_1^3 \leftarrow E_x}$ $\square \leftarrow E_1$ $\square \leftarrow E_x$	No Change	$A \leftarrow E_1$ $B \leftarrow E_x$
11	12	13	14	15
$K_1 \rightarrow BK_1$	$E_1E_x \leftarrow E_1$	$E_1 \rightarrow AE_1$	$K_1 \leftarrow E_x$	$[\backslash\backslash E_1 \rightarrow AE_1$
$\underline{K_1 \rightarrow BK_1^4}$ $A \leftarrow E_1$ $B \leftarrow E_x$	$\underline{K_1 \rightarrow BK_1^4}$ $AB \leftarrow E_1$ $B \leftarrow E_x$	$\underline{K_1 \rightarrow BK_1^4}$ $B \leftarrow E_1$ $B \leftarrow E_x$	$\underline{K_1 \rightarrow BK_1^4}$ $B \leftarrow E_1$ $K_1^5 \leftarrow E_x$	$\underline{K_1 \rightarrow BK_1^4}$ $B \leftarrow E_1$ $\underline{K_1^5 \leftarrow E_x}$
16	17	18	19	20 END
$K_1 \rightarrow E_1]$	$K_1 \rightarrow BK_1$	$E_1E_x \leftarrow E_1$	$\backslash\backslash E_1 \rightarrow AE_1$	$K_1 \rightarrow E_1$
$\underline{K_1 \rightarrow BK_1^4}$ $B \leftarrow E_1$ $B \leftarrow E_x$	$\underline{K_1 \rightarrow BBK_1^6}$ $B \leftarrow E_1$ $B \leftarrow E_x$	$\underline{K_1 \rightarrow BBK_1^6}$ $BB \leftarrow E_1$ $B \leftarrow E_x$	No Change	$K_1 \rightarrow BBBB$

A walk is said to be in *normal form* if:

(1) it has no assignments, no composition loops;

(2) it has exactly one replacement, which makes it end;

(3) there is no more than one contraction for each variable.

Any walk may be brought to a normal form — *normalized*. The normalization process is, essentially, interpretation of the walk with an unknown exact input state. We scan the walk from left to right, but instead of checking constrictions we accumulate them at the beginning (left end) of the walk; and instead of maintaining a stack of variables, where all their values are kept in their final form (as object expressions), we accumulate assignments containing free variables.

First we consider a walk without composition loops. It is a sequence of constrictions, assignments and replacements. We note the following.

(1) Because of the absence of composition loops, the assignments may not contain nonterminal K-symbols.

(2) Nonterminal K-symbols appear in the walk exactly in the same order they will appear in the stack during interpretation. If we make it a rule that we never transpose replacements in the process of normalization,(*) we can be sure that to use a replacement we should apply it to the nearest consequent on the left.

(3) An assignment which ends a walk may be eliminated, because it will have no effect on the interpretation.

The principal idea of normalization is: keeping replacements in their positions relative to one another, move constrictions to the left, and assignments to the right, by *commuting* (transposing) them with neighboring substitutions; ultimately, the assignments are eliminated at the right end, the constrictions accumulate at the left end and get simplified according to the rules of Section 4.3, and replacements, when they are not separated any more by other productions, combine into one final replacement.

(*) A weaker form: those replacements which involve the same K-symbol(s).

The *range* of a bidirectional substitution in a walk is constituted by all those substitutions whose final effect will be influenced by the substitution in question. It is the part of the walk stretching in the direction opposite to the direction of the arrow that will be influenced by a given substitution. Therefore, the range of a constriction or replacement is to the left until the beginning of the walk, and the range of an assignment is to the right until the end. A walk can be thought of as something to be applied to the list of input variables placed at the left end; this is why constrictions and replacements constitute the final normalized walk, and assignments are thrown away.

Let us now formulate commutation rules for bidirectional substitutions. In the following, V_i is an s- or e-variable (nonterminal symbol), R_i is any expression, possibly containing nonterminal K-symbols, E_i is any pattern expression (i.e. not containing K-symbols), L_i is any L-expression; i, as well as j, x, etc. are arbitrary indexes.

Assignment-replacement

(ARP) $(E_i \leftarrow V_i)(K_j \rightarrow R_j) \Rightarrow (K_j \rightarrow (E_i \leftarrow V_i)//\ R_j)(E_i \leftarrow V_i)$

Assignment-contraction

(AC.D) $(E_i \leftarrow V_i)(V_i \rightarrow L_i) \Rightarrow (E_i \rightarrow L_i)$

This rule is written in a symbolic form. By $E_i \rightarrow L_i$ we mean that E_i should be *syntactically recognized* as L_i, and the result should be represented as a sequence of constrictions for the variables in E_i followed by a sequence of assignments for the variables in L_i:

(9) $\ldots (V_{ix} \rightarrow L_{ix}) \ldots (\backslash\backslash\ V_{iy} \rightarrow L_{iy}) \ldots (E_{iz} \leftarrow V_{iz}) \ldots$

As a result of recognition (using the GPA) we may receive more than one class constituting the intersection $E_i \cap L_i$. Then for each of the sequences (9) we make a separate copy of the walk and continue normalization of each walk independently.

Thus we see that normalization of *one* walk may result in *several* normalized walks. Essentially, (AC.D) is the rule of driving.

In the sequence (9) all constrictions precede all assignments; therefore this commutation rule, like the other rules, takes us one step further along the way of normalization. In some cases, though, we may retain the form $E_i \to L_i$ as a sort of contraction, for the sake of brevity. Like normal contractions, this form defines the values of some variables on the right side through the variable(s) in the left side.

Consider now the case of $i \neq j$:

$$(E_i \leftarrow V_i)\ (V_j \to L_j)$$

Both substitutions are placed in the range of one another; therefore both consequents are subject to modification. But we are going to show that it will always suffice to modify only one consequent.

If E_i contains V_j , but L_j does not contain V_i , the commutation rule will be:

(AC.1) $(E_i \leftarrow V_i)\ (V_j \to L_j) \Rightarrow (V_j \to L_j)\ (E_i\ //\ (V_j \to L_j) \leftarrow V_i)$

and this is a most usual case when representation E_i of a value in the stack of values must be updated because a variable V_j present in E_i undergoes a contraction.

Consider the other possibility: when V_i enters L_j. Suppose V_i is an e-variable, e_i. Then e_i cannot enter L_j , because it is different from e_j , but all other e-variables in L_j must be new, hence may not be identical to e_i , which has been in use before. Thus, V_i is $s(P)_i$. This is an example, when the assignment does influence the contraction:

$$(A \leftarrow s_1)\ (s_2 \to s_1)$$

Can this combination appear in an actual walk? In a walk resulting from a production system without compositions it cannot. Because the contraction for s_2 may appear **only in the driving of the** preceding configuration. If in that configuration s_1 was assigned the value A, it must have disappeared from the configuration (being replaced by A), thus it cannot appear as an old variable in any subsequent contraction. But, as we shall see later, an assignment may appear on a given level of a walk as a result of a *contraction* made in a function loop on a deeper level. E.g., the following sentences

$$\phi\ s_1\ s_2 \Rightarrow \phi^1(s_1)(s_2)(\phi^2 s_1 \perp) \perp$$

$$\phi^1(s_1)(s_1)(s_3 e_4) \Rightarrow \ldots$$

$$\ldots$$

$$\phi^2 A \Rightarrow \text{MEAT}$$

$$\ldots$$

will produce the combination in question.

The commutation rule for the case when the assignment influences the contraction is:

(AC.2) $(E_i \leftarrow V_i)\ (V_j \rightarrow L_j) \Rightarrow (V_j \rightarrow (E_i \leftarrow V_i)//\ L_j)(E_i \leftarrow V_i)$

Can it be that both substitutions influence one another? We have established that if the assignment influences the contraction, V_i is $s(P)_i$. Therefore, E_i is syntactically a symbol, i.e. either a specific symbol, or $s(Q)_x$. But for the contraction to influence the assignment, V_j must enter E_i. Therefore, x is identical to j, and both E_i and V_j are $s(Q)_j$. But since V_j is an s-variable, L_j must be syntactically a symbol, and since $s(P)_i$ must enter it, it must be identical to $s(P)_i$. Our case becomes:

$$(s(Q)_j \leftarrow s(P)_i)\ \ (s(Q)_j \rightarrow s(P)_i)$$

Which of the two AC rules should be applied here? Undoubtedly, (AC.2), not (AC.1). If the left substitution were a "real" assignment, there could be no contraction later using $s(P)_i$, because it would have been replaced by $S(Q)_j$. Therefore the assignment is a transformed contraction, and the case should be covered by Rule (AC.2). Applying this rule and cancelling an identity substitution, we get the rule

(AC.3) $(s(Q)_j \leftarrow s(P)_i)\ (s(Q)_j \rightarrow s(P)_i) \Rightarrow (s(Q)_j \leftarrow s(P)_i)$

Finally, we can sum up the AC commutation rule for different variables in the following way: if the assignment influences the contraction, apply Rule (Ac.2) or (Ac.3); otherwise, apply Rule (AC.1).

Assignment-restriction

Since a restriction $\backslash\backslash V_j \rightarrow L_j$ does not (re)define any variables, but only checks a condition for V_j , it is commutable with any substitution which does not define V_j. Thus for $i \neq j$

(AR) $(E_i \leftarrow V_i)\ (\backslash\backslash\ V_j \rightarrow L_j) \Rightarrow (\backslash\backslash\ V_j \rightarrow L_j)\ (E_i \leftarrow V_i)$

For i = j we formulate a symbolic rule, like we did in the case of contraction:

(AR.D) $(E_i \leftarrow V_i)(\backslash\backslash V_i \rightarrow L_i) \Rightarrow (\backslash\backslash\ E_i \rightarrow L_i)\ (E_i \leftarrow V_i)$

Using the technique described in Section 4.3 we transform symbolic restriction $(\backslash\backslash\ E_i \rightarrow L_i)$ into a set,or several sets of standard restrictions for the variables in E_i.

Replacement-contraction

(RPC) $(K_i \rightarrow R_i)\ (V_j \rightarrow L_j) \Rightarrow (V_j \rightarrow L_j)\ (K_i \rightarrow R_i\ //\ (V_j \rightarrow L_j)\ )$

Replacement-restriction

(RPR) $(K_i \rightarrow R_i)(\backslash\backslash V_j \rightarrow L_j) \Rightarrow (\backslash\backslash V_j \rightarrow L_j)(K_i \rightarrow R_i)$

This completes the case of no composition loops. We turn now to the general case.

A composition loop consists of an assignment for the computed variable of the loop and a bracketed walk:

$$(R_x \leftarrow E_x)\ [W]$$

Unlike other assignments, an assignment for a computed variable contains exactly one K-symbol. Hence the rule: the position of computed variable assignments with respect to one another and to replacements must not be changed. Therefore other assignments should be commuted with computed variable assignments like they are commuted with replacements:

(CM.1) $(E_i \leftarrow V_i)(R_x \leftarrow E_x) \Rightarrow ((E_i \leftarrow V_i)//R_x \leftarrow E_x)(E_i \leftarrow V_i)$

The main rule of dealing with brackets may be formulated as the "*transparency principle*": the left bracket is transparent, and the right is not.

Assignments (not for the computed variable, of course) enter brackets from the left, to be used in all lower-level loops (global variables). A copy of each assignment jumps over the brackets, to be used on the main level:

(CM.2) $(E_i \leftarrow V_i)[W] \Rightarrow [(E_i \leftarrow V_i)\ W]\ (E_i \leftarrow V_i)$

Inside the brackets, assignments move to the right and disappear at the right bracket together with those assignments which were borne inside the brackets (local variables).

Contractions borne inside the brackets are *global*, not local. They move to the left, and get out through the left bracket (the principle of transparency). A *reversed copy* of the contraction, i.e. an assignment which expresses the old variable through the new ones, is put behind the right bracket:

(CM.3) $(R_x \leftarrow E_x)[(V_i \rightarrow L_i)W] \Rightarrow (R_x \leftarrow E_x)(V_i \rightarrow L_i)[W](L_i \leftarrow V_i)$

Restrictions born inside the brackets get out through the left bracket. Replacements go out through the left bracket and are applied to R_x in the computed variable assignment. When the brackets are empty, they are eliminated At that time R_x will contain no K-symbols, so that the assignment for the computed variable E_x becomes a regular assignment, and normalization continues.

The rules we formulated are enough to normalize any walk, but to facilitate the process of normalization we introduce one more concept: the *conjunction* of substitutions of the same type but for different variables. The operation of conjunction is represented by joining the operands with the plus sign, and it means that all of the substitutions conjoined should be applied *simultaneously*.

Let, e.g., variable e_1 have value X, and variable e_2 — value Y. A sequence of two assignments, e.g.

(10) $(e_1A \leftarrow e_2)\ (e_2B \leftarrow e_1)$

means the *composition* of the constituents. As a result of reading it from left to right, e_2 will take the value XA, and then e_1 will take value XAB, because at the time the second assignment is used, the value of e_2 has been changed by the first assignment. When we use the conjunction

(11) $(e_1A \leftarrow e_2) + (e_2B \leftarrow e_1)$

we form new values for all variables simultaneously. The resulting value for e_2 will be the same (because in (10) the first assignment is out of the range of the second anyway), but e_1 will take a different value: YB. To represent (10) as a conjunction (to *conjoin* the substitutions) we take into account their interaction:

(12 = 10) $(e_1A \leftarrow e_2) + (e_1AB \leftarrow e_1)$

In general form, the rules for conjoining two assignments are:

(one variable) $(E_1 \leftarrow V_i)(E_2 \leftarrow V_i) \Rightarrow ((E_1 \leftarrow V_i)\ //E_2 \leftarrow V_i)$

(different variables) $(E_i \leftarrow V_i)(E_j \leftarrow V_j) \Rightarrow (E_i \leftarrow V_i)+((E_i \leftarrow V_i)//E_j \leftarrow V_j)$

The rules for conjoining constrictions were presented in Section 4.3 (although we did not use this term at that time). Replacements are never conjoined in the process of normalization, but whenever we have a pair of adjacent replacements we immediately use the composition rule:

$$(K_i \rightarrow C_1K_jC_2)\ (K_j \rightarrow R_j) \Rightarrow (K_i \rightarrow C_1R_jC_2)$$

where C_1 and C_2 are multibrackets.

Conjunction is obviously associative and commutative. It is convenient to agree that the operation + ties closer than concatenation. The conjunction rules for a pair of substitutions can be easily generalized to any number of constituents by adding a sort of associativity law for composition (concatenation) and conjunction. Thus normalization becomes an equivalency transformation over expressions built of bidirectional substitutions using two operations: concatenation and conjunction. The rules of transformation are: commutation rules for substitutions of different type, and conjunction rules for substitutions of the same type.

Consider an example of normalization. Remember production system (8) in Example 2. Let us form a walk by using production rules in the following sequence: (.1), (.6), (.7), (.3), (.4) . It will be:

$$(K_3 \leftarrow E_b)^1 [(E_1 \rightarrow S'_2E_1)^2 (\backslash\backslash S_2 \rightarrow A)^3 (K_3 \rightarrow S_2K_3)^4 (E_1 \rightarrow \square)^5 (K_3 \rightarrow \square)^6]$$
$$(K_1 \rightarrow K_2)^7 (E_b \rightarrow S'_3E_b)^8 (\backslash\backslash S_3 \rightarrow B)^9 (K_2 \rightarrow S_3K_2)^{10} (E_b \rightarrow \square)^{11} (K_2 \rightarrow \square)^{12}$$

(The superscripts are sequential numbers of productions for easy reference: P^1 , P^2 , etc.)

First we take P^2 and P^3 out of the composition loop by the transparency principle and Rule (CM.3). Contraction P^2 generates an assignment immediately outside the right bracket, but since there is no variable E_1 in its range (i.e. the remaining part of the walk until its end), we mentally commute it with every production, and eliminate it at the end of the walk.

Now the beginning of the walk is

$$(E_1 \rightarrow S'_2E_1)^2 (\backslash\backslash S_2 \rightarrow A)^3 (K_3 \leftarrow E_b)^1 [(K_3 \rightarrow S_2K_3)^4 \ldots$$

Taking out replacement P^4 and substituting it into P^1, we transform the latter into

$$(S_2K_3 \leftarrow E_b)^1$$

while P^4 disappears. Treating P^5 and P^6 as we treated P^2 and P^4, we make brackets empty and transform the first line representing the walk into:

$$(E_1 \to S_2'E_1)^2 \ (\backslash\backslash S_2 \to A)^3 \ (E_1 \to \square)^5 \ (S_2 \leftarrow E_b)^1$$

Here we can conjoin P^5 and P^2, which transforms P^2 into

$$(E_1 \to S_2)^2$$

and kills P^5.

Now we move P^1, which became a regular assignment, to the right. It commutes with P^7 and combines with P^8 into

$$(S_2 \to S_3'E_b)$$

by Rule (AC.D). The result of driving is two assignments:

$$(S_2 \leftarrow S_3)^{13} (\square \leftarrow E_b)^{14}$$

which are now to be moved to the right instead of one assignment P^1.

P^{14} combines with P^{11} into a trivial "contraction"

$$(\square \to \square)$$

and dies at the end of the walk.

By Rule (AR.D), P^{13} and P^9 combine into

$$(\backslash\backslash S_2 \to B)^{15} \ (S_2 \leftarrow S_3)^{13}$$

Restriction P^{15} moves to the left, while P^{13} transforms P^{10}:

$$(K_2 \to S_2K_2)^{10}$$

and dies too.

Now we have a sequence of replacements

$$(K_1 \to K_2)^7 \ (K_2 \to S_2K_2)^{10} \ (K_2 \to \square)^{12}$$

Combining them into one, we receive finally the normal form:

$$(E_1 \to S_2') \ (\backslash\backslash S_2 \to A)(\backslash\backslash S_2 \to B) \ (K_1 \to S_2)$$

Normalizing the walk in Example 3, we get:

$$(E_1 \to AAE_1) \ (\backslash\backslash E_1 \to AE_1) \ (K_1 \to BBE_1BE_1E_1E_1BE_1E_1)$$

5.3. Set Selectors.

A *set selector* is a function defined on a subset of the full set of expressions, and such that its value is identical to its argument when the argument is any expression for which the function is defined. So, both the domain and the range of a set selector are the same set of expressions, and on this set the function returns its argument as the value. We shall say that the selector *defines* this set.

Selector

$$\sigma\ L \Rightarrow L$$

where L is an L-expression, defines the same class as L. To define a set which is the union of several L-classes, we write the corresponding number of sentences; e.g., set selector

$$\tau\ s_1 \Rightarrow s_1$$

$$\tau(e_1) \Rightarrow (e_1)$$

defines the set of all *terms*.

A selector definition may contain "negative" sentences; e.g., the set of all symbols different from A may be defined by selector:

$$\alpha\ \ A \Rightarrow k?\perp$$

$$\alpha\ \ s_1 \Rightarrow s_1$$

Here the question mark "?" is the determiner of a function which is not defined (this simply means that no sentence in the algorighm has "?" as the determiner). Accordingly, when the argument of the selector function is A, the Refal machine will have an abnormal stop. With any other symbol, it will return the argument.

