The Concept of a Supercompiler

VALENTIN F. TURCHIN
The City College of New York

A supercompiler is a program transformer of a certain type. It traces the possible generalized histories of computation by the original program, and compiles an equivalent program, reducing in the process the redundancy that could be present in the original program. The nature of the redundancy that can be eliminated by supercompilation may be various, e.g., some variables might have predefined values (as in partial evaluation), or the structure of control transfer could be made more efficient (as in lazy evaluation), or it could simply be the fact that the same variable is used more than once. The general principles of supercompilation are described and compared with the usual approach to program transformation as a stepwise application of a number of equivalence rules. It is argued that the language Refal serves the needs of supercompilation best. Refal is formally defined and compared with Prolog and other languages. Examples are given of the operation of a Refal supercompiler implemented at CCNY on an IBM/370.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**] Applicative (Functional) Programming; D.3.1 [**Programming Languages**] Formal Definition and Theory; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, optimization, compiler generators*; F.1.1 [**Computation by Abstract Devices**] Models of Computation

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: Program transformation, lazy evaluation, Refal, metasystem transition, driving, supercompiler, Prolog

1. INTRODUCTION

Consider a function $f(x, y)$ and give to y some value Y, while leaving x variable. This defines the function $g(x) = f(x, Y)$. For a mathematician, $g(x)$ and $f(x, Y)$ represent the same function of one variable. For a computer scientist, a function is not only its definition but also an algorithm used to compute the values. From this point of view, $g(x)$ and $f(x, Y)$ stand for different things. If there is an algorithm for $f(x, y)$, then $f(x, Y)$ stands for the algorithm defined as follows: give to y the value Y and activate the algorithm for $f(x, y)$. What $g(x)$ means algorithmically is left open. There is often an algorithm for $g(x)$ that is much more efficient than $f(x, Y)$. For example, if $f(x, y)$ does not depend on x at all, we can compute the constant $f(x, Y)$ and define $g(x)$ to be this constant. With $f(x, Y)$, we compute this constant each time we need it.

This work was supported in part by the National Science Foundation under grant DCR-8007565.
Author's address: Department of Computer Science, The City College of New York, New York, NY 10031.

Thus the following problem, known as *partial evaluation* of a function, can be formulated. Given an algorithm—presumably efficient—for $f(x, y)$, and a value $y = Y$, create an efficient algorithm for $g(x) = f(x, Y)$. This is one of the problems that can be solved by a program we call a *supercompiler*. As we shall see, the name reflects the way the program works and one of its immediate applications.

Although supercompilation includes partial evaluation, it does not reduce to it. Supercompilation can lead to a very deep structural transformation of the original program; it can improve the program even if all the actual parameters in the function calls are variable. The supercompilation process aims at the reduction of redundancy in the original program, but this redundancy does not necessarily come from fixed values of variables; it can result from nested loops, repeated variables, and so on. We give examples of how a supercompiler transforms a two-pass algorithm into a one-pass, and how an improvement can result from the simple fact that the same variable is used in two places.

A supercompiler is a program transformer of a certain type. The usual way of thinking about program transformation is in terms of some set of rules which preserve the functional meaning of the program, and a step-by-step application of these rules to the initial program. This concept is suggested by axiomatic mathematics. A rule of transformation is seen as an axiom, and the journal of a transformation process as the demonstration of equivalency. The concept of a supercompiler is a product of cybernetic thinking. A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the running of the machine that is represented by the program; let us call this machine M_1. In observing the operation of M_1, the supercompiler COMPILES a program which describes the activities of M_1, but it makes shortcuts and whatever clever tricks it knows in order to produce the same effect as M_1, but faster. The goal of the supercompiler is to make the definition of this program (machine) M_2 self-sufficient. When this is acheived, it outputs M_2 in some intermediate language L^{sup} and simply throws away the (unchanged) machine M_1.

The supercompiler concept comes close to the way humans think and make science. We do not think in terms of rules of formal logic. We create mental and linguistic *models* of the reality we observe. How do we do that? We observe phenomena, generalize observations, and try to construct a self-sufficient model in terms of these generalizations. This is also what the supercompiler does. Generalization is the crucial aspect of supercompilation. A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between possible configurations of the computing system. However, this process (called *driving*) can usually go on infinitely. To make it finite, the supercompiler performs the operation of generalization on the system configurations in such a manner that it finally comes to a set of generalized configurations, called *basic*, in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one.

Supercompilers can have numerous applications, among which we would like to stress the following.