A selector definition may be recursive, such as

$$\nu \Rightarrow$$

$$\nu\ 1\ e_x \Rightarrow 1\ \nu\ e_x \perp$$

which defines the set of all sequences of 1's including the empty sequence. If we were to exclude the empty sequence, we would modify the definition in this way:

$$\nu\ 1 \Rightarrow 1$$
$$\nu\ 1\ e_x \Rightarrow 1\ \nu\ e_x \perp$$

To define more complicated set selectors one may introduce a hierarchy of auxiliary selectors, much in the same manner as when using the Backus normal form. It is especially easy to transform a BNF into an equivalent selector definition if the BNF meets certain requirements, which we will presently formulate.

To each alternative in a BNF we can relate a *covering* L-*class*. Let us take as an example the first line in the syntax of ALGOL-60 numbers as defined in the famous *Report on the Algorithmic Language ALGOL 60:*

<unsigned integer> ::= <digit>|<unsigned integer> <digit>

Since a digit is one of ten symbols 0123456789, we represent the first alternative by the L-expression

$$s(0123456789)_x$$

For the second alternative we construct, as a rough approximation, the L-class

$$e_x s(0123456789)_y$$

which is obtained by replacement of a metavariable whose values generally are expressions from the Refal viewpoint, by an e-variable. In the general case, for an alternative which corresponds to a set S of terminal expressions we can take as a *covering* L-*class* any L-class L such that $S \subseteq L$.

Suppose now that these rough approximations are sufficient to separate the alternatives for one metavariable in the BNF. More precisely, let it be possible to order the alternatives and the corresponding sets S_i in such a way that for each i:

$$S_i \cap L_1 = \emptyset$$
$$S_i \cap L_2 = \emptyset$$
$$\cdots\cdots$$
$$S_i \cap L_{i-1} = \emptyset$$

where L_j are covering classes for S_j.

Then we can define the set selector corresponding to the metavariable in question by making L_i the left side of the i-th sentence (i.e. parsing the argument as the covering L-class), and applying the corresponding set selectors to define the free variables in the right side. For unsigned integers this gives:

$$k/\text{UNSIGN-INT}/\ s(0123456789)_1 \Rightarrow s_1$$

$$k/\text{UNSIGN-INT}/\ e_1 s(0123456789)_2 \Rightarrow k/\text{UNSIGN-INT}/\ e_1 \perp s_2$$

Let us examine from this point of view the rest of the syntax of ALGOL-60 numbers:

```
<integer> ::= <unsigned integer>|+<unsigned integer>
                    |-<unsigned integer>
<decimal fraction>  ::=  .<unsigned integer>
<exponent part>     ::=  10 <integer>
<decimal number>    ::= <unsigned integer>|<decimal fraction>
                         |<unsigned integer> <decimal fraction>
<unsigned number>   ::= <decimal number>|<exponent part>
                         |<decimal number> <exponent part>
<number>            ::= <unsigned number>|+<unsigned number>
                         |-<unsigned number>
```

The definitions of integer, decimal fraction, exponent part and number meet our requirement, although those for integer and number need reordering. The definitions of decimal number and unsigned number do not satisfy the condition. But they can be rearranged to satisfy it.

What we need of course is unsigned number which is a string composed of three elements in fixed order: (1) unsigned integer or empty, (2) decimal fraction or empty; (3) exponent part of empty, with a restriction that all three cannot be empty. Therefore, there are three stages in the process of parsing, to which three recursive metavariable definitions must correspond: (1) when we are inside the unsigned integer, a decimal fraction and exponent part still may be encoutnered, (2) when we are inside the decimal fraction, an exponent part may be encountered, (3) when we are inside the exponent part, there will be no subsequent parts. We will assign to the first stage metavariable <ife>, and to the second stage metavariable <fe>. For the third stage we can use the already existing metavariable <integer>. This part of the BNF becomes:

$$
\begin{array}{lll}
\langle\text{unsigned number}\rangle & ::= & \langle\text{digit}\rangle\langle\text{ife}\rangle \mid .\langle\text{digit}\rangle\langle\text{fe}\rangle \mid {}_{10}\langle\text{integer}\rangle \\
\langle\text{ife}\rangle & ::= & \langle\text{digit}\rangle\langle\text{ife}\rangle \mid .\langle\text{digit}\rangle\langle\text{fe}\rangle \mid {}_{10}\langle\text{integer}\rangle \\
 & & \qquad\qquad \mid \langle\text{empty}\rangle \\
\langle\text{fe}\rangle & ::= & \langle\text{digit}\rangle\langle\text{fe}\rangle \mid {}_{10}\langle\text{integer}\rangle \mid \langle\text{empty}\rangle
\end{array}
$$

This BNF meets our criterion. Metavariables <decimal fraction> and <exponent part> become unnecessary, and the definition of the set selector /NUMBER/ will be:

$k/NUMBER/ + e_x \Rightarrow + k\ /UNSIGN\text{-}NUM/\ e_x \perp$

$k/NUMBER/ - e_x \Rightarrow - k\ /UNSIGN\text{-}NUM/\ e_x \perp$

$k/NUMBER/\ e_x \Rightarrow k\ /UNSIGN\text{-}NUM/\ e_x \perp$

$k/UNSIGN\text{-}NUM/\ s(0123456789)_1 e_2 \Rightarrow s_1\ k/IFE/e_2 \perp$

$k/UNSIGN\text{-}NUM/\ .s(0123456789)_1 e_2 \Rightarrow .s_1 k/FE/e_2 \perp$

$k/UNSIGN\text{-}NUM/\ {}_{10}e_1 \Rightarrow {}_{10}\ k/INTEGER/e_1 \perp$

$k/IFE/\ s(0123456789)_1 e_2 \Rightarrow s_1\ k\ /IFE/\ e_2 \perp$

$k/IFE/\ .s(0123456789)_1 e_2 \Rightarrow .s_1 k/FE/e_2 \perp$

$k/IFE/\ {}_{10}\ e_1 \Rightarrow {}_{10}\ k\ /INTEGER/\ e_1 \perp$

$k/IFE/ \Rightarrow$

$k/FE/\ s(0123456789)_1 e_2 \Rightarrow s_1 k/FE/e_2 \perp$

$k/FE/\ {}_{10}\ e_1 \Rightarrow {}_{10}\ k\ /INTEGER/\ e_1 \perp$

$k/FE/ \Rightarrow$

$k/INTEGER/ + e_1 \Rightarrow + k\ /UNSIGN\text{-}INT/\ e_1 \perp$

$k/INTEGER/ -e_1 \Rightarrow - k\ /UNSIGN\text{-}INT/\ e_1 \perp$

$k/INTEGER/\ e_1 \Rightarrow k\ /UNSIGN\text{-}INT/\ e_1 \perp$

It is easy to build the intersection of two sets defined through their set selectors. If selector σ^1 defines set S_1, and selector σ^2 defines set S_2, then the selector defining the intersection $S_1 \cap S_2$ is simply

$$\sigma\ e_x \Rightarrow \sigma^1\ \sigma^2\ e_x \perp \perp$$

But it is impossible to define the complement and the union of sets without reshaping the definitions of corresponding selectors. Very often it is more convenient to deal with classical recursive predicates as definitions of sets. Let a predicate P be given, which takes the truth value T if and only if the argument belongs to set S. A selector for S can be built without intervening into the definition of P. We define a universal function ε:

$$k\ \varepsilon\ s_p e_x \Rightarrow k\ \varepsilon^1 (ks_p\ e_x \perp)\ e_x \perp$$
$$k\ \varepsilon^1 (T) e_x \Rightarrow e_x$$

Now, function σ defined by

$$\sigma\ e_x \Rightarrow k\ \varepsilon\ P\ e_x \perp$$

will be a set selector for the set S defined by the classical predicate P.

Just as we use free variables to represent classes of expressions, we may use set selector calls to represent more sophisticated sets. The expression $\sigma\ e_x \perp$ stands for any e_x which is an element of the set defined by selector σ. We can limit the domain of a function by applying a set selector to the function's argument. Let ϕ be a function, and σ a selector defining set S. It may happen that we are interested in function ϕ as applied only to elements of S. Or we may know for sure that in some program the function ϕ will be applied only to such expressions which are again elements of S. Then we define a modified function

$$\phi^1 e_x \Rightarrow \phi\ \sigma\ e_x \perp \perp$$

and in the process of compilation, or making equivalence transformations in any other way, we use ϕ^1 instead of ϕ. It may lead to a very significant simplification of the resulting program.

As an example, let us consider the following theorem: if *X* is a string of 1's, then 1 *X* is identical to *X* 1. To formulate it in Refal we introduce the set selector

$$\sigma \Rightarrow$$
$$\sigma\ 1\ e_x \Rightarrow 1\ \sigma\ e_x \perp$$

and the predicate of identity:

$$k = (\ )\ (\ ) \Rightarrow T$$
$$k = (s_1 e_2)(s_1 e_3) \Rightarrow k(e_2)(e_3) \perp$$
$$k = e_x \Rightarrow F$$

Then we form the predicate which verifies the proposition stated in the theorem:

$$kC^1(e_x) \Rightarrow k{=}(1\ \sigma\ e_x\perp)(\sigma\ e_x\perp\ 1)\perp$$

The theorem will be proven if we transform this definition into

$$kC^1(e_x) \Rightarrow T$$

Let us use the compilation process. By the *outside-in* strategy, we find that the first subexpression to be driven is the second call of $\sigma\ e_x \perp$. The driving produces the following transitions and configurations:

(1 → 2) $e_x \rightarrow \square$

(2) $k = (1)\ (1) \perp$

(1 → 3) $e_x \rightarrow 1\ e_x$

(3) $k = (1\ 1\ \sigma\ e_x\perp)(1\ \sigma\ e_x \perp 1) \perp$

Configuration C^2 immediately becomes T ; configuration C^3 is also transitory and passes into C^1. Thus we obtain:

$$k\ C^1(\ ) \Rightarrow$$
$$kC^1(1e_x) \Rightarrow kC^1(e_x) \perp$$

This definition is transformed by induction (see Section 4.6) into

$$kC^1(e_x) \Rightarrow T$$

which proves the theorem.

Set selectors can also be used to define parametric sets, i.e. sets depending on a parameter. A parametric set selector has the format:

$$k\ F\ (P)\ E \perp$$

where F is a determiner, P a parameter, and E the argument proper. The value of this function when it exists will always be identical to E. For instance, this is the definition of the set of all strings built by the repetition of the same symbol (which is a parameter):

$$\begin{aligned} &\sigma(s_a) \Rightarrow \\ &\sigma(s_a)s_a e_x \Rightarrow s_a\sigma(s_a)e_x \perp \end{aligned}$$

One can see that the above theorem and its proof can be easily generalized for this set of strings.

5.4. Covering Context-Free Grammars.

The production systems we introduced in Section 5.2 are two-leveled. As they are defined (on the second metasystem level), they are context-free grammars producing certain objects. These objects (walks) are productions themselves, but of a different nature. They constitute the first metasystem level and produce input and output expressions on the zero level. First-level grammars are not context-free, because productions on this level appear in groups as the result of free choice on the second level, and once having appeared, they must be used in the process of walk interpretation. But we can build *covering* context-free grammars for input and output sets by "disassembling" productions of the first level. We will allow combining these productions freely. By doing so we certainly may only expand the set of produced expressions. These "covering" sets will give us first approximations to exact sets.

We shall discuss here only output sets. The output expression ultimately results from performing a series of replacements. So, as "the very first" approximation, we can collect all the replacements scattered throughout the right sides of the second-level production system, and ignore all the rest. Then we change nonterminal S- and E-symbols into s- and e-variables, and add restrictions if there are any. In Example 2 (Section 5.2) the resulting covering grammar will be:

$$K_1 \to K_2$$

$$K_2 \to \square$$

$$K_2 \to C\ K_2$$

$$K_2 \to s_3K_2 \ \backslash\backslash\ s_3 \to B$$

This is already something significant.

To construct a better grammar we should take into account that nonterminal symbols other than those corresponding to input variables may (and usually do) have more restricted ranges of values than the sets of all symbols and expressions. Therefore we add to our "dissassembled" system all the assignments for these nonterminals, which can be found in the original second-level system. In this process, we turn assignments into productions by swapping the sides and changing the direction of the arrow. This amounts in fact to the following way of interpreting a walk: first we move along the walk from left to right performing replacements; on coming to the end we reverse the direction of movement and apply assignments, which, being read from right to left, work now as contractions for nonterminals entering the result of conretization.

What should we do with contractions in this approach? We ignore them when moving from left to right since we are not interested in the input set. But when we are moving backwards we must remember that contractions not only limit the input set, but also very often *define* new variables. If we ignore it, we will not be able to ultimately reduce nonterminal symbols to terminal elements: symbols, parentheses, and free (input) variables.

Variables defined in the right side of a contraction are certain functions of the variable in the left side. If we introduce special functional designations for variables involved in elementary contractions, we will be able to replace a contraction for the left side variable by a set of assignments defining the right-side variables. At least this is true with respect to the four contractions which define new variable by breaking down an e-variable into two parts. The functional designations we choose are presented in the following table:

Contraction	Equivalent Assignments
$E_i \to S_j E_k$	$(\sigma^{\ell}(E_i) \leftarrow S_j)(\varepsilon^{r}(E_i) \leftarrow E_k)$
$E_i \to E_k S_j$	$(\sigma^{r}(E_i) \leftarrow S_j)(\varepsilon^{\ell}(E_i) \leftarrow E_k)$
$E_i \to (E_j) E_k$	$(\pi^{\ell}(E_i) \leftarrow E_j)(\varepsilon^{r}(E_i) \leftarrow E_k)$
$E_i \to E_k (E_j)$	$(\pi^{r}(E_i) \leftarrow E_j)(\varepsilon^{\ell}(E_i) \leftarrow E_k)$

All eight functions (which will be referred to as *split functions*) are defined so that if the required split of the argument is impossible then the function call has no value, and the expression in which this function call appears is declared to correspond to the empty set $\emptyset$ (not to the empty expression □!). A production containing such a function call in its right side should be cancelled.

So we introduce split function calls into the right sides of the productions in the covering context-free grammar. But there will be no split functions in any of the terminal expressions produced by the grammar. When all nonterminal symbols in the argument of a split function are replaced by terminal elements, its value can be easily *computed*. This is obvious when the argument consists of symbols and parentheses only, e.g.

$$\sigma^{\ell}(ABC) = A$$

$$\pi^{r}(X(YZ)) = YZ$$

etc. When there are free variables in the argument, one should remember that in a context-free grammar they represent corresponding *full sets* of values, and not some *unspecified* values, as is the case in driving. Also the value of a splitting function is *a set*, and no mapping is required between the argument set and the value set. Thus there will be no contractions. Not only

$$\sigma^{\ell}(e_1) = s_i$$

where i is any new index, but also

$$\sigma^{\ell}(e_1 s_2) = s_i$$

etc.

Using the definition of splitting functions, we can always transform the grammar so as to eliminate function calls from the final product. We are helped in this enterprise by the fact that we are after all building a *covering* grammar: if worse comes to the worst, we replace the unyielding function call by the appropriate full set s_x, or e_x.

Turning again to Example 2, we disassemble the W-production system into the following context-free grammar:

(1) $K_1 \to K_2$

(2) $K_2 \to C\ K_2$

(3) $K_2 \to S_3 K_2 \ \backslash\backslash\ S_3 \to B$

(4) $K_2 \to \square$

(5) $K_3 \to B\ K_3$

(6) $K_3 \to S_2 K_3 \ \backslash\backslash\ S_2 \to A$

(7) $K_3 \to \square$

(8) $E_b \to K_3$

(9) $E_b \to \varepsilon^{r}(E_b)$

(10) $S_3 \to \sigma^{\ell}(E_b)$

(11) $S_2 \to \sigma^{\ell}(e_1)$

The nonterminal symbol E_1 corresponds to an input variable and has been therefore changed into the free variable e_1. Production (11) is immediately transformed into

$$S_2 \rightarrow s_2$$

Now (6) becomes

$$K_3 \rightarrow s_2K_3 \backslash\backslash\ s_2 \rightarrow A$$

and (11) can be cancelled.

For S_3 we have only one production (10), which is nonrecurrent; thus we substitute it into (3) and cancel (10).

Now let us transform the productions for E_b. Nonrecurrent production (8) expresses E_b through K_3. Recurrent production (9) may be cancelled. Indeed any chain of applications of (9) will ultimately end with an application of (8). The combined result of the last two applications will be

$$E_b \rightarrow \varepsilon^r(K_3)$$

Unfolding K_3 , we use (5), (6) and (7) with the result

$$E_b \rightarrow K_3$$

$$E_b \rightarrow K_3$$

(The third production is annihilated, because $\varepsilon^r(\square)$ is $\emptyset$.) This shortens by one the length of the chain of the applications of (8). Since this procedure may be repeated any number of times, we come to the conclusion that the effect of the two productions (8) and (9) is exactly the same as that of one production (8).

Using (8) we transform (3) into

(3') $$K_2 \rightarrow \sigma^{\ell}(K_3)\ K_2\ \backslash\backslash \sigma^{\ell}(K_3) \rightarrow B$$

For the function call in (3') we obtain by unfolding K_3:

(12) $$\sigma^{\ell}(K_3) \rightarrow B$$

(13) $$\sigma^{\ell}(K_3) \rightarrow s_2 \backslash\backslash\ s_2 \rightarrow A$$

Combining (3') with (12) we get no production because of the restriction; combining it with (13) we get

(3") $\qquad K_2 \rightarrow s_2K_2 \backslash\backslash\ s_2 \rightarrow A \backslash\backslash\ s_2 \rightarrow B$

Production (2) can now be submerged by (3"), and we finally get:

$$\begin{array}{l}
K_1 \rightarrow K_2 \\
K_2 \rightarrow s_2K_2 \ \backslash\backslash\ s_2 \rightarrow A \ \backslash\backslash\ s_2 \rightarrow B \\
K_2 \rightarrow \square
\end{array}$$

The covering output set thus obtained is by the way equal to the exact output set.

To make covering grammars still more precise, we introluce *check functions* in addition to split functions. Check functions have the form $\zeta(C)$, where C is a contraction or restridtion for nonterminals, e.g., $\zeta(S_a \rightarrow B)$, $\zeta(\backslash\backslash S_1 \rightarrow B)$, $\zeta(E_x \rightarrow \square)$ etc. A check function has value □ if the condition specified by the argument is satisfied, otherwise it has no value and the expression in which the function call appears will correspond to the empty set ∅ of terminal expressions, as in the case of an unfeasible split.

We use check functions to take into account constrictions when disassembling a production system. If a path has a constriction C in it, we can add $\zeta(C)$ to the consequent of any replacement or assignment in this path. Consider the following example, which will be used in Section 5.5.

<u>Example 4</u>

$$\begin{array}{lll}
\psi\ A^4e_3 & \Rightarrow & B\ \beta\ \psi\ e_3 \perp \perp \\
\psi\ A^5e_3 & \Rightarrow & C\ \beta\ \psi\ e_3 \perp \perp \\
\psi\ P^6 & \Rightarrow & \\
\beta\ Ze_x & \Rightarrow & Z\ B\ \beta\ e_x \perp \\
\beta\ s_ae_x & \Rightarrow & s_a\beta\ e_x \perp \\
\beta & \Rightarrow &
\end{array}$$

The production system corresponding to this definition is:

$$W_1 \Rightarrow (E_3 \to A^4 E_3)(K_1 \to B\ K_2)(K_1 \leftarrow E_x)[W_1]W_2$$

$$W_1 \Rightarrow (E_3 \to A^5 E_3)(K_1 \to C\ K_2)(K_1 \leftarrow E_x)[W_1]W_2$$

$$W_1 \Rightarrow (E_3 \to P^6)(K_1 \to \Box)$$

$$W_2 \Rightarrow (E_x \to Z\ E_x)(K_2 \to Z\ B\ K_2)\ W_2$$

$$W_2 \Rightarrow (E_x \to S'_a E_x)(\backslash\backslash S_a \to Z)(K_2 \to S_a K_2)W_2$$

$$W_2 \Rightarrow (E_x \to \Box)(K_2 \to \Box)$$

Disassembling it into a context-free grammar for the output set, we obtain:

(1) $K_1 \to B\ K_2$

(2) $K_1 \to C\ K_2$

(3) $K_1 \to \Box$

(4) $K_2 \to Z\ B\ K_2\ \zeta(\sigma^{\ell}(E_x) \to Z)$

(5) $K_2 \to S_a K_2\ \zeta(\backslash\backslash S_a \to Z)$

(6) $K_2 \to \Box$

(7) $E_x \to K_1$

(8) $S_a \to \sigma^{\ell}(E_x)$

Substituting (7) into (4) and using (1), (2) and (3), we see that the check function in (4) never says yes, therefore we eliminate rule (4). From (8), (7) and the rules for K_1 again, we obtain:

$$S_a \to B$$
$$S_a \to C$$

Therefore we receive three productions for K_2 , whose right sides are completely identical to those for K_1. Identifying K_1 and K_2, we finally get:

$$K_1 \to B\ K_1$$

$$K_1 \to C\ K_1$$

$$K_1 \to \Box$$

In equivalence transformation, covering context-free grammars for output sets may be used in the following way. Suppose we have a composition, e.g.

$$kC^1(kC^2(e_1)(e_2) \perp)(e_3) \perp$$

Suppose we simplified the definition of C^2 as we could, and we did our best to drive the composition as a whole, yet could not avoid the necessity of decomposition. Then we may construct a covering context-free grammar for C^2 , and define the corresponding set selector $\sigma^2 e_x \perp$. Now intead of driving the outer configuration just as it is defined:

$$kC^1(e_x)(e_3) \perp$$

we will drive the configuration

$$kC^1(\sigma^2 e_x \perp)(e_3) \perp$$

exploiting in this way some properties of the argument e_x resulting from concretization of C^2. An example of using this technique will be given in Section 5.5.

5.5. Differential Metafunction.

Walk normalization defined in Section 5.2 applies to *terminal* walks, and produces walks in normal form. We introduce now the function τ, the *equivalence transformation* of a walk, which may be applied to both terminal and nonterminal walks. Function τ is such that if W_t is a terminal walk, then $\tau\ W_t \perp$ is a walk which has the same normal form as W_t. If W is a nonterminal walk, it represents a set S of terminal walks. Then for any $W_t \in S$, $\tau\ W_t \perp$ shall have the same normal form as W_t. Since normalization leaves the walk invariant with respect to interpretation, the equivalence transformation will also have this property. It will be presumed in the following that function τ is such that it makes the walk "closer", in some sense, to the normal form. In particular, for a terminal walk W_t the result of the concretization of $\tau\ W_t \perp$ should always be identical to the normal form of W_t (although the process of transformation should not necessarily be exactly the same as described in Section 5.2).

We presume furthermore that there is a definition of the function τ in Refal; however, we shall deal with it informally, appealing to our understanding of the normalization process and generalizing it for the case when there are some nonterminal W-symbols in the walk under transformation. There is only one difference to mention between the equivalence transformation we are going to use and the normalization process as described in Section 5.2. It concerns composition loops. We recall that a composition loop is a construction of the form

$$(1) \qquad (K_i \leftarrow E_x)[W_i]$$

where W_i is the nonterminal symbol for a walk with the head K-symbol K_i (indexes i and x are arbitrary). In the normalization process, replacements borne by W_i and coming up to the left bracket from inside are immediately used in the consequent of the assignment for the computed variable E_x. In the equivalence transformation we shall always keep the initial form of this assignment (changing, possibly, only the index i of the K-symbol). Instead, we shall add new assignments for E_x. Specifically, let the loop become:

$$(2) \qquad (K_i \leftarrow E_x)[(K_i \rightarrow C_1K_jC_2)\ W_j]$$

where C_1 and C_2 are multibrackets, and j may be equal to i or different from it. Instead of transforming it into

$$(3) \qquad (C_1K_jC_2 \leftarrow E_x)[W_j]$$

we shall reshape it in this manner:

$$(4) \qquad (K_j \leftarrow E_x)(C_1E_xC_2 \leftarrow E_x)[W_j]$$

which is equivalent to (3) provided that C_1 and C_2 do not contain E_x (if they do, a renaming procedure will be needed). Since after normalization W_j will not contain E_x , we transpose the second assignment and transform finally (4) into

$$(5) \qquad (K_j \leftarrow E_x)[W_j](C_1E_xC_2 \leftarrow E_x)$$

Let a function F be given by its graph of states G. A set W of all walks in G corresponds to F. A function δ^f will be referrned to as the *differential metafunction*, or *metaderivative* of F, if it is defined on the set W, and for each walk $W \in$ W, concretization $\delta^f W \perp$ gives the normal form of W. If σ^f is a set selector for W, then the metaderivative will be defined by the sentence:

$$\delta^f\ e_w \Rightarrow \tau\ \sigma^f\ e_w \perp \perp$$

Consider Example 1 from Section 5.2. Function F is:

$$\begin{aligned} &kF{+}e_1 && \Rightarrow && -kFe_1 \perp \\ &kFs_a e_1 && \Rightarrow && s_a kFe_1 \perp \\ &kF && \Rightarrow && \end{aligned}$$

Its graph of states is in Figure 17, and the production system defining set W is given by the rules (7), page 146. It is easy to turn this production system into a selector definition, because it is a context-free grammar, with the right sides which are nonoverlapping L-expressions with respect to nonterminal W-symbols (see Section5.3):

$$\sigma^f(E_1 \rightarrow +E_1)(K_1 \rightarrow -K_1)e_w \Rightarrow (E_1 \rightarrow +E_1)(K_1 \rightarrow -K_1)\ \sigma^f e_w \perp$$

$$\sigma^f(E_1 \rightarrow S'_a E_1)(\backslash\backslash S_a \rightarrow +)(K_1 \rightarrow S_a K_1)e_w \Rightarrow$$
$$(E_1 \rightarrow S'_a E_1)(\backslash\backslash S_a \rightarrow +)(K_1 \rightarrow S_a K_1)\ \sigma^f\ e_w \perp$$

$$\sigma^f(E_1 \rightarrow \square)(K_1 \rightarrow \square) \Rightarrow (E_1 \rightarrow \square)(K_1 \rightarrow \square)$$

This gives us immediately the metaderivative δ^f.

The argument of the metaderivative, as we have defined it, is an exact copy of a walk before normalization. It may be very long, and that is inconvenient, We now generalize the concept of metaderivative so that it may be defined on any set of expressions which are in one-to-one correspondence with walks — the codes of walks; as we shall see later, it is only the output set of the metaderivative that matters. Let us code the arcs of the graph of states which lead to active vertices by the letter A with superscripts, the arcs leading to passive vertices will be coded by P with superscripts. This gives us a convenient representation of walks which we shall use in the arguments of σ^f and δ^f. Let the arc in the first production be A^1, in the second A^2, and in the third P^3. Then function σ^f (which now is not, strictly speaking, a set selector, but rather a set *generator* for walks) becomes:

$$\sigma^f A^1 e_w \Rightarrow (E_1 \to +E_1)(K_1 \to -K_1)\sigma^f e_w \perp$$

$$\sigma^f A^2 e_w \Rightarrow (E_1 \to S'_a E_1)(\backslash\backslash S_a \to +)(K_1 \to S_a K_1)\ \sigma^f\ e_w \perp$$

$$\sigma^f P^1 \Rightarrow (E_1 \to \square)(K_1 \to \square)$$

A fundamental property of the equivalence transformation τ is that parts of the argument may be subject to a preliminary transformation without changing its overall result:

$$(6)\qquad \tau\ W_1 \perp \tau W_2 \perp W_3 \perp \ = \ \tau\ W_1\ W_2\ W_3 \perp$$

where W_1 , W_2 and W_3 are walks (terminal or nonterminal, or empty). Using the definition of δ^f, executing one step of driving, and using (6), we get a definition of δ^f, which does not depend on σ^f:

#1.1 $\qquad \delta^f A^1 e_w \Rightarrow \tau(E_1 \to +E_1)(K_1 \to -K_1)\ \delta^f\ e_w \perp\perp$

#1.2 $\qquad \delta^f\ A^2 e_w \Rightarrow \tau(E_1 \to S'_a E_1)(\backslash\backslash S_a \to +)(K_1 \to S_a K_1)\ \delta^f e_w \perp\perp$

#1.3 $\qquad \delta^f P^1 \Rightarrow (E_1 \to \square)(K_1 \to \square)$

Any expression from the output set (range) of function δ^f may be interpreted as a walk, and corresponds therefore to a one-step subfunction of the function F. Let us introduce a Refal function which will interpret walks. Its determiner will be <, and the format:

$$(7)\qquad k <W> E \perp$$

Here W is the walk being interpreted, E is the argument of the subfunction, and the sign > separates the walk from the argument. We define the function < so that the result of the concretization will be the value of the subfunction (i.e. the consequent of the final replacement, not the whole replacement), which is of course identical (when it exists) to the result of concretization of $kF\bar{E} \perp$.

We shall use $<\mathcal{W}>$ symbolically as the "determiner" of the subfunction it represents. The expression in the angular brackets may contain function calls. We may write e.g.

$$< \delta^f \ P^1 \perp >$$

This may be viewed as the determiner of a function defined by the sentence:

$$k<(E_1 \rightarrow \square)(K_1 \rightarrow \square)> \ (\ ) \ \Rightarrow$$

Or we may write

$$F^1 = < \delta^f \ A^2 \ A^1 \ P^1 \perp >$$

Then function F^1 will be defined by:

$$kF^1(+e_1) \ \Rightarrow \ K? \perp$$

$$kF^1(s_a \ +) \ \Rightarrow \ s_a-$$

By a *variable form* we mean a sequence (possibly empty) of terms $(\mathcal{V}_i)$, where $\mathcal{V}_i$ is an s- or e-variable, e.g.

$$(e_1)(s_a)(e_2)$$

For each point in a walk *the valid variable form* is defined which comprises all variables defined at that point. For each walk *the input variable form* is defined, which comprises all input variables and can be found as described in Section 5.2. Functions corresponding to walks (and later, to walk sets) will be defined in the format

$$k \ \mathcal{D} \ \textit{input-variable-form} \perp$$

where $\mathcal{D}$ is a regular determiner, or a "determiner" as in (7).

Since a walk is essentially a replacement for the head K-symbol, we can write symbolically:

(8) $\mathcal{W}_i = (K_i \rightarrow k<\mathcal{W}_i> \ \textit{input-variable-form} \ \perp)$

Indeed, walk W_i in its normal form consists of a conjunction of contractions and restrictions, which may define some new variables, followed by a replacement for the head symbol K_i, which may use new variables in the consequent. The right side of (8) is a replacement for the same nonterminal K_i , of which the consequent is doing with a function call the same job, as the normal form walk does with constrictions. In fact, equality (8) is a metasystem reduction rule. We make a metasystem transition when we consider the space of walks and functions defined in it, like τ and δ^f. To come back we express walks through functions in the original space of zero level expressions.

If a number of functions are defined on nonoverlapping sets, we call the *union* of these functions a function which is defined on the union of the domains, and takes a value using an appropriate function. The metaderivative function δ^f defines an infinite number of one-step functions

$$<\delta^f\ P^1 \perp >$$

$$<\delta^f\ A^1\ P^1 \perp >$$

$$<\delta^f\ A^2\ P^1 \perp >$$

$$<\delta^f\ A^1A^2P^1\perp >$$

etc., which have nonoverlapping domains. The original function F is the union of this infinite set of functions. Symbolically

$$F = \int_W < \delta^f\ W \perp >$$

which gave us reason to call δ^f the metaderivative. Function δ^f defines the breaking down of the original function F into a (generally infinite) set of elementary "differentials" — one-step subfunctions corresponding to different walks (paths taken by the computation process). The argument of the metaderivative function is a walk in some representation. The value is the corresponding one-step function. The full set of values — the range of the metaderivative is a definition of the original function as dissected into elementary pieces: one-step subfunctions.

The potentiality of taking metaderivative for equivalence transformations is that these pieces may be regrouped in different ways, bringing about different finite (recursive) definitions of the original function.

Let us generalize our symbolism by considering arbitrary sets of walks. If $\{W_i\}$ is *any* set of walks for the same head symbol K_i, then a function will correspond to it, which is the union of all functions corresponding to the elements of the set. Our notation for this function will be:

$$<\{W_i\}> = \int_{\{W_i\}} <W_i>$$

A set of walks may be represented with the help of free variables, e.g.

$$F = <\delta^f e_w \perp> = <\sigma^f e_w \perp>$$

Rule (8) is now generalized to

$$(9) \qquad \{W_i\} = (K_i \to k<\{W_i\}> \textit{input-variable-form} \perp)$$

In particular

$$(9') \qquad \delta^f e_w \perp = (K_i \to kF \textit{ input-variable-form} \perp)$$

Consider the transformation of the range of a metaderivative function into a definition of the original function in Example 1.

The metaderivative function δ^f is defined by three sentences #1.1 to 1.3. Nonterminal K_1 relates to the original function F. The first sentence contributes the set

$$\tau(E_1 \to +E_1)(K_1 \to -K_1)\delta^f e_w \perp \perp$$

to the range of the function δ^f. Using (9') and performing the equivalence transformation τ we get:

$$(10.1) \qquad (E_1 \to +E_1)(K_1 \to -kF(E_1) \perp)$$

The general rule of how to transform a walk into a sentence is: treat nonterminals S_i and E_i as corresponding free variables, and apply the walk to the "skeleton" of the sentence:

$$\# \; kF \textit{ input-variable-form} \Rightarrow K_1$$

Thus we receive:

$$kF(+e_1) \Rightarrow -kF(e_1) \perp$$

Treating #1.2 in the same way, we get a "normalized" nonterminal walk

(10.2) $(E_1 \to S'_aE_1)(\backslash\backslash S_a \to +)(K_1 \to S_a\ kF(E_1)\perp)$

It cannot be translated into a sentence because of the restriction, so let us regard it, for the time being, as another arc on the graph of states with independent arcs. The third arc will be simply

(10.3) $(E_1 \to \square)(K_1 \to \square)$

So, we have constructed a graph of states for the function F, which is no worse than a list of sentences. How to transform sets of independent arcs into sequences of sentences where restrictions are only implied is a different problem (which is neither difficult, nor terribly important). We might, e.g., decompose all contractions into elementary contractions, which would replace $(E_1 \to +E_1)$ by

$$(E_1 \to S'_aE_1)(S_a \to +);$$

then "take out of the brackets" the first contraction in (10.1) and (10.2), and then establish that the restriction $(\backslash\backslash S_a \to +)$ in (10.2) becomes unnecessary if the corresponding sentence is placed after (10.1), because (10.1) includes the contraction $(S_a \to +)$.

In this example we did not apply any equivalence transformations to the metaderivative function; we only came over to metaderivative, and then back to the original function, receiving the same (not counting small format differences) definition. But the reason for introducing a metaderivative is of course to subject it to transfomations and then return to the zero level with a new definition of the original function. When transforming metaderivative functions, we shall extensively use property (6) of the function τ, and its consequence:

$$\tau\ \delta^f\ e_w \perp \perp = \delta^f\ e_w \perp$$

These properties in fact make it unnecessary to keep in the record explicit invocations of the function τ. It is much more convenient to skip them and keep in mind that whenever there is a possibility to transform a walk by the function τ, this should be done immediately.

We also introduce another technical device. It will often facilitate equivalence transformations of walks if in the representation of the walk we keep constrictions separated from the rest of the walk. We shall achieve this by using the following format for walks:

C(*constrictions*) *the rest of the walk*

This representation will be used in parallel with the unformatted representation, so as not to encumber the record with the format when it is not needed. Even without any comments, we cannot confuse one representation for another, because a formatted walk always begins with the letter C, while an unformatted walk begins with a parenthesis.

Consider Example 2 from Section 5.2. The walk space is defined by the production system (8). Our first task is to code the arcs of the graph of states. In addition to horizontal arcs leading to an active or passive vertex, which we agreed to code by A^i and P^i respectively, we see here a composition loop, reflected by the presence of square brackets in (8.1). We shall encode a combination of a horizontal arc and a composition rule by a superscripted C followed by a pair of parentheses enclosing the code of the inner walk in the loop.

This means that to formulate such a convention we need a free variable representing the inner walk. In our case:

$$C^1(e_w) \;::=\; (K_3 \leftarrow E_b)[e_w](K_1 \rightarrow K_2)$$

The rest of the conventions will be:

$$A^1 \quad ::= (E_b \rightarrow BE_b)(K_2 \rightarrow CK_2)$$

$$A^2 \quad ::= (E_b \rightarrow S'_3E_b)(\backslash\backslash S_3 \rightarrow B)(K_2 \rightarrow S_3K_2)$$

$$P^3 \quad ::= (E_b \rightarrow \square)(K_2 \rightarrow \square)$$

$A^4 \quad ::= (E_1 \to AE_1)(K_3 \to BK_3)$

$A^5 \quad ::= (E_1 \to S_2'E_1)(\backslash\backslash S_2 \to A)(K_3 \to S_2K_3)$

$P^6 \quad ::= (E_1 \to \square)(K_3 \to \square)$

To illustrate these conventions, the code

$$C^1(P^6)P^3$$

means

$$(K_3 \leftarrow E_b)[(E_1 \to \square)(K_3 \to \square)](K_1 \to K_2)(E_b \to \square)(K_2 \to \square)$$

To each of the nonterminals W_i , $i = 1,2,3$, a metaderivative function δ^i will correspond. We are of course interested in δ^1 which is the metaderivative of the original function α. Functions δ^1 and δ^2 are metaderivatives of the respective auxiliary functions F^b and F^a. Transforming production rules into sentences defining metaderivatives, and ignoring function τ as explained above, we get the following definitions:

#2.1 $\delta^1 C^1(e_3)e_2 \Rightarrow (K_3 \leftarrow E_b)[\delta^3 e_3 \perp](K_1 \to K_2)\delta^2 e_2 \perp$

#2.2 $\delta^2 A^1 e_2 \Rightarrow (E_b \to BE_b)(K_2 \to CK_2)\delta^2 e_2 \perp$

#2.3 $\delta^2 A^2 e_2 \Rightarrow (E_b \to S_3'E_b)(\backslash\backslash S_3 \to B)(K_2 \to S_3K_2)\delta^2 e_2 \perp$

#2.4 $\delta^2 P^3 \Rightarrow (E_b \to \square)(K_2 \to \square)$

#2.5 $\delta^3 A^4 e_3 \Rightarrow (E_1 \to AE_1)(K_3 \to BK_3)\delta^3 e_3 \perp$

#2.6 $\delta^3 A^5 e_3 \Rightarrow (E_1 \to S_2'E_1)(\backslash\backslash S_2 \to A)(K_3 \to S_2K_3)\delta^3 e_3 \perp$

#2.7 $\delta^3 P^6 \Rightarrow (E_1 \to \square)(K_3 \to \square)$

To transform function δ^1 , we use the compilation process augmented by the rules of the equivalence transformation τ in the walk space. Our goal is to obtain a definition, which (1) would be interpretable on the zero level as a recursive definition of the original function α, and (2) would correspond to a more perfect graph of states than the original definition of α. The first objective will be attained if the configuration being interpreted is of the form

(11) $\mathcal{L}(K_1 \to C_\ell \; K_j \; C_r) \; \{\mathcal{W}_j\}$

where L is a list of contractions and restrictions, C_ℓ and C_r are multibrackets, $\{W_j\}$ is any nonterminal walk for the head symbol K_j , and j may be either 1 or different from 1. Using rule (9) we interpret such a configuration as the arc

$$L(K_1 \to C_\ell\, k<\{W_j\}> \textit{input-variable-form} \perp C_r)$$

Therefore, to receive a self-sufficient recursive definition, we shall try to find in the process of compilation such configurations of the form (11), where $\{W_j\}$ is a configuration already met previously.