Programming Systems and Compilers. Consider a programming language L whose semantics are defined operationally. This means that, in some algorithmic

metalanguage M, a function $L(p, x)$ is defined such that if $p = P$ is a program in L and $x = X$ is some input data for this program, then the evaluation of this function is the application of P to X. If M is implemented, we can use L in the interpretation mode, which is, of course, very inefficient. Suppose, however, that we have a supercompiler. Given a program P, we supercompile (partially evaluate) $L(P, x)$, with p fixed at P, but x variable. The result is an efficient program in L^{sup} equivalent to P (i.e., the translation of P into L^{sup}). This can then be translated into any other language. Moreover, if the same language L^{sup} is used both at the input and at the output of the supercompiler, then we can automatically produce an efficient (compiled) compiler for L, and even a compiler generator for an arbitrary L defined in M (see [3, 6, 10, 22]). Thus a supercompiler is both a universal compiler and a metacompiler.

With a good supercompiler we can create a programming system which fixes only a metalanguage and allows the user to introduce a hierarchy of ad hoc programming languages specialized for the current problem. Each such language will have to be defined by the programmer; but it need be only an interpretive definition in some semantic metalanguage. The programmer will have to deal with a mathematical model of his objects only, and will not have to think in terms of efficiency. The supercompiler will take care of transforming semantic definitions into compilers and efficient target programs.

Design and Optimization of Algorithms. The design of algorithms, like other human activities, may include bursts of creativity, but it is mostly the use of some general rules, or methods, which could in principle be formalized and transformed into computer programs. Then why is there not much we can boast of in the computerized creation of algorithms? Our answer is: because of the absence of good supercompilers.

Indeed, suppose we have formulated such a general method, or *meta-algorithm*, in an algorithmic form. Then we have defined an algorithm $A(s, x)$ which operates over the data x, but depends also on the specification s of a particular, special situation in which we apply the general method. Formally, this is the same situation as in the case of programming languages. But we cannot say that by $A(s, x)$ we have defined a programming language for the design of algorithms, because a direct, interpretive execution of $A(S, x)$ for any specific situation $s = S$ would miss the point completely. It would not constitute the creation of a new algorithm, but simply the use of the general meta-algorithm A. However, the process of supercompilation applied to $A(S, x)$ *will* create a new algorithm specialized for the situation S. Supercompilation is creation of algorithms.

As an example, consider the following problem. Let a recursive predicate $P(x, y)$ be given. For every given $y = Y$, find an x for which $P(x, Y)$ is true. The variables x and y may stand for lists of unknown and known variables, respectively. The problem is, essentially, that of designing an algorithm to solve an equation.

The general meta-algorithm $A(s, y)$ for this problem has as its input the definition s of the predicate P and the value $y = Y$ in $P(x, Y)$. Its output should be a value X of x such that $P(X, Y)$ is true. The method is: unfold the definition of P and narrow the set of possible solutions x until you come (if you come) to some satisfactory x. A problem of this type, namely, the construction of the

algorithm for subtraction of binary numbers, was considered in [19]. For this case, $P(x, y)$ is $a + x = b$; y is the pair (a, b). It was shown in the paper cited that the general method yields, after supercompilation, the usual efficient algorithm of binary bit-by-bit subtraction.

Optimization of algorithms (which is in fact hardly different from creation) can also often be expressed in terms of very general methods, defined in the interpretation mode as modifications of the algorithmic process. A direct implementation of such a method would defeat its purpose because of overheads. But one can hope that supercompilation will eliminate overheads and lead to the same algorithm that would have been created in each specific case.

Problem Solving and Theorem Proving. A program that can cause a deep transformation of function definitions can be used as a problem-solver and a theorem-prover. Take again the algorithm of binary addition, but in a simpler setting than above. Suppose we only want to know whether there is such an x that $11011110 + x = 100010100$. Using the general algorithms for binary addition and equality, we define the predicate $P(x)$ which checks that $11011110 + x = 100010100$ and give it to the CCNY supercompiler. It transforms $P(x)$ to the very simple form which says that $P(x)$ is T if and only if $x = 110110$ (see [23]). This can certainly qualify as problem solving.

To prove the universally quantified statement $(Ax)P(x)$, we have to transform the original definition of $P(x)$ into $P(x) = T$ identically. Examples of this kind can be found in [23]. In the present paper we give an example of program transformation that can be seen as proving the theorem: $*S = S*$, where S is some string of symbols, then S consists only of asterisks *.

Error-Free Software. Proving the correctness of a program is theorem proving, so a supercompiler can be relevant. For example, if we want to check that the output of a function $F(x)$ always has the property $P(x)$, we can try to transform the function $P(F(x))$ into an identical T.

But supercompilation is relevant in an even more direct way for the problem of the generation of error-free computer code. There are two aspects to supercompilation. The first is the algorithm of driving and some service programs (such as checking the correctness of a generalization, etc.) used in the construction of the graph of states. The second is the *strategy* of supercompilation which controls the process of decision-taking with regard to the course of driving, in particular, what configurations of the computing system shall be taken as basic. The first part is quite limited in volume and is not going to change with time. It can be written, carefully debugged, and, with a reasonable degree of certainty, considered error-free. The second part, on the contrary, is going to change all the time, and grow in volume as the methods we use in supercompilation become more sophisticated. The strategy may even include an interaction with the user, with the acceptance of all the possible errors resulting from that. However, no variations or errors in the strategy of supercompilation can lead to an incorrect result (i.e., to a program that is not equivalent to the original program) if the first part, the core of the supercompiler, is correct.

Therefore, we can envisage the following programming system. The inputs of a programming job are of two kinds: (1) definitions of relevant concepts,

presumably limited in size, verifiable, and written without any concern about algorithmic efficiency; (2) very general methods, or meta-algorithms mentioned above, such as the finding of an object with required properties. The job specification then uses both kinds of entities. A supercompiler converts this specification into an exactly equivalent and efficient program.

Knowledge Bases and Expert Systems. An expert system can be represented by a function $E(q, k)$, where q is a question to the system and k is its current knowledge. The procedure of getting the answer includes examination of knowledge and logical inference. The algorithms for this kind of activity are not sophisticated if represented in an abstract way, that is, in the interpretation mode. The trouble is that for a large-scale expert system the direct interpretive programs will be too slow. It is not only the question of a big volume of knowledge. An expert system will typically involve several conceptual levels: the basic knowledge K_0; the rules K_1 of dealing with the knowledge K_0, which are referred to as a metaknowledge of the first level; the metarules K_2 for applying the rules K_1 (the metaknowledge of the second level), and so on. If at each level we use a direct interpretation, the slowdown will grow exponentially with the number of levels. Yet in order to instruct the computer in how to behave intelligently, we must use, in one form or another, interpretive definitions of information handling: interpretation, after all, is the meaning of linguistic objects. The supercompiler helps to resolve this contradiction. Write a driver $D_i(K_i, K_{i-1})$ of the ith level that applies the metaknowledge of the ith level as certain rules to handle the metaknowledge of the $i - 1$st level. Then supercompile $D_i(K_i, k)$ with the fixed rules K_i and a variable knowledge k. The result is an efficient program of knowledge-handling at the $i - 1$st level. This is essentially an automatization of what the makers of efficient expert systems do. When K_i changes, which presumably happens less frequently than changes in K_{i-1}, the driver must, of course, be resupercompiled.

Supercompilation can also be performed throughout all levels of the system. Fix $k = K$ and supercompile $E(K, q)$ with a variable q. You have an expert system that cannot learn, but answers questions very quickly. You can restore the ability of the system to learn by endowing it with two memories: short-term and long-term. To answer a question, both memories must be scanned. The long-term memory has those procedures that are obtained by using a supercompiler with the knowledge present at the moment of the last supercompilation session. The short-term memory includes universal interpretive procedures operating on the incremental knowledge received after the last supercompilation session. Such a system can be both fast and able to learn. When it has no questions to answer (the periods of "sleep"), it will execute supercompilation procedures, converting its short-range memory into the long-range one. This would be a step towards a *computer individual* (see [13]).

2. HISTORICAL AND COMPARATIVE REMARKS

Work on the supercompiler project was started by the author in Moscow in the early 1970s. Form its very inception, the project has been tied to a specific programming language, or rather metalanguage, Refal, which is defined in

Section 4. The philosophy behind the design of Refal was to have a language which would facilitate the formalization of *metasystem transitions* (i.e., transitions from a system to a metasystem) to be performed in the computer and repeated automatically as many times as necessary. The idea of a supercompiler is an outgrowth of that philosophy.