The second objective is attained almost automatically, because as we drive the metaderivative function we consider longer and longer walks; those which are unfeasible will be turned into □ by function τ.

Our initial configuration is $\delta^1 e_1 \perp$. Using #2.1 we change it into

$$(12) \qquad (K_3 \leftarrow E_b)[\delta^3 e_3 \perp](K_1 \to K_2)\ \delta^2 e_2 \perp$$

(Contraction $(e_1 \to C^1(e_3)e_2)$ has been made.)

After each step of driving we use the equivalence transformation in the walk space, driving the invisible function τ. Whenever it is allowed by the rules of the equivalence transformation to move a substitution in the desirable direction (i.e. contractions, restrictions, and replacements to the left, and assignments to the right), function τ will do it. In (12) we have only one movable substitution — replacement $(K_1 \to K_2)$ (assignment $(K_3 \leftarrow E_b)$ is part of a composition loop). To see whether or not we can commute a substitution with a functional loop, we mentally substitute for the bracketed nonterminal walk its equivalent according to rule (9), in this case:

$$\delta^3 e_3 \perp = (K_3 \to k <\delta^3 e_3 \perp > (E_1) \perp)$$

We see that the loop does not contain nonterminals K_1 or K_2 , therefore we commute our replacement with it. Configuration (12) is transitory and passes into

$$(13) \qquad (K_1 \to K_2)(K_3 \leftarrow E_b)[\delta^3 e_3 \perp]\ \delta^2\ e_2 \perp$$

Now we drive subexpression $\delta^3 e_3 \perp$ in (13). Three continuations are possible, according to three sentences #2.5 to 2.7. Let us examine the arc with the contraction $(e_3 \to A^4 e_3)$, #2.5:

(14) $(K_1 \to K_2)(K_3 \leftarrow E_b)[(E_1 \to AE_1)(K_3 \to BK_3)\delta^3 e_3 \perp]\ \delta^2 e_2 \perp$

A straightforward transformation of (14) by the rules produces:

(15) $(E_1 \to AE_1)(K_1 \to K_2)(K_3 \leftarrow E_b)[\delta^3 e_3 \perp](BE_b \leftarrow E_b)(AE_1 \leftarrow E_1)\delta^2 e_2 \perp$

The result of concretizing $\delta^2 e_2 \perp$ does not include E_1 , as one can establish by analyzing #2.2 to 2.4. Therefore, we commute assignment $(AE_1 \leftarrow E_1)$ with $\delta^2 e_2 \perp$ and eliminate it. But we cannot move assignment $(BE_b \leftarrow E_b)$ further to the right, because $\delta^2 e_2 \perp$ is not independent of E_b. According to the strategy *inside from outside*, function τ will demand that $\delta^2 e_2 \perp$ be driven. Again, three continuations are possible. The one corresponding to #2.2 and contraction $(e_2 \to A^1 e_2)$ produces:

(16) $(E_1 \to AE_1)(K_1 \to K_2)(K_3 \leftarrow E_b)[\delta^3 e_3 \perp](BE_b \leftarrow E_b)(E_b \to BE_b)(K_2 \to CK_2)\delta^2 e_2 \perp$

The assignment and the contraction for E_b are annihilated, the replacement $(K_2 \to CK_2)$ drifts left and gets used in the replacement $(K_1 \to K_2)$, the result being

(17) $(E_1 \to AE_1)(K_1 \to CK_2)(K_3 \leftarrow E_b)[\delta^3 e_3 \perp]\ \delta^2 e_2 \perp$

Two compilation strategies may be formulated, which will lead to a successful transformation. The more straightforward one is just L-generalization, as described in Section 4.6. Comparing (17) with (13), we find an L-generalization

(18) $$e_x(K_3 \leftarrow E_b)[\delta^3 e_3 \perp]\ \delta^2 e_2 \perp$$

Repeating the compilation process for the new initial configuration (18) we find out that the specific part of (18), which is common for (13) and (17):

(19) $$(K_3 \leftarrow E_b)[\delta^3 e_3 \perp]\ \delta^2 e_2 \perp$$

is recurrent and self-sufficient. In the final result, the original function α will be found identical to the function corresponding to the walk set (19), for which we will have received a recursive definition. One can see that the necessity of introducing an auxiliary function (which is in fact identical to the original function α) in this approach reflects the fact that it is the configuration $kF^bkF^ae_1\perp\ \perp$, corresponding to the walk set (19), that is recurrent, and not function α corresponding to the walk set (13).

The more sophisticated strategy is to try to leave on the right, when moving contractions and replacements to the left, as many of the original substitutions *unchanged* as possible. The purpose is to make $\{\mathcal{W}_j\}$ in the form (11) as close to the original function as possible. Thus instead of combining the two replacements:

$$(K_1 \rightarrow K_2)(K_2 \rightarrow CK_2)$$

we shall *commute* them, transforming this composition into the composition:

$$(K_1 \rightarrow CK_1)(K_1 \rightarrow K_2)$$

which is equivalent to it. (It is not difficult to formulate algorithmically this transformation.)

Now the whole configuration (13) becomes recurrent, Gathering all the contractions we have made, we obtain the following sentence for function δ^1 :

#3.1 $\delta^1C^1(A^4e_3)A^1e_2 \Rightarrow (E_1 \rightarrow AE_1)(K_1 \rightarrow CK_1)\delta^1C^1(e_3)e_2\perp$

There are three possible contractions for e_3 and three possible contractions for e_2 , so we receive eight more sentences in the same manner as we received #3.1. Some of them will have □ in the right side. E.g., if on receiving configuration (15) we were to choose contraction $(e_2 \rightarrow A^2e_2)$ suggested by #2.3, we would receive a walk containing the following composition:

$$(BE_b \leftarrow E_b)(E_b \rightarrow S_3'E_b)(\backslash\backslash S_3 \rightarrow B)$$

Transforming the first pair by Rule (AC.D), Section 5.2, we have:

$$(B \leftarrow S_3)(\backslash\backslash S_3 \rightarrow B)$$

which is unfeasible, so that function τ turns the whole walk into the empty expression.

The full definition of function δ^1 resulting from the compilation process will be:

#3.1 $\delta^1 C^1(A^4 e_3)A^1 e_2 \Rightarrow (E_1 \to AE_1)(K_1 \to CK_1)\ \delta^1 C^1(e_3)e_2 \perp$

#3.2 $\delta^1 C^1(A^4 e_3)A^2 e_2 \Rightarrow$

#3.3 $\delta^1 C^1(A^4 e_3)P^3 \Rightarrow$

#3.4 $\delta^1 C^1(A^5 e_3)A^1 e_2 \Rightarrow (E_1 \to BE_1)(K_1 \to CK_1)\ \delta^1 C^1(e_3)e_2 \perp$

#3.5 $\delta^1 C^1(A^5 e_3)A^2 e_2 \Rightarrow (E_1 \to S_2^{\prime} E_1)(\backslash\backslash S_2 \to S(AB)_2)$
$(K_1 \to S_2 K_1)\ \delta^1 C^1(e_3)e_2)\ \perp$

#3.6 $\delta^1 C^1(A^5 e_3)P^3 \Rightarrow$

#3.7 $\delta^1 C^1(P^6)A^1 e_2 \Rightarrow$

#3.8 $\delta^1 C^1(P^6)A^2 e_2 \Rightarrow$

#3.9 $\delta^1 C^1(P^6)P^3 \Rightarrow (E_1 \to \square)(K_1 \to \square)$

Transforming the range of this function into a function definition, we receive four sentences for the original function α:

#4.1 $\alpha(A\ e_1) \Rightarrow C\ \ \alpha(e_1) \perp$

#4.2 $\alpha(B\ e_1) \Rightarrow C\ \ \alpha(e_1) \perp$

#4.3 $\alpha(s_2 e_1) \Rightarrow s_2\ \alpha(e_1) \perp$

#4.4 $\alpha(\ ) \Rightarrow$

In the transformed definition of the metaderivative #3, it is very clearly seen what the transformation is: a *regrouping* of walks in the full set of walks. There are nine classes of walks, of which five classes are found to include unfeasible walks only, so that they may be discarded. In the remaining four classes, all walks are feasible. And there are some equivalence relations established between walks inside the classes.

One might say: so what? The transformation we have performed was also performed in Section 4.5 without introducing a metaderivative, just by the compilation process.

It is true. But let us consider the following definition.

Example 5.

#5.1 $\alpha\ e_1 \Rightarrow kF^b\ kF^a\ (\ )\ e_1 \perp \perp$

#5.2 $kF^a(e_1)Ae_2 \Rightarrow kF^a(e_1B)e_2 \perp$

#5.3 $kF^a(e_1)s_xe_2 \Rightarrow kF^a(e_1s_x)e_2 \perp$

#5.4 $kF^a(e_1) \Rightarrow e_1$

#5.5 $kF^bBe_1 \Rightarrow C\ kF^be_1 \perp$

#5.6 $kF^bs_xe_1 \Rightarrow s_x\ kF^b\ e_1 \perp$

#5.7 $kF^b \Rightarrow$

Let us first try to transform function α by driving. In configuration

(20) $kF^b\ kF^a\ (\ )\ e_1 \perp \perp$

the strategy *from without within* demands that we drive the F^a call. The result will be three configurations, of which the first is

(21) $kF^b\ kF^a\ (B)\ e_1 \perp \perp$

In this configuration the F^a call is to be driven again. One can see that no matter how many steps of driving an F^a call we perform, there will be configurations like (20) and (21) in the graph of states (and in fact their number will grow), which do not allow driving the F^b call. Therefore, acting on any compilation strategy we shall have to decompose (20) or its successors, so that the new graph of states will be no more efficient than the original.

At the same time we notice that function α is equivalent to function α from Example 2. The difference in its definition is only that one of the auxiliary functions, F^a, is defined not by *recursion* — in the sense this term is used in programming — but by *iteration*. The simple method working in Example 2 does not work in this case.

Let us compute the definition of the metaderivative function δ^1:

#6.1	$\delta^1 C^1(e_3)e_2$	$\Rightarrow$	$(\Box \leftarrow E_2)(K_3 \leftarrow E_b)[\delta^3 e_3\bot](K_1 \rightarrow K_2)\delta^2 e_2 \bot$
#6.2	$\delta^2 A^1 e_2$	$\Rightarrow$	$(E_b \rightarrow BE_b)(K_2 \rightarrow CK_2)\ \delta^2 e_2\bot$
#6.3	$\delta^2 A^2 e_2$	$\Rightarrow$	$(E_b \rightarrow S_3' E_b)(\backslash\backslash S_3 \rightarrow B)(K_2 \rightarrow S_3 K_2)\delta^2 e_2 \bot$
#6.4	$\delta^2 P^3$	$\Rightarrow$	$(E_b \rightarrow \Box)(K_2 \rightarrow \Box)$
#6.5	$\delta^3 A^4 e_3$	$\Rightarrow$	$(E_1 \rightarrow AE_1)(E_2 B \leftarrow E_2)\ \delta^3 e_3 \bot$
#6.6	$\delta^3 A^5 e_3$	$\Rightarrow$	$(E_1 \rightarrow S_2' E_1)(\backslash\backslash S_2 \rightarrow A)(E_2 S_2 \leftarrow E_2)\ \delta^3 e_3 \bot$
#6.7	$\delta^3 P^6$	$\Rightarrow$	$(E_1 \rightarrow \Box)(K_3 \rightarrow E_2)$

We start to transform δ^1 the same way we did in Example 2. Driving $\delta^3 e_3\bot$ for the case $(e_3 \rightarrow A^4 e_3)$, #6.5, we again move out of the brackets the contraction $(E_1 \rightarrow AE_1)$, but instead of the replacement we had in Example 2, we now have an assignment, which can be neither taken out, nor commuted with $\delta^3 e_3\bot$:

$$[(E_2 B \leftarrow E_2)\ \delta^3 e_3 \bot\ ]$$

Therefore, the implied function τ will demand a driving of $\delta^3 e_3\bot$. This demand would be repeated infinitely should we comply with it, which means that configuration $\delta^3 e_3\bot$ must be separated by decomposition — just as in the case of the original function.

So, let us separate function δ^3 and try to transform it into something more manageable. Compute δ^3 for several simple arguments with the view of using the L-generalization technique:

(P^6)	$C((E_1 \rightarrow \Box))(K_3 \rightarrow E_2)$
$(A^4 P^6)$	$C((E_1 \rightarrow A))(K_3 \rightarrow E_2 B)$
$(A^5 P^6)$	$C((E_1 \rightarrow S_2')(\backslash\backslash S_2 \rightarrow A))(K_3 \rightarrow E_2 S_2)$
$(A^4 A^4 P^6)$	$C((E_1 \rightarrow AA))(K_3 \rightarrow E_2 BB)$

The algorithm of L-generalization gives a generalizing configuration

$$C(e_c)(K_3 \rightarrow E_2 e_x)$$

The induction hypothesis is:

$$\delta^3 e_3 \bot = C(\psi^c\ e_3 \bot\ )\ (K_3 \rightarrow E_2\ \psi^x\ e_3 \bot\ ) \qquad (22)$$

Substituting this into #6.5 in accordance with the general algorithm described in Section 4.6, we have:

$$\#6.5' \quad k\ C(\psi^{c}\ A^{4}e_{3}\bot)\ (K_{3} \rightarrow E_{2}\ \psi^{x}\ A^{4}e_{3}\ \bot\ ) \ \Rightarrow$$

$$C((E_{1} \rightarrow AE_{1}))(E_{2}B \leftarrow E_{2})\ C(\psi^{c}e_{3}\bot)(K_{3} \rightarrow E_{2}\psi^{x}\ e_{3}\ \bot\ )$$

To transform the walk in the right side, we have to transpose the assignment for E_2: first with $C(\psi^{c}e_{3}\ \bot\ )$, and then with the remaining replacement. The first transposition will not change any of the parties, because the constrictions depend only on E_1 , not on E_2. We leave it unformalized, in order not to be buried in details. A way to formalize (by which we always mean to perform *algorithmically*) this and like transformations is to make the format of a walk still a bit more sophisticated by including into it the list of nonterminals on which the constrictions depend, e.g.

C((*nonterminals*) *constrictions*) *the rest of the walk*

Then in the process of L-generalization we would receive a generalized form:

$$C(\ (E_{1})\ e_{c})$$

which will enable function τ to transpose it with the assignment.

Making the other transposition by rule (ARP), Section 5.2, we change the replacement into:

$$(K_{3} \rightarrow E_{2}B\ \beta\ \psi^{x}\ e_{3}\ \bot\ \bot\ )$$

where the function β is performing the substitution $//E_{2}B \leftarrow E_{2}$:

$$\beta\ E_{2}\ e_{x} \Rightarrow E_{2}\ B\ \beta\ e_{x}\ \bot$$

$$\beta\ s_{a}\ e_{x} \Rightarrow s_{a}\ \beta\ e_{x}\ \bot$$

$$\beta \qquad \Rightarrow$$

(In fact, $\psi^{3}e_{3}\bot$ does not include symbols E_2 , but the algorithm does not yet know it.)

After merging the constrictions, which have now become adjacent, we obtain from #6.5' the recursive relations:

$$\psi^{c}\ A^{4}\ e_{3} \Rightarrow (E_{1} \rightarrow AE_{1})\ \psi^{c}\ e_{3}\ \bot$$

$$\psi^{x}\ A^{4}\ e_{3} \Rightarrow B\ \beta\ \psi^{x}\ e_{3}\ \bot$$

Processing #6.6 and #6.7 analogously, we receive the following complete definitions of functions ψ^c and ψ^x:

#7.1 $\psi^c A^4 e_3 \Rightarrow (E_1 \to AE_1)\ \psi^c\ e_3 \perp$

#7.2 $\psi^c A^5 e_3 \Rightarrow (E_1 \to S_2' E_1)(\backslash\backslash S_2 \to A)\ \psi^c\ e_3 \perp$

#7.3 $\psi^c P^6 \Rightarrow (E_1 \to \square)$

#8.1 $\psi^x A^4 e_3 \Rightarrow B\ \beta\ \psi^x\ e_3 \perp \perp$

#8.2 $\psi^x A^5 e_3 \Rightarrow S_2\ \beta\ \psi^x\ e_3 \perp \perp$

#8.3 $\psi^x P^6 \Rightarrow$

We cannot simplify the definition of ψ^x by driving, so we try the techniques described in Section 5.4, which makes use of covering context-free grammars. In Example 4, Section 5.4, we found that the covering grammar for ψ^x will be:

$$K_1^2 \to B\ K_1^2$$
$$K_1^2 \to S_2 K_1^2$$
$$K_1^2 \to \square$$

(In Section 5.4 we used symbol C instead of S_2 to avoid confusion; here we use K-symbols of the second metasystem level.)

This corresponds to the set selector:

$$\sigma\ B\ e_x \Rightarrow B\ \sigma\ e_x \perp$$
$$\sigma\ S_2 e_x \Rightarrow S_2\ \sigma\ e_x \perp$$
$$\sigma \Rightarrow$$

We change $\psi^x e_3 \perp$ in the right sides of the sentences into $\sigma\ \psi^x\ e_3 \perp \perp$. Now, composition $\beta\sigma e_x \perp \perp$ may be transformed by driving and L-generalization into an identity function, so that the definition of ψ^x becomes:

#8.1' $\psi^x A^4 e_3 \Rightarrow B\ \psi^x\ e_3 \perp$

#8.2' $\psi^x A^5 e_3 \Rightarrow S_2\ \psi^x\ e_3 \perp$

#8.3' $\psi^x P^6 \Rightarrow$

and (22) becomes the new definition of δ^3:

#6.X $\delta^3 e_3 \Rightarrow C(\psi^c\ e_3 \perp)\ (K_3 \to E_2\ \ \psi^x\ e_x \perp)$

Now we once more try to transform function δ^1 by driving. The initial configuration is $\delta^1\ e_1 \perp$. Using #6.1 and #6.X, we turn it into

(23) $\psi^c\ e_3 \perp (K_1 \to K_2)(\psi^x\ e_3 \perp \leftarrow E_b)\ \delta^2\ e_2 \perp$

(It was taken into account here, that $\psi^c\ e_3 \perp$ depends only on E_1 and its offshoots, and $\delta^2 e_2 \perp$ depends on E_b and its offshoots.)

We drive now configuration (23) as we did configuration (13) in Example 2. The effect of combined contractions $(e_3 \to A^4 e_3)$ (see #7.1 and #8.1') and $(e_2 \to A^1 e_2)$ (see #6.2) is:

(24) $(E_1 \to AE_1)\psi^c e_3 \perp\ (K_1 \to K_2)(B\psi^x e_3 \perp \leftarrow E_b)(E_b \to BE_b)(K_2 \to CK_2)\delta^2 e_2 \perp$

The clash of an assignment and a contraction for E_b produces the assignment

$$(\psi^x\ e_3 \perp \leftarrow E_b)\ .$$

Taking the replacement $(K_2 \to CK_2)$ to the left as far as possible and keeping old substitutions unchanged, the same way as we did in Example 2, we obtain the recursive relation

$$\delta^1 C^1 (A^4 e_3) A^1 e_2 \Rightarrow (E_1 \to AE_1)(K_1 \to CK_1)\delta^1 C^1 (e_3)\ e_2 \perp$$

which is exactly the same as #3.1 in Example 2. Proceeding in this manner, we reproduce the full text of definitions #3.1-3.9. Returning to the object space, we obtain the efficient definition #4 for α.

Taking the metaderivative of a function of several variables, we can treat some of the variables as parameters; they will remain free variables, while all of the other variables will be transformed into nonterminals. Thus different variables find themselves assigned to different metasystem levels. We refer to this procedure as a *metasystem split of variables*. Its importance for the equivalence transformations will be demonstrated in the next section in the context of taking the *metaintegral*. With respect to the metaderivative, the notion of metasystem split leads to partial differentiation. We use notation

$$k\ \partial(e_w)\ F(E_1)\ldots(E_m)(e_{m+1})\ldots(e_n) \perp$$

to represent the most general set selector for partial metaderivatives. Here e_w is the free variable which has walks as its values. Function F depends on n variables $e_1, e_2, \ldots, e_m, e_{m+1}, \ldots, e_n$. We treat the last n-m of these as parameters, thereby defining a function of m variables $e_1, \ldots, e_m$. It is the walks in the graph of states of this function that are values of e_w. We do not include the equivalence transformation into the definition of the partial metaderivative. Thus the metaderivative function δ^f which was used above may be defined using this notation as

$$\delta^f\ e_3 \Rightarrow \tau k\ \partial(e_3)\ F(E_1) \perp \perp$$

5.6. Integral Metafunction.

Let us try to prove the commutativity of addition by equivalent transformation of the corresponding recursive predicate F:

Example 6.

#9 $\quad kF(e_1)(e_2) \Rightarrow k{=}(k{+}(e_1)(e_2)\perp)(k{+}(e_2)(e_1)\perp)\perp$

#10.1 $\quad k{+}(e_1)(0) \Rightarrow e_1$

#10.2 $\quad k{+}(e_1)(e_2 1) \Rightarrow k{+}(e_1)(e_2) \perp 1$

#11.1 $\quad k{=}(0)(0) \Rightarrow T$

#11.2 $\quad k{=}(e_1 1)(e_2 1) \Rightarrow k{=}(e_1)(e_2) \perp$

#11.3 $\quad k{=}e_x \Rightarrow F$

Driving the right side of #9 by the inside from outside strategy, we come to the necessity of a split according to #10:

(c^1) $\quad k = (k + (e_1)(e_2) \perp)\ (k + (e_2)(e_1) \perp) \perp$

$(1 \to 2)$ $\quad (e_2 \to 0)(K_1 \to K_2)$

$(1 \to 3)$ $\quad (e_2 \to e_2 1)(K_1 \to K_3)$

(c^2) $\quad k = (e_1)(k + (0)(e_1) \perp) \perp$

(c^3) $\quad k = (k + (e_1)(e_2) \perp 1)(k + (e_2 1)(e_1) \perp) \perp$

Configuration C^2 is easily transformed by driving and induction:

$(2 \to \square)$ $\qquad$ $(e_1 \to 0)(K_2 \to T)$

$(2 \to 4)$ $\qquad$ $(e_1 \to e_1 1)(K_2 \to K_4)$

(C^4) $\qquad$ $k = (e_1 1)(k + (0)(e_1) \perp 1) \perp$

$(4 \to 2)$ $\qquad$ $(K_4 \to K_2)$

Translating this graph of states into a Refal program, we have:

$$kC^2(0) \Rightarrow T$$

$$kC^2(e_1 1) \Rightarrow kC^2(e_1) \perp$$

which by L-generalization and induction is transformed into

$$kC^2(e_1) \Rightarrow T$$

Configuration C^3, however, does not yield itself to a transformation which would make some later stage equal to a previous stage. Driving C^3 according to the inside from outside strategy, we make a contraction for e_1 resulting from #10:

$(3 \to 5)$ $\qquad$ $(e_1 \to 0)(K_3 \to K_5)$

$(3 \to 6)$ $\qquad$ $(e_1 \to e_1 1)(K_3 \to K_6)$

(C^5) $\qquad$ $k = (k + (0)(e_2) \perp)(e_2) \perp$

(C^6) $\qquad$ $k = (k + (e_1 1)(e_2) \perp)(k + (e_2 1)(e_1) \perp) \perp$

Configuration C^5 is analogous to C^2 and can be as easily transformed into T, but a further driving of C^6 will only lead to accumulation of ones. Attempting an L-generalization will lead to an extremely general configuration

$$k = (k + (e_1)(e_2)\perp)(k + (e_3)(e_4) \perp) \perp$$

which cannot be shown to be always T because it is not.

We may try a more sophisticated technique of generalization. Transform configurations C^1 and C^6 by metacode. They become object expressions:

(C^{1*}) $\quad *K(=(*K(+(E_1)(E_2)))(*K(+(E_2)(E_1))))$

(C^{6*}) $\quad *K(=(*K(+(E_1 1)(E_2)))(*K(+(E_2 1)(E_1))))$

Making an LE-generalization *now*, we get this class on the first metasystem level:

(C^{x*}) $\quad *K(=(*K(+(E_1 e_x)(E_2)))(*K(+(E_2 e_x)(E_1))))$

By metasystem reduction it corresponds to the class

(C^x) $\quad k = (k + (e_1 e_x)(e_2) \perp)(k + (e_2 e_x)(e_1) \perp) \perp$

on the ground level, which is quite a clever generalization; its concretization can give only T. Unfortunately, this does not bring us closer to the solution of the problem. We express C^x through itself by simultaneous recursion by e_1 and e_2 , but with $e_2 \rightarrow 0$ we receive a configuration

$$k = (e_1 e_x)(k + (0 e_x)(e_1) \perp) \perp$$

which again expresses the commutativity of addition, and is no easier to transform than C^1.

In search of a solution, let us compare our approach with the approach of axiomatic formal arithmetic. What we express by adding digit 1 on the right to a number is usually expressed in formal arithmetic by adding a prime '. Variables are represented by small letters. The axioms for addition and the basic axioms for equality are close analogues to our recursive definitions. Besides, there is an additional axiom for equality (transitivity), and the axiom of induction. The syntax of formal arithmetic leads to more compact expressions than in Refal, which is, of course, a consequence of its narrow specialization.

We notice first of all that our transformation of configuration C^2 is a proof of the theorem:

(T1) $\quad 0 + x = x$

We notice also that this theorem was never set as a subgoal: it just appeared as a by-product when we were applying our general algorithm of equivalence transformation. This exemplifies the fundamental distinction of our method from the axiomatic proof.

The axiomatic method is *synthetic*, its working principle is *construction*. Using this method, we set a goal: to construct a demonstration, which is a certain formal object. To achieve this goal we set subgoals, which in their turn generate subgoals, etc. Our approach is *analytic*, we only examine how configurations turn into one another.

Configuration C^3 in formal arithmetic looks like

(T2) $(x + y)' = y' + x$

Our goal is to transform it into C^1:

(T3) $x + y = y + x$

If we could prove that

(T4) $y' + x = (y + x)'$

then we would combine (T4) and (T2) by the transitivity axiom into:

(T5) $(x + y)' = (y + x)'$

which because of #11.2 turns immediately into (T3). Setting (T4) as a subgoal, we prove it easily by induction, both in formal arithmetic, and in Refal.

This course of action provides a speedy proof of commutativity of addition in formal arithmetic, but guessing (T4) as a subgoal with the subsequent use of transitivity of equality (which for us is only one of the recursive functions!) goes against the grain of our method. We shall try a different approach.

We received C^3 through contractions for e_2. Let us perform one more step of driving with the contractions for e_2:

(3 → 7) $(e_2 \rightarrow 0)(K_2 \rightarrow K_7)$

(3 → 8) $(e_2 \rightarrow e_2 1)(K_2 \rightarrow K_8)$

(C^7) $k = (e_1 1)(k + (01)(e_1) \perp)$

(C^8) $k = (k + (e_1)(e_2) \perp 11)(k + (e_2 11)(e_1) \perp) \perp$

Configuration C^7, like C^2, is easily transformed into T by driving and induction. We may perform several more steps of driving, and we will find that whenever we have our original configuration F *with any specific number* replacing e_2, i.e.:

$$\begin{array}{lll} kF(e_1)(0)\perp & \text{which is} & C^2(e_1) \\ kF(e_1)(01)\perp & \text{which is} & C^7(e_1) \\ kF(e_1)(011)\perp & & \\ \cdots & & \end{array}$$

etc., we are able to transform it into T by applying our equivalence transformation. Thus the idea occurs to us: by analyzing the process of transformation of such configurations, to prove that any of them will be reduced to T; it is equivalent to transforming the original configuration into T.

The formalism which exploits this idea rests on the concept of *integral metafunction*, or just *metaintegral*. In this case we are interested in the metafunction which will be denoted as:

$$k \int F(E_1)(e_2) \perp$$

We read it: the metaintegral of F over e_1. This is a function which depends only on e_2, because E_1 is just a symbol (although nonterminal). For any value $\mathcal{E}$ of e_2, the value of this function is an expression which can be interpreted as a full definition of the function F^1:

$$kF^1(e_1) \Rightarrow kF(e_1)(\mathcal{E}) \perp$$

of one argument e_1.

As an expression representing the definition of a function, we shall use the list of primary walks in the graph of states of this function, and not just the metacode of the definition. There are two advantages to this: first, walks are independent (i.e. interpretation of one does not depend on another); second, it allows us to define a function given by a configuration, without attaching to it any determiner. Consequents of all substitutions in walks should be written in metacode. Recall that by primary walks we mean walks corresponding to the arcs in the graph of states; they may be terminal or nonterminal. The list of all primary walks completely defines the graph of states. Indeed, if K_i is the antecedent of the first replacement on the top level of the walk, then this arc starts from vertex V^i; if the consequent of the last replacement on the top level contains K_j , then the arc leads to an active vertex V^j; if the last consequent does not contain nonterminal K-symbols, the arc leads to a passive vertex (which need not be numbered). We shall separate walks by commas. In addition, we allow the taking out of parentheses of the common parts of walks on the left, so that

(1) $W_c(W_1,\ldots,W_n)$ *is equivalent to* $W_c\, W_1,\ldots,W_c\, W_n$

To give an example, the definition of addition, #10, will be:

$$(E_2 \rightarrow 0)(K_1 \rightarrow E_1),\ (E_2 \rightarrow E_2 1)(K_1 \rightarrow K_1 1)$$

This representation of a function definition is a generalization of the representation we used in Section5.5 for functions defined by one terminal walk. We generalize correspondingly the definition of the *interpretation function* < . From now on it will be applicable to any function definition, and if D^F is the definition of function F, then

(2) $k\langle D^F\rangle$ *input-variable-form* $\perp$ = kF *input-variable-form* $\perp$

We also generalize the definition of function τ: it will be applicable now not only to one walk, but to any list of walks. Thus by τ we shall mean some function which performs an equivalence transformation of a function definition. In virtue of this definition:

(3) $$k\ \langle \tau e_d \perp\rangle\ e_v \perp\ =\ k\ \langle e_d\rangle\ e_v \perp$$

The *full metaintegral* is the metaintegral over all free variables in a function form, e.g.,

$$k \int F(E_1)(E_2) \perp$$

where F is any function of two variables. The full integral is a constant which represents the definition of the function, e.g.

$$(4) \quad k \int + (E_1)(E_2) \Rightarrow (E_2 \to 0)(K_1 \to E_1),\ (E_2 \to E_2 1)(K_1 \to K_1 1)$$

We assume that the full metaintegral of each used function is given (this only means that the function is defined). Then we can find any metaintegral by putting before the full metaintegral assignments $(e_i \leftarrow E_i)$ for those variables over which there is *no integration*, e.g.

$$(5) \quad k \int F(E_1)(e_2) \Rightarrow (e_2 \leftarrow E_2)(k \int F(E_1)(E_2) \perp)$$

$$(6) \quad k \int F(e_1)(E_2) \Rightarrow (e_1 \leftarrow E_1)(k \int F(E_1)(E_2) \perp)$$

$$(7) \quad k \int F(e_x)(E_y)(e_z) \Rightarrow (e_x \leftarrow E_x)(e_z \leftarrow E_z)(k \int F(E_x)(E_y)(E_z) \perp)$$

The metaintegral over *no* variables is by no means equivalent to the original function (there is no *integral*, but *meta* remains). The function

$$(8) \quad k \int F(e_1) \Rightarrow (e_1 \leftarrow E_1)(k \int F(E_1) \perp)$$

with any specific $e_1 = \mathcal{E}$ has a value, which is the definition of a function of no variables, whose value is defined and coincides with the result of concretization

$$kF(\mathcal{E}) \perp$$

if and only if this concretization is possible.

If there is metaintegration over some of the variables in the argument of a function, a *metasystem split of variables* occurs. Let e_1 represent the variables over which there is no integration, and e_2 represent the integrated variables. Then for any e_1 , the metaintegral function will give us something which defines the value of the function with this e_1 and *all possible* e_2. This is why we call this metafunction integral. The values of the differential metafunction define the function on certain minimal, in a sense, subsets of arguments. The values of the integral metafunction define the function on the full set of values of those variables which were removed from the object space. Thus:

$$k \int F(E_1)\ldots(E_n) \perp = \int_{e_w} k\ \partial(e_w)\ F(E_1)\ldots F(E_n) \perp$$

Computing a metafunction (differential or integral) takes us one level up in the metasystem stairway; computing an interpretation function brings us one level down. To describe different schemes of using metasystem transition we use *MST-formulas* (MST stands for "metasystem transition").

An MST-formula for a function F is a definition of F in Refal, which is functionally equivalent to the original definition, but expressed in terms of: (1) computing a metafunction, (2) making equivalence transformation of a definition, and (3) interpreting a definition.

Our examples will be for a function of two e-variables. The process of direct computation of a function call in the Refal machine is described by the formula:

$$(9)\quad kF(e_1)(e_2) \Rightarrow k<k \int F(E_1)(E_2) \perp> (e_1)\ (e_2) \perp$$

Introducing a designation

$$\mathcal{D}^F = k \int F(E_1)(E_2) \perp$$

for the initial definition of function F, we put it in a shorter form:

$$(9')\qquad kF(e_1)(e_2) \Rightarrow k<\mathcal{D}^F> (e_1)(e_2) \perp$$

The process of interpretation of a walk dependent on the initial values assigned to the input variables is the same as the interpretation of this same walk modified by adding corresponding assignments at the beginning. This applies also to a set of walks, i.e. to a function definition. Therefore, (9) may be also written in an equivalent form:

$$(9'')\quad kF(e_1)(e_2) \Rightarrow k<(e_1 \leftarrow E_1)(e_2 \leftarrow E_2)(\mathcal{D}^F)> \perp$$

where the process of interpretation does not require any additional information.

Using a metaintegral over no variables we may write the same MST-formula in one more form:

$$(9''')\quad kF(e_1)(e_2) \Rightarrow k<k \int F(e_1)(e_2) \perp> \perp$$

which very clearly expresses the inverse relationship between metaintegral and interpretation.

The interpretation function < is the definition of the Refal machine in Refal. It is convenient to think of this definition as written for another copy of the Refal machine, which is "observing" the activity of the first, *performing* machine. Then MST-formulas should be thought of as written for the observing machine and defining the use of the performing machine. If we look into the right side of (9) and its equivalents, we read: take the definition of function F in Refal, assign some specific values represneted by e_1 and e_2 in the observing machine to nonterminals E_1 and E_2 representing free variables in the performing machine, and start the performing machine.

Now consider a metasystem split of variables. This is a formula for a direct interpretation of a metasystem integral over one of the variables:

$$(10) \qquad kF(e_1)(e_2) \;\Rightarrow\; k <k \int F(E_1)(e_2) \perp> (e_1) \perp$$

It may be read: take a specific e_2 and put it into the definition of F; then interpret this definition with a specific e_1. One can see that there is no essential difference between this plan of action and the one given in (9). It can be shown formally. Using (5) for the metaintegral in (10), and taking a variable inside the angular brackets as we did in passing from (9') to (9"), we obtain:

$$(10') \qquad kF(e_1)(e_2) \;\Rightarrow\; k <(e_1 \leftarrow E_1)((e_2 \leftarrow E_2)(\mathcal{D}^F))> \perp$$

which because of (1) is equivalent to (9").

For significant new results some equivalence transformation τ must be used. We may introduce τ both in (9) and in (10), obtaining, correspondingly:

$$(11) \qquad kF(e_1)(e_2) \;\Rightarrow\; k<\tau\ k \int F(E_1)(E_2) \perp \perp> (e_1)(e_2) \perp$$

$$(12) \qquad kF(e_1)(e_2) \;\Rightarrow\; k<\tau\ k \int F(E_1)(e_2) \perp \perp> (e_1) \perp$$

These MST-formulas describe the use of equivalence transformation: take a function definition, apply τ to it, and put the result into the Refal machine; then obtain the values of variables and start the machine. The methods of the compilation theory, as described in Chapter 4, were all algorithmical. They can be formalized into a Refal program defining function τ; it will be implied in the following that τ is of that kind, if the opposite is not stated.

No function τ is of course omnipotent. If a function definition is "bad in τ's judgement", it will improve it. Otherwise it will leave it unchanged as "good enough". In the case (11) these considerations are applied directly to the initial definition of function F. In the case (12), however, function τ processes the result of a metasystem split of variables: the definition of a function of one variable expressed through the function F of two variables with the second variable taking a certain value. Thus even if the definition of F is quite "good", the definition to be processed by τ in (12) will be normally "bad". Formula (12), if read in more detail than above, says: take a specific value of e_2 (an object expression), form the metaintegral, transform it by τ exploiting the fact that E_2 in the definition is substituted by an object expression, and then use this definition to compute the overall result as a function of e_1.

For a simple example let us take function + as F. Let e_1 take value 011, and e_2 value 0111. The metaintegral in (12) is:

(13) $(0111 \leftarrow E_2)((E_2 \rightarrow 0)(K_1 \rightarrow E_1), \quad (E_2 \rightarrow E_2 1)(K_1 \rightarrow K_1 1))$

For a human being it is easier to deal with configurations than with walks. Applying τ to (13) is equivalent to driving

$$k + (e_1)\ (0111) \perp$$

which gives

$$e_1 111$$

without any contractions. Therefore, the result of concretizing the τ call will be

$$(K_1 \rightarrow E_1 111)$$

Now we face a problem of concretizing the function $<$, which has not been formally defined. In the case when the contents of the angular brackets is an object expression, concretization of $<$ is very simple: we just use the contents as the definition of our function:

$$k<(K_1 \rightarrow E_1 111)> (e_1) \Rightarrow e_1 111$$

substituting 011 for e_1 , we get the result: 011111 .

Now we make the second metasystem transition. Let us consider the transformed metaintegral in (12) as a function of e_2 :

(14.1) $k/MIF1/(e_2) \Rightarrow \tau\ k \int F(E_1)(e_2) \perp \perp$

Hence formula (12) is represented as

(12') $kF(e_1)(e_2) \Rightarrow k<k/MIF1/(e_2) \perp> (e_1) \perp$

Function /MIF1/, whose computation is an equivalent transformation, will now be subject itself to the same equivalent transformation by using the equivalent of (11) for a function of one variable:

(11') $k/MIF1/(e_2) \Rightarrow k< \tau\ k \int /MIF1/(E_2) \perp \perp> (e_2) \perp \perp$

From (11') and (12') we receive a new MST-formula:

(14.2) $kF(e_1)(e_2) \Rightarrow k<k<\tau k \int /MIF1/(E_2) \perp \perp> (e_2) \perp> (e_1) \perp$

which, together with (14.1), defines a new equivalence transformation.

If function τ is formally defined in Refal, as it should be, we need not bother about the understanding of how formulas (14) work; we just use them and see what happens. As it happens, the metasystem transition largely widens the scope of function definitions which yield themselves to significant improvement. We will show it even for such a "good" function as + , which is characterized by a perfect graph (this means that the effect will not be achieved by the compilation process alone, but generalization and induction will be used). Since we do not have a formal definition of τ, we shall use our human understanding of equivalence transformation.

Function τ which begins the computation in formula (14.2) makes the equivalent transformation of function /MIFl/ defined by (14.1). Let us see what will be happening. The transformation will start with the compilation process. The initial configuration will be:

$$\tau(e_2 \leftarrow E_2)((E_2 \rightarrow 0)(K_1 \rightarrow E_1),\ (E_2 \rightarrow E_2 1)(K_1 \rightarrow K_1 1)) \perp$$

For $e_2 \rightarrow 0$ we receive a very simple passive configuration:

$$(K_1 \rightarrow E_1)$$

For $e_2 \rightarrow e_2 1$ we have the following sequence of configurations:

$$\tau(e_2 1 \leftarrow E_2)((E_2 \rightarrow 0)(K_1 \rightarrow E_1),\ (E_2 \rightarrow E_2 1)(K_1 \rightarrow K_1 1)) \perp$$

$$\tau(e_2 1 \leftarrow E_2)(E_2 \rightarrow E_2 1)(K_1 \rightarrow K_1 1)(\mathcal{D}^+) \perp$$

$$\tau(e_2 \leftarrow E_2)(K_1 \rightarrow K_1 1)(\mathcal{D}^+) \perp$$

$$\tau(K_1 \rightarrow K_1 1)\tau(e_2 \leftarrow E_2)(\mathcal{D}^+) \perp \perp$$

(Here by $\mathcal{D}^+$ we denote the definition of function + which appears in all configurations.)

We see that the initial configuration is recurrent, and the result is the following definition:

#12.1 $\quad$ $k/\text{MIFl}/(0) \Rightarrow (K_1 \rightarrow E_1)$

#12.2 $\quad$ $k/\text{MIFl}/(e_2 1) \Rightarrow (K_1 \rightarrow K_1 1)\ k/\text{MIFl}/(e_2) \perp \perp$

Computing several argument-value forms:

$$\begin{aligned} (0) &= (K_1 \rightarrow E_1) \\ (01) &= (K_1 \rightarrow E_1 1) \\ (011) &= (K_1 \rightarrow E_1 11) \end{aligned}$$

we make an LE-generalization

$$(0e_2) = (K_1 \rightarrow E_1 e_2)$$

with no variables in the right side different from the variable e_2, which appears in the left side. This hypothesis checks true against sentences #12, and this gives us the final definition:

#13 $$k/MIFl/(0e_2) \Rightarrow (K_1 \rightarrow E_1e_2)$$

The τ function call on the second metasystem level (i.e. in the one in (14.2)) will give as output the graph of states corresponding to #13:

$$(E_2 \rightarrow 0E_2)(K_1 \rightarrow (K_1^2 \rightarrow E_1^2E_2)\ )$$

Nonterminals of the second order appear here as the result of the metacode transformation; this shows cearly that we are on the second level.

Now we just drive the MST-formula (14.2). Notice that driving the interpretation function < can be performed by using the same equivalence transformation function τ again! The inner configuration, if represented in a no-argument form, is:

$$k<(e_2 \leftarrow E_2)(E_2 \rightarrow 0E_2)(K_1 \rightarrow (K_1^2 \rightarrow E_1^2E_2)\ )> \perp$$

The clash $(e_2 \leftarrow E_2)(E_2 \rightarrow 0E_2)$ produces the contraction $(e_2 \rightarrow 0e_2)$, which goes into the argument of function F, and the assignment $(e_2 \leftarrow E_2)$, which modifies the replacement. The value of the interpretation function is the consequent of the final replacement for K_1, which results in the following sentence:

$$kF(e_1)(0e_2) \quad \Rightarrow \quad k<(K_1 \rightarrow E_1e_2)>\ (e_1) \perp$$

Driving the second interpretation function call in the same manner, we get the final definition of addition:

#14 $$k + (e_1)(0e_2) \Rightarrow e_1e_2$$

We return now to commutativity of addition, function F being defined again by #9. The metaintegral over e_1 is:

#15 $$k/MIFl/(e_2) \Rightarrow \tau(K_1 \rightarrow =(+(E_1)(e_2))(+(e_2)(E_1))(\mathcal{D}^{+=}) \perp$$

We have used here a specialized metacode, which differs from the standard by the absence of concretization brackents. This is possible thanks to the rigid functional formats we are using; it is assumed, of course, that function τ is modified correspondingly. Also, our representation of definitions is not standard: compositions of configurations are written in their natural form, and not decomposed into a walk with the help of redundant variables. By $\mathcal{D}^{+=}$ we have denoted the definition of functions + and = .

We need now more insight into the performance of function τ, in order to be able to deal with configurations involving τ without having access to its formal definition. There are two aspects to the equivalence transformation. The first is driving, which we already understand well enough. The second is decision making and control of driving, i.e. the strategy of transformation. We shall make use of two facts concerning the strategy.

The first is that function τ does not keep transitory configurations on any level of structure. Wherever a transitory configuration appears, it is immediately transformed into its successor, until it is either passive, or requires a contraction. This may be taken into account by introducing a function, say α, which leaves its argument unchanged if it is not a transitory configuration, and drives it the necessary number of steps if it is transitory. We might then apply function α to every configuration in the argument of τ. We shall not do this literally, in order to keep the record readable, but shall proceed as if this were done, i.e. use driving on the first metasystem level (with respect to E_1) automatically whenever it does not require contraction.

The second fact we need to know about the strategy is how a recurrent configuration is discovered and what happens next. After each step of driving, function τ compares the new configuration with the preceding one, and if they are equal, it tries to use the principle of induction in its simplest form. Namely, if the definition at the moment is of the form:

$$kC^{i}(0) \Rightarrow Z$$
$$kC^{i}(e_{v}1) \Rightarrow kC^{i}(e_{v}) \perp$$

where C^{i} is any configuration, e_{v} is any variable, and Z is any object expression, then it is transformed into

$$kC^{i}(e_{v}) \Rightarrow Z$$

One can see that it is a very primitive strategy, but it is sufficient for both τ-calls (one on the first and the other on the second metasystem level) in this example.

Bearing this in mind we set forth to transform function /MIF1/ defined by #15. To avoid confusion, we shall refer to the function τ on the first metasystem level, which appears in the right side of #15,as ${}^{1}\tau$, while the function τ on the second level, i.e. that which appears in the right side of the MST-formula (14.2) and whose performance we imitate, will be referred to as ${}^{2}\tau$.

Driving ${}^{1}\tau$ we immediately split #15 into:

$$\#16.1 \quad k/MIF1/(0) \Rightarrow {}^{1}\tau(K_1 \to =(E_1)(+(0)(E_1)))\ (\mathcal{D}^{+=}) \perp$$

$$\#16.2 \quad k/MIF1/(e_2 1) \Rightarrow {}^{1}\tau(K_1 \to =(+(E_1)(e_2)1)(+(e_2 1)(E_1)))(\mathcal{D}^{+=}) \perp$$

The right side of #16.1 is something which we very well know already: the theorem about the left addition of zero. It is easily transformed on the first metasystem level, i.e. computed by ${}^{1}\tau$:

$$\#16.1' \qquad k/MIF1/(0) \Rightarrow (K_1 \to T)$$

The configuration in #16.2 is, with respect to ${}^{2}\tau$, transitory, because the next step of driving ${}^{1}\tau$ leads to a split over E_1 on the first level, independently of the value of e_2 on the second level. The following configuration results:

$$(C^{16}) \quad \begin{aligned} &{}^{1}\tau((E_1 \to 0)(K_1 \to =(+(0)(e_2))(e_2)),\\ &\qquad (E_1 \to E_1 1)(K_1 \to =(+(E_1 1)(e_2))(+(e_2 1)(E_1))))(\mathcal{D}^{+=}) \perp \end{aligned}$$

Function ${}^{2}\tau$ will at this stage demand a decomposition to separate the recurrent subconfiguration

$$\alpha = (\alpha + (0)(e_2)\perp)(e_2)\perp$$

It is again the left addition of zero, so that it is again easily transformed (although on the second metasystem level) into T. The configuration to which the arc with the contraction $(E_1 \to E_1 1)$ leads should be compared with the preceding configuration (in #16.2). Thus function ${}^1\tau$ will generate a subfunction call:

$$k/EQ/(=(+(E_1 1)(e_2))(+(e_2 1)(E_1)))(=(+(E_1)(e_2)1)(+(e_2 1)(E_1)))\perp$$

where function /EQ/ is a predicate checking the equality of expressions, not only numbers as = . By removing identical parts it will be right away transformed into configuration:

$$(C^{17}) \quad k/EQ/(+(E_1 1)(e_2))(+(E_1)(e_2)1) \perp$$

Since driving this configuration necessitates a contraction of e_2 , and the configuration proves recurrent, function ${}^2\tau$ will separate it by decomposition, and transform independently. In fact, it is theorem (T4), but with two variables split between two metasystem levels!

Transforming the function of e_2 given by configuration (C^{17}), we should not forget that there are invisible functions α in it, which cause driving in the arguments of function /EQ/ so that C^{17} behaves as if pluses were function calls, i.e. exactly as (T4):

$$\begin{aligned} kC^{17}(0) &\Rightarrow k/EQ/(E_1 1)(E_1 1) \Rightarrow T \\ kC^{17}(e_2 1) &\Rightarrow k/EQ/(+(E_1 1)(e_2)1)(+(E_1)(e_2)11) \\ &\Rightarrow k/EQ/(+(E_1 1)(e_2))(+(E_1)(e_2)1) \\ &\Rightarrow kC^{17}(e_2) \perp \end{aligned}$$

Function ${}^2\tau$ transforms this into

$$kC^{17}(e_2) \Rightarrow T$$

Since the procedure of comparison gives a positive answer, the replacement of the second arc in (C^{16}) will be $(K_1 \rightarrow K_1)$. The whole configuration becomes

$$\begin{aligned} {}^1\tau(\ &(E_1 \rightarrow 0)(K_1 \rightarrow T), \\ &(E_1 \rightarrow E_1 1)(K_1 \rightarrow K_1)\ )\ (\mathcal{D}^{+=}) \perp \end{aligned}$$

which turns into

$$(K_1 \rightarrow T)$$

by induction.

The sentence #16.2 will now become:

#16.2' $\quad k/MIF1/(e_2 1) \Rightarrow (K_1 \rightarrow T)$

so that we can uniteit with #16.1' into one sentence:

$$k/MIF1/(e_2) \Rightarrow (K_1 \rightarrow T)$$

Therefore, the result of concretization of ${}^2\tau$ will be

$$(K_1 \rightarrow (K_1^2 \rightarrow T))$$

and by driving two interpretation functions in (14.2), the same way we did in the preceding example, we get the final result:

$$kF(e_1)(e_2) \Rightarrow T$$

which proves commutativity of addition.

Formulas (14.1) and (14.2) define a new equivalence transformation, constructed on the basis of transformation τ. Put in words it is: take (14) as a new definition of F. Obviously, it can be written as a Refal program.

5.7. Metasystem Analysis.

The use of differential and integral metafunctions, as illustrated in Sections 5.5 and 5.6, opens a new approach to the problems of logic, mathematics and computer science, which we shall call *metasystem analysis*. In this section we only very briefly summarize some primary ideas of metasystem analysis, which still remains to be developed into a full theory.

1. What in an axiomatic theory is a set of mathematical (specific for the theory) axioms, in metasystem analysis is a set of recursive function definitions. This set is called a *mathematical machine*.

2. What in axiomatic theories is a set of logicalaxioms and rules of inference in metasystem analysis is a recursive function τ of equivalence transformation of recursive functions. This function is called a *logical machine*, or just *logic*.

3. The logical machine has a mathematical machine as input, and produces on output its *model*, which is another mathemati-

cal machine. There are three elements, of which the operation of a logical machine is composed:

(1) concretization (computation), including driving;

(2) generalization (empirical induction) with a subsequent proof by mathematical induction;

(3) metasystem transition.

These also are basic elements of human thinking (see Section 5.1).

4. Given a number of logical machines, we can unite them into one, more potent machine, which will make use of them all, and choose the best (in a specified sense) resulting model.

5. Given a logical machine, we can construct a more potent machine using an MST-formula. An example was given in Section 5.6. We saw that while the original τ could not prove the commutativity of addition, the new τ did it. In Section 5.5 we saw an example when a transformation was achieved by taking the metaderivative. It is our hypothesis that with a certain minimum of computational and generalizational capacities, all of the complication necessary to match and transcend human thinking in a computer may be achieved by multiple metasystem transitions.

6. Both metaderivative and metaintegral functions may be used in MST-formulas, which can become very sophisticated. We note that even when a function is written in a format with only one variable, this variable can be *factorized*, i.e. represented as a pattern expression including several new variables; then we can split these variables between two or more metasystem levels. In particular, the metaderivative function is naturally seen as a function of as many variables, as many recurrent configurations are there in the graph of states. Thus while taking the metaintegral reduces the number of variables, taking the metaderivative usually increases it. This provides for many diverse MST-formulas.

7. Progress in mathematics is construction of more and more mathematical machines which model mathematical machines of lower hierarchic levels and phenomena of nonmathematical reality. Progress in logic is construction of more and more logical machines which are more and more potent in producing models. No logic is supreme, because a more potent one can always be constructed by a metasystem transition. A system producing new logic is a *metalogic*. If formalized into a deterministic or nondeterministic machine, a metalogic may not be supreme either, and for the same reason. There exists no ultimate criterion of the reliability of a logic or metalogic other than proof in practice and the resulting intuition.

8. A logical machine is called *an individual logic* if its output is defined only when one specific mathematical machine is input. The usual axiomatic logic is, from the viewpoint of metasystem analysis, a system using only individual logics (called proofs) and a formalized metalogic (called formal logic).

9. Starting with Gödel's theorem, metasystem transition has been extensively used in logic and mathematics to obtain negative results (incompleteness, insolvability, etc.). We embark on using metasystem transition in a positive way: to actually expand (and in the needed direction) the range of possibilities of each specific machine, not only to show that it has limits. Although the range of each machine remains, of course, limited, the process of expansion itslef is unlimited — as far as we can see it now. The Gödel theorem and other negative results set limits for those systems which do not incorporate metasystem transition. Our theory does incorporate metasystem transition as one of its formalized elements. This is why it is free from Gödel's limits. What other limits it has, if any, is not easily seen at the present time.

10. To repeat metasystem transition unlimitedly, we must have a comprising system, which makes metasystem transition uniformly feasible on any level. As shown in [1], evolutionizing biosphere

is one of the systems of that kind, called there *ultrametasystems*. We use Refal as an ultrametasystem — a sort of *characteristica universalis* of Leibnitz. We try to create in the material of symbols a self-developing, evolutionizing system, which would model living structures. Using this model (or theory, as models made in the material of symbols are usually called) we shall possibly be able to better understand the nature and the limits of evolution.

5.8. Algorithmic Impossibility of Ultimate Perfection.

Theorem 5.1. There exists no algorithm which could transform any graph of states into an equivalent perfect graph.

We shall prove this theorem by modeling the functions of formal arithmetic in Refal and showing that if we had an algorithm A referred to in the theorem, we would be able to decide any problem in arithmetic, which is impossible because of Church's theorem.

There are three functions in formal arithmetic, which are modeled by three recursive functions:

$$
\begin{array}{ll}
k = (0)\ (0) & \Rightarrow T \\
k = (e_x 1)(e_y 1) & \Rightarrow k{=}(e_x)(e_y) \perp \\
k = e_z & \Rightarrow F \\
k + (e_x)(0) & \Rightarrow e_x \\
k{+}(e_x)(e_y 1) & \Rightarrow k{+}(e_x)(e_y) \perp 1 \\
k \times (e_x)(0) & \Rightarrow 0 \\
k{\times}(e_x)(e_y 1) & \Rightarrow k{+}(k\ (e_x)(e_y) \perp)(e_x) \perp
\end{array}
$$

Using these functions we can model any predicate $P(x_1,\ldots,x_n)$ in arithmetic which does not include quantifiers. For such predicates P we can prove the following lemma.

Lemma. If we have an algorithm A referred to in Theorem 4.5, we have the decision algorithm for all formulas of the form:

(1) $(Ax_1)(Ax_2)\ldots(Ax_n)(P(x_1,x_2,\ldots,x_n))$

and

(2) $(Ex_1)(Ex_2)\ldots(Ex_n)(P(x_1,x_2,\ldots,x_n))$

Suppose indeed that we do have A. Then we form the graph of states of the function computing P, and transform it into a perfect graph. Consider all those passive terminal vertices in this graph, which are on the same bracket level as the head vertex, i.e. not a part of any composition loop. They fall into three types:

(1) type T, comprising those configurations which always become T after the substitutions of free variables (if any) by their values (the input set corresponding to the vertex being not empty, because the graph is perfect);

(2) type F, defined analogously for the truth value F ;

(3) type T/F, for which two sets are not empty: the set of those exact input states for which concretization stops at the considered vertex V^i with the result T, and the corresponding set for the truth value F.

There is a procedure which for each vertex V^i with configuration C^i decides what type it is. If the vertex is T or F, then it obviously is of the respective type (1) or (2). Otherwise, we make use of the following facts, resulting from the impossilility for the predicate P to take on any value different from T or F:

(a) If configuration C^i includes any symbol at all it must be either T or F. The remaining part of the configuration must in this case be a string of e-variables, which always take only empty values, and we can therefore ignore them. There may be no more than one symbol, nor any parentheses in the configuration.

(b) If the configuration is a string of free variables, then no more than one of them may be an s-variable, and at the end of concretization exactly one variable takes a nonempty value, which may be only T or F.

(c) Since the input variables may consist only of symbols 0 and 1, no variable resulting from contraction may enter the output configurations, all varibles in these configurations must receive their values in assignments. Moreover, the left sides of these assignments cannot be variables defined in contractions, for the same reason. Thus any variable entering configuration C^i must receive its value, in the last analysis, through an assignment of a specific symbol T or F.

Take a variable V_x , which enters configuration C^i. With respect to this variable, we can classify each vertex in the graph as having type T, F, or T/F, depending on whether there is at least one walk passing through this vertex, in which V_x takes a given truth value. It is easy to see that by tracing the graph not more than twice we can label each vertex at which V_x is defined with an indicator of its type. This labeling affects also our vertex V^i.

We label the graph with respect to each variable entering C^i. Then we determine the type of C^i as a sort of conjunction of its types with respect to the entering variables by the rules:

T & T = T

T & F = F & T = T/F

F & F = F

T/F & *anything* = T/F

The justification of these rules is based on the fact that the graph of states is perfect, and on (b) mentioned above. Suppose e.g. that with respect to one of the variables V_1 , the type of V^i is T. This means that there is at least one walk leading to V^i, on which V_1 takes value T. Then because of (b), all other variables that possibly enter C^i must take empty values on this walk, and the result of substitution in C^i will be T. On the other hand, variable V_1 is never F, so that in deciding whether or not configuration C^i may become F, we can discard V_1. Reasoning in this manner, we come to the above rules.

On labeling all passive terminal vertices on the main level of the graph according to their types, we make the decision referred to in the lemma by this simple rule:

formula (1) is true if and only if all the vertices are of type T; formula (2) is true if and only if there is at least one vertex of the type different from F. The lemma is proved.

To sum up, we have demonstrated that assuming the existence of the algorithm A we can define two procedures D^a and D^e which give answers as to the truth of (1) and (2) correspondingly. These procedures can be defined as recursive functions in Refal. Their argument will be the definition of a given predicate P in Refal also. Thus to decide on the truth of (1) and (2) we only have to compute

(D1) $\qquad k\ D^a\ k \int P(E_1)\ \ldots\ (E_n) \perp \perp$

and

(D2) $\qquad k\ D^e\ k \int P(E_1)\ \ldots\ (E_n) \perp \perp$

Now consider the decision problem for:

(3) $(Ex_{n+1})\ldots(Ex_{n+m})(Ax_1)\ldots(Ax_n)(P(x_1,\ldots,x_{n+m}))$

and

(4) $(Ax_{n+1})\ldots(Ax_{n+m})(Ex_1)\ldots(Ex_n)(P(x_1,\ldots,x_{n+m}))$

Let us define function P^3 as

$$k\ P^3\ (e_{n+1})\ldots(e_{n+m}) \rightarrow k\ D^a\ k \int P(E_1)\ldots(E_n)(e_{n+1})\ldots(e_{n+m}) \perp \perp$$

With any specific set of arguments $(e_{n+1} = x_{n+1}),\ldots,(e_{n+m} = x_{n+m})$ predicate P^3 will tell us whether predicate P is true for every set $x_1,\ldots,x_n$. Therefore, if we apply procedure D^e to P^3, i.e., compute

(D3) $\qquad k\ D^e\ k \int P^3(E_{n+1})\ldots(E_{n+m}) \perp \perp$

we shall know whether there is a set of $x_{n+1},\ldots,x_{n+m}$, with which P is true for any $x_1,\ldots,x_n$. Therefore, computing (D3) is a decision procedure for (3). Analogously,

(D4) $\qquad k\ D^a k \int P^4(E_{n+1})\ldots(E_{n+m}) \perp \perp$

where

$$k\ P^4(e_{n+1})\ldots(e_{n+m}) \rightarrow k\ D^e k \int P(E_1)\ldots(E_n)(e_{n+1})\ldots(e_{n+m}) \perp \perp$$

is a decision procedure for (4).

It is easy to write the corresponding MST-formula for any alteration of universal and existential quantifiers. Since any formula in arithmetic can be written in normal prenex form, we have a universal decision procedure. This is impossible, and by contradiction this proves our theorem.

This was an exercise in using metasystem transition for traditional negative purposes.

5.9. Neighborhoods.

Let a function F be given. The set of all expressions for which F is intended (not necessarily *defined*) constitutes *the object space* of function F. We shall refer to the elements of this set as *points*, and denote them by small Latin letters in this section.

To each point *a* in the object space a unique walk corresponds, which is taken by the Refal-machine when it concretizes kF *a* $\perp$. By *subwalks* of a walk we mean parts of it generated by some number *n* of concretization steps, this number being referred to as the *length* of the subwalk. Consider the starting subwalk of length n of the point *a*. The set of all those points in the object space of a function F which have the same starting subwalk of the length n as point *a* is called *the neighborhood of the* n-*th order* of point *a*, and denoted as $\varepsilon^{n}(a)$. The set of all points which have the same full walk as point *a* is referred to as its *ultimate* neighborhood $\varepsilon^{\infty}(a)$. The ultimate neighborhood of a point *a* includes only points for which function F is defined (including point *a* itself). On the other hand, we can speak of starting subwalks for points in which function F is not defined because later in the walk an abnormal stop occurs, or the walk never ends. Thus neighborhoods of a finite order may be defined for points in which function F is not defined, and may include such points even if the point *a* itself is within the domain of function F.

The function which has points of the object space as its arguments, and the corresponding walks as its values, is called the *metaderivative of the second kind* of function F. The right sides of the sentences which define it are the same as in the case of the metaderivative ("of the first kind") introduced in Section 5.5; the left sides are the same as in the original function F. E.g., the function F:

$$\begin{aligned} &kF{+}e_1 &&\Rightarrow -kFe_1 \perp \\ &kFs_a e_1 &&\Rightarrow s_a kFe_1 \perp \\ &kF &&\Rightarrow \end{aligned}$$

for which the metaderivative of the first kind was given in Section 5.5, #1, has this metaderivative of the second kind:

$$\begin{aligned} &\Delta^f{+}e_1 &&\Rightarrow \tau(E_1 \to +E_1)(K_1 \to -K_1)\Delta^f e_1 \perp \perp \\ &\Delta^f s_a e_1 &&\Rightarrow \tau(E_1 \to s'_a E_1)(\backslash\backslash s_a \to +)(K_1 \to s_a K_1)\Delta^f e_1 \perp \perp \\ &\Delta^f &&\Rightarrow (E_1 \to \Box)(K_1 \to \Box) \end{aligned}$$

(Both metaderivative functions could be defined without introducing function τ into the right sides; then we would apply τ to function calls of metaderivatives, as we did in the case of metaintegral.) Using the metaderivative of the second kind, we can find for each point the corresponding walk and determine its input set, thus finding the ultimate neighborhood of the point. Equally easily we find neighborhoods of finite orders.

A neighborhood of a point is generally a union of restricted L-classes. Picking up the class to which the considered point belongs, we receive an L-*neighborhood* of the point. Unfortunately, an L-neighborhood may not be unique, because although in any given partition of a neighborhood into s-restricted L-classes the classes are not overlapping, there may be more than one way to construct the paritition. As an example, consider a function defined by two sentences with the left sides:

$$kF\ A\ e_1\ B \Rightarrow \ldots$$
$$kF\ e_1 \qquad \Rightarrow \ldots$$

The neighborhood of the first order of a point which chooses the second sentence will be

$$e_1 \backslash\backslash\ e_1 \rightarrow Ae_1B$$

Here are two ways to represent this set (which is restricted, but not an s-restricted L-class) as a union of s-restricted L-classes:

First Representation		Second Representation	
(C_1^1)	□	(C_1^2)	□
(C_2^1)	$s_2e_1 \backslash\backslash s_2 \rightarrow A$	(C_2^2)	$e_1s_2 \backslash\backslash s_2 \rightarrow B$
(C_3^1)	A	(C_3^2)	B
(C_4^1)	$Ae_1s_2 \backslash\backslash s_2 \rightarrow B$	(C_4^2)	$s_2e_1B \backslash\backslash s_2 \rightarrow A$

Consider expression AC as a point. If we build its L-neighborhood according to the first representation, it will be C_4^1 ; according to the second representation, it will be C_2^2. We note that in this particular case the second representation gives a better result, i.e. a larger neighborhood, because $C_4^1 \subset C_2^2$; thus we have good reason to choose C_4^1 as the L-neighborhood. But consider point CD. The first representation gives C_2^1 , the second representation gives C_2^2 , and neither is more general than the other.

Neighborhoods are useful for *controlled concretization*, by which we mean a concretization process which uses knowledge about future concretization steps. This knowledge may be such as:

(1) concretization will never end, therefore there is no sense in continuing it;

(2) concretization will certainly end, so that we can continue it in a usual (uncontrolled) way; in addition we may obtain an estimate of the number of steps required to bring it to an end.

(3) although the stage of the process we are in is repretitive, it cannot repeat but a finite number of times; this may encourage

us to continue concretization through this stage to see what will happen next;

(4) no prediction can be made at this time.

Consider e.g. a function

#1.1 $kFAe_1 \Rightarrow kFA \perp$

#1.2 $kFe_1 \Rightarrow e_1$

and suppose we concretize $kFABC \perp$. After the first step we have $kFA\perp$. We continue concretization and after the second step receive $kFA \perp$ in the view-field. We notice that it is exactly the same state of the Refal machine as we had at the preceding step, hence it will repeat itself endlessly, and concretization will never end.

Consider another example:

#2.1 $kF(e_1)+ \Rightarrow kF(e_1+)+ \perp$

#2.2 $kF(e_1)- \Rightarrow e_1$

and we start with $kF(A)+ \perp$ in the view field. After the first step it will become $kF(A+)+ \perp$, then $kF(A++)+\perp$, then $kF(A+++)+ \perp$ and so on to infinity. However we have a different view-field at each stage, and the simple criterion used in the first example will not work.

To make predictions about the behavior of a process becomes easier if the process is a repetitive application of the same transformation at each step. This is the case for a type of function, which we shall call *while functions*. The definition of a while function may consist only of sentences of two types:

(1) $kFL \Rightarrow kFR \perp$

and

(2) $kFL \Rightarrow R$

Here by F we understand any determiner, L is an L-expression and R is a *pattern* expression. Sentences of both types define, essentially, a transformation of one pattern expression into another:

(3) $L \Rightarrow R$

the difference between them being that type (1) causes continuation of the transformation, while type (2) ends it.

The graph of states of a while function has a very simple structure presented in Figure 19.

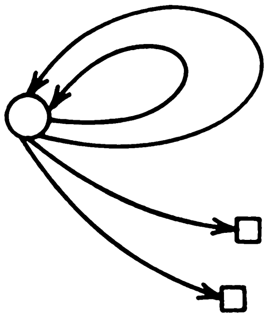


Figure 19

The walks in this graph have no composition loops and are concatenated as strings, not as expressions. The effect of any walk may be discussed in terms of the transformations (3).

Although while functions in Refal are very special, they are sufficient to define any algorithm. Any function defined in Refal can be redefined as a while function using a simple mechanical procedure. The essence of this procedure is to code the whole view-field of the Refal machine as the argument of the new function, and imitate the operation of the Refal machine, which is the same from step to step. One of the two components of executing a step in the Refal machine is the search of the leading concretization sign; this may be done using the multibracket techniques of an all-level scan described in Section 2.7. The other component, the application of sentences, may parallel very closely the original sentences; the difference will be that it will only affect a part of the argument.

As an example, let us present the definition of function F of Section 5.6, expressing the proof of the commutativity of addition (see p. 197), as a while function:

#3.1 $\mathrm{kF}(e_1)(e_2) \Rightarrow \mathrm{kF1}((=))+(e_1)(e_2)((+(e_2)(e_1))*) \perp$

#3.2 $\mathrm{kF1}((=))+(e_1)(0)\ e_x((+(e_2)(e_1))*)$

$$\Rightarrow \mathrm{kF1}((=(e_1e_x)))+(e_2)(e_1)(*) \perp$$

#3.3 $\mathrm{kF1}((=(e_1)))+(e_2)(0)\ e_x\ (*) \Rightarrow \mathrm{kF} = (e_1)(e_2e_x) \perp$

#3.4 $\mathrm{kF1}(e_x)+(e_1)(e_2 1)e_y \Rightarrow \mathrm{kF1}(e_x)+(e_1)(e_2)1\ e_y \perp$

#3.5 $\mathrm{kF} = (0)(0) \Rightarrow \mathrm{T}$

#3.6 $\mathrm{kF}=(e_1 1)(e_2 1) \Rightarrow \mathrm{kF}=(e_1)(e_2) \perp$

#3.7 $\mathrm{kF} = e_x \Rightarrow \mathrm{F}$

(This definition is only a little longer than the original, but what a difference in readability!)

Suppose we noticed in the course of a controlled concretization that a subwalk W has a tendency to reproduce itself. Let point *a* be the argument at the beginning of the subwalk W (in any of its occurrences), and point *a'* the argument at the end of the walk. The effect of walk W is a transformation (3) of an input set *L* into an output set *R*. If the length of walk W is n, the set *L* is the neighborhood of the n-th order of the point *a*. Thus

$$a \Rightarrow a'$$
$$a \in L$$
$$a' \in R$$

Since immediately after the end of the considered occurrence of W another occurrence of W begins (as we have assumed), point *a'* must also belong to *L*:

$$a' \in L$$

What is the relation between the sets *L* and *R*? There are three possibilities, depicted in Figure 20:

(a) $R \subseteq L$

(b) $R \supset L$

(c) $R \supset R \cap L \supset \emptyset \ \& \ \neg R \supset L$

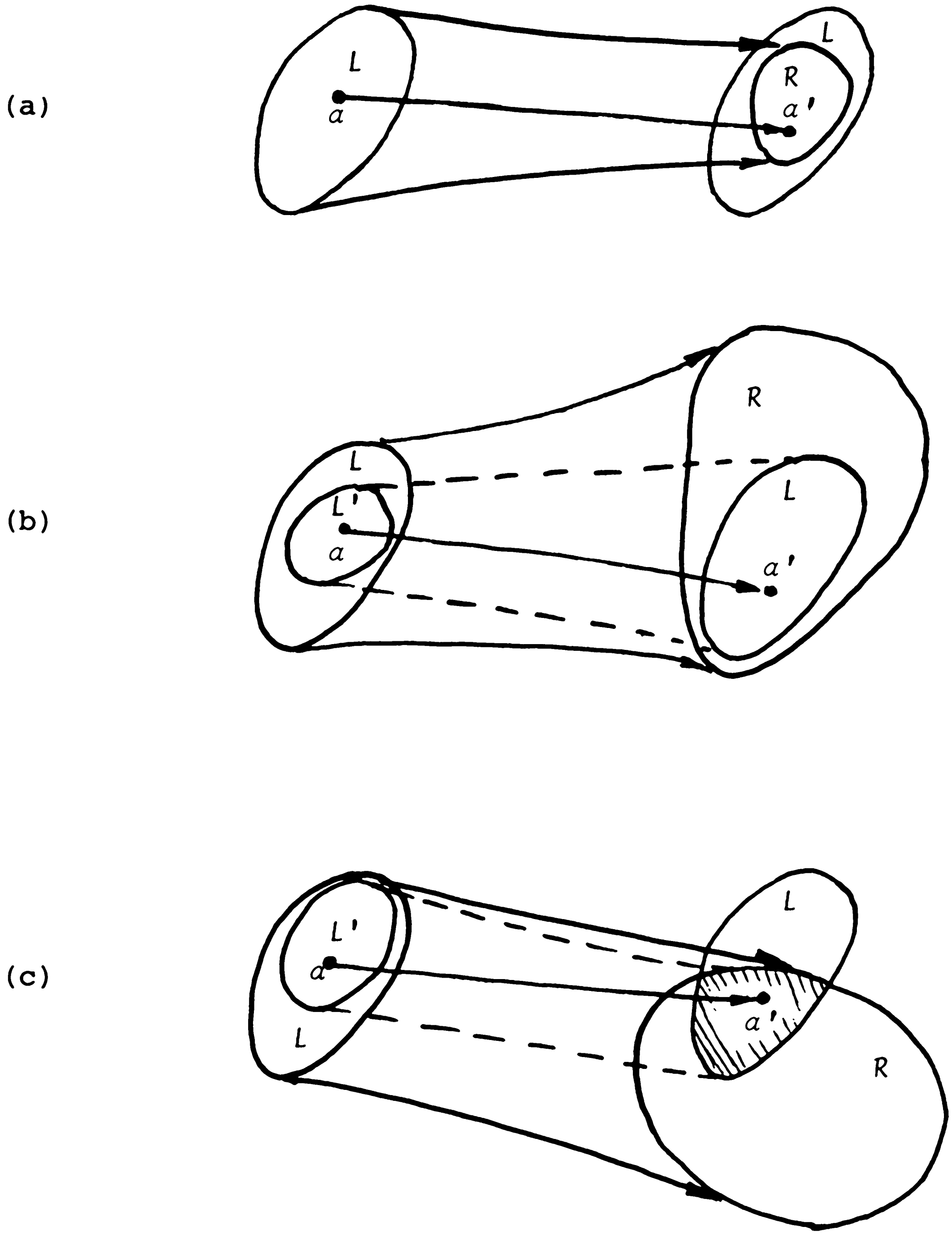


Figure 20

Consider case (a). All points of the neighborhood L of point a remain in the neighborhood L after transformation. This means that after any occurrence of subwalk W another occurrence of W will invariably follow, and concretization will never end. This gives us a criterion of endless concretization. In the example of #2, the neighborhood L is $(e_1)+$, and its transformation R is $(e_1+)+$. Therefore $R \subseteq L$, and our criterion correctly predicts a nonstop computation.

Cases (b) and (c) we shall consider assuming that L is one s-restricted L-class. If it is a union $L_1 \cup L_2 \cup \ldots \quad L_n$ then to each L_i its R_i will correspond, and instead of one subwalk we shall have to consider n subwalks.

Both in case (b) and (c) there are points in R which lie outside of L. Take one such point b' and consider point b whose transformation it is. If by L' we denote the neighborhood of point a corresponding to the doubled subwalk WW, then b must be outside of L', because if it were in L' its first transformation should have been in L to repeat subwalk W. Thus $L' \subset L$.

In case (b) there exists a substitution Δ such that

$$L = R \mathbin{//} \Delta \tag{4}$$

By substitution we mean here any *constriction term* (see Section 4.3), e.g.

$$\mathbin{//} \Delta = (\mathbin{//} e_1 \rightarrow s_a e_1)\ (\mathbin{//} e_2 \rightarrow \square)(\backslash\backslash\ s_a \rightarrow A)$$

and it is only for the sake of brevity that we are using here the notation sign // for "positive" contraction.

Applying substitution Δ to (3) and using (4) we have:

$$L \mathbin{//} \Delta \Rightarrow L \tag{5}$$

Let us denote by $I^n(W)$ the input set of the sequence of n subwalks W. By definition, $I^{n+1}(W)$ consists of all those points, which first are in the input set of the subwalk W and second, after being transformed, hit the set $I^n(W)$. Both conditions are reflected in the relation:

(6) $$I^{n+1}(W) \Rightarrow I^{n}(W)$$

which can be taken as a recursive definition (necessary and sufficient) of the input sets $I^n(W)$. For n = 1:

(6') $$I^{1}(W) = L$$

Because of (6) we can see (5) as the definition of $\varepsilon^2(a)$:

(7) $$\varepsilon^2(a) \;=\; L' = I^2(W) = L \,//\, \Delta \Rightarrow L = I^1(W)$$

Making the substitution Δ again, we obtain

(8) $$\varepsilon^3(a) = I^3(W) = L \,//\, \Delta \,//\, \Delta \Rightarrow L \,//\, \Delta = I^2(W)$$

and so on to infinity: in the sense that we may write such a formula for the neighborhood of any order n, while point a is not specified. But with any *specific* a we will sooner or later come to a contradiction. We saw that L' is a proper subset of L, so that substitution Δ, which is the same on all steps, is not trivial. Therefore, we are building a sequence of different s-restricted L-classes which all contain point a. According to Theorem 4.5 there may be only a finite number of such classes. With some n, point a will find itself outside of the input set of the sequence of n subwalks W. Subwalk W may not be repreated endlessly, and this is true for any a.

Consider this simple example. Subwalk W is the first sentence in the definition of function F:

$$\begin{aligned} &kFs_1e_2 \Rightarrow kFe_2 \perp \\ &kF \quad\;\; \Rightarrow \end{aligned}$$

Here L is s_1e_2 , and K is e_2. Case (b) is taking place. Since W is the only nonterminal subwalk in the graph of states, our criterion predicts a finite concretization process for any argument, without actually doing the concretization.

Case (c) requires a more detailed consideration, which will not be carried out here. Our aim in this section is only to present the basic ideas concerning neighborhoods and their use in the analysis of algorithms.

There is a useful hybrid between concretization and driving: driving of a *point with a neighborhood*. Consider a pattern expression E and a substitution Δ, which turns *every* free variable present in E into an object expression. This pair will be referred to as a point with a neighborhood, or a "neighbored point". We shall denote a neighbored point as $E // \Delta$ without actually making the substitution. Should we do so, the result will be the "point"; the "neighborhood" is E.

The essence of the process is as follows. Suppose a point, i.e. an object expression, E_0 is given. For the beginning, we convert it to a neighbored point by attaching to it the universal neighborhood e_1:

(9) $$e_1 // e_1 \rightarrow E_0$$

Then we start driving the neighborhood, but instead of examining all branches resulting from different sentences, as in full driving, we choose at each step only one branch: that taken by the point. Thus each time when a contraction of a variable in the neighborhood becomes necessary we consult the substitutions defining the point. If the contraction, which we shall refer to as C, does not contradict the value taken by the variable according to the substitution, then it is carried out, which changes both the neighborhood and the substitution (without changing the point, naturally); the neighborhood narrows around the point. Applying a sentence we change, of course, the neighborhood without changing the substitution. If contraction C is impossible, it can be so because of two reasons. First, it may happen that the neighborhood as a whole is such that it does not allow contraction, i.e. no point of it will take this branch. In this case we do not change anything but just come over to the next sentence. Second, the neighborhood may allow contraction, but the point lies in that part of it which does not take the considered branch. In this case we start the process of narrowing the neighborhood around the point performing elementary contractions constituting C, and continue until the moment when the narrowed neighborhood no longer allows the next needed elementary contraction. Then we come to the next sentence (branch). Acting in this fasion, we shall have at each stage of concretization *a* neighborhood of our point, i.e. *a* set of points which have exactly the same experience in being treated by the Refal machine as the considered point E_0. It does not guarantee us though that the set includes *all* such points.

Driving of a point with a neighborhood may be used in controlled concretization, and also in the compilation process to find generalized configurations. We shall give an example of the latter.

Consider a function definition

#4 $\quad kF^0e_1 \Rightarrow kF(\ )e_1 \perp$

#5.1 $\quad kF(e_1)+e_2 \Rightarrow kF(e_1-)e_2 \perp$

#5.2 $\quad kF(e_1)s_ae_2 \Rightarrow kF(e_1s_a)e_2 \perp$

#5.3 $\quad kF(e_1) \Rightarrow e_1$

Let the initial configuration be $kF^0e_1 \perp$ (our goal is to just reproduce the definition by driving, because it is perfect). The algorithm of driving is dealing with metacodes of function definitions. The metacode of the initial configuration is

$$*K(\ F^0E_1) \tag{10}$$

Instead of simply applying the algorithm of driving to (10), i.e. concretizing

$$k\ /DRIVING/\ *K\ (F^0E_1) \perp$$

we are going to drive (10) as a point with a neighborhood. The reason is to bring to light all (or at least some) configurations which are indistinguishable from (10) in the eyes of the driving algorithm, and merge them into one generalized configuration. Reviewing the principles of the theory of compilation, we find this method of generalization highly adequate: on the one hand, we move towards the goal of building a complete graph, so long as we succeed in generalizing; on the other hand, we cannot lose efficiency by overgeneralization and the resulting shift to interpretation, because the individual configurations merged into one are treated by the Refal machine the same as the generalized configuration.

Doing driving by hand, it is more convenient to deal with configurations in their natural form (zero metasystem level) than in metacode, like (10); we have been accustomed to this representation already. But neighborhoods of the points of the first metasystem level like (10) are represented through free variables. When we step down to the zero level, we need some representation for these first-level free variables, in order not to confuse them with zero-level variables e_1 , e_x , etc., which are images of nonterminals E_1,E_x,etc. We shall call them metavariables and represent them as $\bar{e}_1$,$\bar{e}_x$,etc.

So we build the universal neighborhood of (10):

(11) $$k\ \bar{e}_1 \perp // \ \bar{e}_1 \rightarrow F^0\ e_1 \ ,$$

and start driving. We try to apply #4. The result will depend on the first symbol of the value of the metavariable $\bar{e}_1$. Thus the narrowing of the neighborhood becomes necessary:

(12) $$kF^0\bar{e}_1 \perp // \ \bar{e}_1 \rightarrow e_1$$

Now the whole neighborhood is driven through #4, and the new point with a neighborhood is:

(13) $$kF(\ )\ddot{e}_1 \perp // \ \bar{e}_1 \rightarrow e_1$$

Each configuration which is potentially recurrent should be traced starting with the universal neighborhood, in order not to lose any chance of generalization. Configuration (12) proved transitory, and we lose interest in it. We now trace (13), starting again with maximum generalization:

(14) $$k\,\bar{e}_1 \perp // \bar{e}_1 \rightarrow F(\ )e_1$$

Driving through #5.1 implies three consecutive contractions for $\bar{e}_1$:

$$(\bar{e}_1 \rightarrow F\bar{e}_1)\ (\bar{e}_1 \rightarrow (\bar{e}_2)\,\bar{e}_1)\ (\bar{e}_1 \rightarrow +\,\bar{e}_1)$$

The first two of these lead to the narrowing of the neighborhood:

(15) $$kF(\bar{e}_2)\,\bar{e}_1 \perp // \bar{e}_1 \rightarrow e_1 // \bar{e}_2 \rightarrow \square$$

The last one cannot be done in the general form, because e_1 is, in fact, an object expression E_1 (*E1, to be precise), which is not identical to + . Since in order to discover it we had to examine the substitution in (15) this means one more narrowing is needed:

(16) $$kF(\bar{e}_2)\,e_1 \perp // \bar{e}_2 \rightarrow \square$$

Now we must remember that the driving of a point with a neighborhood refers to the function /DRIVING/, not to the function F. Driving will demand the contraction $e_1 \rightarrow +e_1$, which will generate a new configuration

(17) $$kF(\bar{e}_2\ -)\,e_1 \perp // \bar{e}_2 \rightarrow \square$$

By *metasystem reduction* (see Section 5.1) we treat the neighborhood in (17) as the configuration

(C^{17}) $$kF(e_2\ -)\,e_1 \perp$$

and compare it with the preceding configuration, resulting in the same manner from the neighborhood in (16):

(C^{16}) $$kF(e_2)\,e_1 \perp$$

We see that $C^{17} \subseteq C^{16}$, which allows looping, Continuing exploration of configuration C^{16}, we easily reproduce the original definition. The important thing was to find configuration C^{16}, and this was done by driving with a neighborhood.

5.10. Supercompiler System.

The supercompiler consists of two parts: a program which performs the equivalence transformation τ, and a program which maps the resulting graph of states on the target machine. The first part is by far more important; the second part only transfers the algorithm from one machine to another (*algorithmic* equivalence as compared to *functional* equivalence in the case of τ), which may be accompanied by no more than a constant factor gain in efficiency. We shall indicate that the output of the equivalence transformation is a program for a machine different from the Refal machine by adding the superscript m to τ; thus the supercompiler will be a Refal function τ^m.

To write MST-formulas in a situation involving a target machine different from the Refal machine, we shall use a notation closely related to the interpretation function < for the Refal machine. We wrote

$$k <P> A \perp$$

to represent the work of the Refal machine applying a function definition (i.e. a program) P to a list of arguments (input set) A. To indicate that a different machine is meant, we add the superscript m to the concretization sign k. Thus

$$k^m <P> A \perp$$

will represent the process and the result of the target machine's work when loaded with program P and input data A.

In the beginning of Chapter 4 we started reviewing the use of languages defined in the Refal system by their interpreting functions. Let us now complete the review assuming that we have the supercompiler τ^m for a given target machine.

Let L be the interpreting function of a language L, so that

(1) $$kL(e_p)(e_d) \perp \perp$$

is the application of a program e_p in L to input data e_d.

The most straightforward way to use the supercompiler and the target machine is to translate the Refal program for L into the language of the target machine with the supercompiler and turn over the result to the target machine for execution. This is described by the MST-formula:

$$(2) \quad kL(e_p)(e_d) \Rightarrow k^m <\tau^m\ k \int L(E_p)(E_d) \perp \perp> \ (e_p)\ (e_d) \perp$$

The expression in the angular brackets (the program for the target machine) does not depend on either e_p or e_d. One can compute it only once by the Refal interpreter, based on the definition of L, and then use it on the target machine each time when e_p and e_d are given. The transition from (1) to (2) may result in an essential gain in efficiency because of two reasons. First, the definition of the language L in Refal may be in a "heavy" interpretation mode, using a hierarchy of auxiliary functions. In this case the redundancy of the definition will be eliminated during the compilation process, and the resulting program will be much more efficient than direct concretization of (1). This gain in efficiency is not connected with the transition from the Refal machine to the target machine: we could achieve it inside the Refal system using equivalence transformation τ according to the formula:

$$(3) \quad kL(e_p)(e_d) \Rightarrow k<\tau k \int L(E_p)(E_d) \perp \perp> (e_p)(e_d) \perp$$

The second source of the possible gain in efficiency is in transition from concretization in the Refal machine (which in practice is interpreted in a computer) to direct operation of the target machine. As mentioned above, this may multiply the efficiency by a constant factor.

With all that, the use of the language L according to (2) remains interpretive, and therefore not fully efficient. Can we produce an efficient program for the target machine corresponding to a specific program in L (i.e. the value of e_p), when only this program, but no input data e_d , is given? Yes, we can. It is a metasystem split of variables. Consider the partial metaintegral

(4) $$kI^1(e_p) \Rightarrow \tau^m k \int L(e_p)(E_d) \perp \perp$$

and the corresponding MST-formula:

(5) $$kL(e_p)(e_d) \Rightarrow k^m \langle kI^1(e_p) \perp \rangle \; (e_d) \perp$$

When a program e_p is given, we compute function (4) on the Refal machine: once and forever. It is an efficient compiled program for the target machine, to be used with any input data e_d according to (5).

So in function $I^1(e_p)$ we have a compiler for the language L. It may give high quality programs, but as it is defined, it works itself in the interpretation mode, depending on the Refal machine. Can we build a compiler which would work fast and nice on the target machine? Certainly. We make one more metasystem transition:

(6) $$kI^2 \Rightarrow \tau^m k \int I^1 (E_p) \perp \perp$$

(7) $$kL(e_p)(e_d) \Rightarrow k^m \; \langle k^m \langle kI^2 \perp \rangle \; (e_p) \perp \rangle \; (e_d) \perp$$

Function I^2 of no variables can be computed on the Refal machine. It is a compiled compiler for the language L. According to (7), we first use the target machine to translate e_p , then apply the result to input data e_d. Note that to produce this compiler we need only function τ^m (the supercompiler), applying it, as one can see comparing (6) and (4), to itself.

Furthermore we can produce a *compiler compiler* implemented on the target machine, using the Refal interpreter with the function τ^m only once. According to the definition of partial metaintegral,

$$(8)\qquad k \int L(e_p)(E_d) \Rightarrow (e_p \leftarrow E_p)(k \int L(E_p)(E_d) \perp)$$

The full metaintegral in the right side is the definition of the function L. Let us generalize function I^1 to I^{f1}, which includes the dependence on the functional definition:

$$(9)\qquad k\ I^{f1}(e_p)(e_f) \Rightarrow \tau^m(e_p \leftarrow E_p)(e_f)\perp$$

Accordingly,

$$(10)\qquad k\ I^{f2}(e_f) \Rightarrow \tau^m\ k \int I^{f1}(E_p)(e_f) \perp \perp$$

Now the metaintegral we need for (6) is expressed through the generalized metaintegral as

$$(11)\qquad k\ I^2 \Rightarrow k\ I^{f2}(\ k \int L(E_p)(E_d) \perp\ ) \perp$$

Making the third metasystem transition, we receive the compiler compiler:

$$(12)\qquad k\ I^{f3} \Rightarrow \tau^m\ k \int I^{f2}(E_f) \perp \perp$$

which is used according to the formula:

$$(13)\quad kL(e_p)(e_d) \Rightarrow k^m{<}k^m{<}k^m{<}kI^{f3}\perp{>}(k\int L(E_p)(E_d)\perp)\perp{>}(e_p)\perp{>}(e_d)\ \perp$$

The compiler compiler (12) is as universal as the supercompiler τ^m itself. It can be computed using the Refal machine only once.

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 an 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* progorams 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 a *target machine.* Counting on the supercompiler, we can program in a much freer style than we did in Chapter 2 when the program was 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 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 target 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 it to the algorithm in order to achieve high efficiency. In 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 target 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 compiler from L was made.

* * * * * * * * * *

REFERENCES

(The titles of the papers in Russian are given in English translation only, which is indicated by enclosure within square brackets.)

1. *Bazisnyi REFAL i yego realizatsiya na vychislitelnykh mashinakh* [Basic REFAL and its implementation on computers]. GOSSTROY SSSR, TsNIPIASS, Moscow, 1977. (The authors are not indicated in the book. In fact they are: Khoroshevsky, V.F., Klimov And. V., Klimov Ark. V., Krasovsky A.G., Romanenko S.A., Shchenkov, I.B., Turchin, V.F.)
2. Turchin, V.F., *The Phenomenon of Science*, Columbia University Press, New York, 1977.
3. Turchin, V.F., "[A metalanguage for formal description of algorithmic languages]," in: *Tsifrovaya Vychislitelnaya Technika i Programmirovanie*, Sov. Radio, Moscow, 1966, pp. 116-124.
4. Turchin, V.F., "[The Metaalgorithmic Language]", Kibernetika No. 4, 1968, pp. 45-54.
5. Turchin, V.F., *Algoritmicheskiy Yazyk Rekursivnykh Funktsiy - REFAL* [Recursive Functions Algorithmical Language - REFAL], preprint IPM*, Moscow, 1968.
6. Turchin, V.F. and Serdobolski, V.I., "[The Language REFAL and its Application for Algebraic Manipulation]", Kibernetika, No. 3, 1969, pp. 58-62.
7. McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine," Comm. ACM 3, 184 (1960).
8. Yngve, V.H., "COMIT," Comm. ACM 6, 83 (1963).
9. Markov, A. A., *Teoria Algorifmov* (The Theory of Algorithms), Trudy Matem. Inst. AN SSSR, 1954.
10. Dijkstra, E.W., "An attempt to unify the constituent concepts of serial program execution," in: Symbol Languages in Data Processing, Gordon & Breach, 1962.

*Institute of Applied Mathematics, Academy of Sciences of the USSR.

11. Dijkstra, E.W., "On the design of machine independent programming languages," in: Annual Review in Autom. Progr. 3, 1963.

12. Van Wijngaarden, A., "Generalized ALGOL," in: Annual Review in Autom. Progr. 3, 1963.

13. Farber, D. J., Griswold R.E., Polonsky I.P., "SNOBOL, A String manipulation language," J. ACM 11 (1) (Jan. 1964).

14. Guzman, A., and McIntosh, H.V., "CONVERT", Comm. ACM 9 604 (1966).

15. Florentsev, S.N., Oliunin, Yu.V., Turchin, V.F., "[REFAL-interpreter]", *Trudy I Vsesoyuznoi Konfer. po Programmirovaniyu*, Kiev, 1968.

16. Florentsev, S.N., Oliunin, Yu. V., Turchin, V.F., *Effektivnyi interpretator dlya yazyka REFAL* [An efficient interpreter for REFAL]. Preprint, IPM AN SSSR,* 1969.

17. Bobkova, O. F., et al. "[A REFAL interpreter for M-220 computer]" in: *Yazyki programmirovaniya i metody ikh realizatsii*, Kiev, 1973.

18. Romanenko, S.A. and Turchin, V.F. "[A REFAL-compiler]" in: *Trudy 2 Vsesoyuznoi konfer. po programmirovaniyu*, Novosibirsk, 1970.

19. Klimov, A.V., Romanenko, S.A., Turchin, V.F. *Kompilator s yazyka REFAL* [A compiler for the language REFAL]. Preprint IPM AN SSSR,* 1972.

20. Turchin, V.F. *Programmirovaniye na yazyke REFAL* [Programming in the language REFAL]. Preprints No. 41, 43, 44, 48, and 49 of the IPM AN SSSR,* 1971.

21. Turchin, V.F. "[An ALGOL translator written in REFAL]" in: *Trudy I Vsesoyuznoi Konfer. po programmirov.*, Kiev, 1968.

* Institute of Applied Mathematics, Academy of Sciences of the USSR.

22. Bychkov, S.P., et al. *Yazyk SIMULA v monitornoi sisteme DUBNA dlya BESM-6* [The language SIMULA in the Monitoring System DUBNA for BESM-6 computer]. Preprint 118 IPM AN SSSR,* 1975.

23. Budnik, A.P., et al. "[The Basic Wave Functions and Operators Matrices in the Collective Nuclear Model]", Yadernaya Fizika 14, No. 2, 1971, p. 304-313.

24. Turchin, V.F. "[Equivalent transformation of Recursive Functions defined in the language REFAL]", in: *Trudy Vsesoyuznogo simposiuma "Teoriya Yazykov i Metody Programmirovaniya , Alushta,"* Kiev, 1972, p. 31-42.

25. Turchin, V.F. "[Equivalent transformation of REFAL programs]", *Avtomatizirovannaya Sistema Upravleniya Stroitelstvom. Trudy TsNIPIASS*, GOSSTROY, Moscow, 1974, pp. 36-38.

26. Ershov, A.P., "[On the essence of translation]", *Programmirovanie, 5,* p. 21-39, 1977. English translation: Neuhold, E.J., Editor, *Formal Description of Programming Concepts*, North-Holland Publ. Co., 1978, pp. 391-418.

27. Turchin, V.F., "A supercompiler system based on the language REFAL," *SIGPLAN Notices 14* (2) (Feb. 1979) 46-54.

28. Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R., *Structured Programming*, Academic Press, 1972.

INDEX