The equivalence transformations necessary for supercompilation were defined in [19]. An important aspect of the philosophy of metasystem transition is the requirement that the algorithms of supercompilation be written in the same language in which the programs to be transformed are written. If this condition is met, then, having only a supercompiler, we can automatically produce compilers for newly defined programming languages; we have only to define the semantics of the new language in Refal through the process of program interpretation. That interpreters can be automatically converted to compilers by partial evaluation was first discovered by Futamura [6]. Several years later the present author rediscovered this independently in the Refal context; we also noticed that a compiler compiler can be automatically produced in this way (see [17, 21, 22]). In the English language, the supercompiler project was first described in [20]. A systematic exposition of the project, including the definition of Refal and some programming techniques, can be found in [21].

Program transformation is a field in which a lot of work is being done currently (see a recent review [15]). All the usual arguments for the importance of program transformation systems are applicable, especially to supercompilers. We can find parallels between our method and the methods used by other researchers in this field; the work of the Edinburgh School, in particular, should be noted as very relevant (see [2]). The basic step in the process of supercompilation can be described as an application of the UNFOLD rule of Burstall and Darlington. Looping back and declaring the recurrent configuration basic is analogous to an application of the FOLD rule. Yet there are important differences between these concepts. Ershov [4] formulates partial evaluation in terms of transformation rules. We definitely want to avoid this approach. Supercompilation has its specificity, because of which presenting it as a stepwise program transformation misses an important point; it does not catch the essence of the method.

In stepwise program transformation we take a program on the input, apply to it certain transformation rules, and produce the transformed program on the output. This is the way most program transformation systems work; some have hundreds of user-supplied rules. As we have mentioned, the supercompiler works differently. At no point is the original program really transformed; the supercompiler only runs it, observes and analyzes its operation, and compiles an entirely new program. The FOLD rule applies an equation in the order inverse to its natural use in the computation process. The essence of supercompilation is in always moving in the direction of time, and never against it. This gives us a guiding principle in constructing various compilation strategies, and makes irrelevant the persistent problem of transformation systems: how to know which rules to apply and in which order to apply them. We never think in terms of combining rules (there are too many possible combinations!); we explore what actually happens in the computation process and construct a self-sufficient model of it.

In many applications of supercompilation, the function call for which an optimized program must be constructed has partially defined arguments. The idea to systematically use partial evaluation as a programming tool goes back to [12], and was further developed in [1, 3, 4, 6], and in a review paper [7]. Important work on partial evaluation in the context of programming systems is being done by N. Jones and coworkers in Denmark [10, 11].

In the strategy of supercompilation that we use there are close parallels to the concepts of delayed rules [24], lazy evaluation of Lisp programs [8], and the use of suspensions [5]. Points in common can be found between our work and the latest work of other researchers in functional programming; but this is not the place for a review.

3. BASIC DEFINITIONS FOR APPLICATIVE LANGUAGES

We now proceed with more precise definitions of the basic concepts. Our formalism is based on the concept of a computing system. Whenever we speak of functions, we have in mind computable partial functions. A function in such an approach is simply a process in the computing system dependent on some initial (input) parameters.

Suppose we have a computing system (machine) that is finite at every moment, but capable of potentially infinite expansion. It can be in different states, and we assume that there is a language to describe these states. We call this language *the basic descriptive language.* We assume further that the elements (words) of this language are strings of letters in a certain alphabet that includes, among others, two distinguished characters, the left and right parentheses "(" and ")", and that only those words are permissible in which the parentheses are properly paired. Accordingly, we call the words of the descriptive language *expressions.* The use of expressions instead of strings makes it easier to represent complex, structured objects (states of the computing system). Subsystems of a computing system can be naturally represented by subexpressions of the overall expression. For instance, the state of a computer might be described by the expression $(C)M$, where C is (represents the state of) the controlling device, and M is the memory. The subexpression C might have the form $(R_1)(R_2) \ldots (R_n)$, where R_i for $i = 1, 2, \ldots n$, are binary words representing the state of the registers.

The process of computing is a sequence of states of the computing machine, which are referred to as the *stages* of the process. A process may be deterministic or nondeterministic, finite or infinite. We distinguish *passive* and *active* states. A state is passive if, by the nature of the system, it cannot change in time. A passive state is an object, an unchangeable detail of the computing machine. An active state is capable, at least potentially, of changing in time. It stands for a process, not an object. If some stage of a process becomes passive, the process is finite, and this stage is its last stage. We can say that time stops for a process when it reaches a passive stage. While the stage is active, the process continues (even though it may infinitely repeat itself). Expressions representing passive or active states are, respectively, *passive* or *active.*

Each next stage in a process is the result of one *step* of the computing machine. If the machine is deterministic, the next stage is uniquely defined by the step function, which is a mapping from the set of active states S^a to the set of all

states S. If it is nondeterministic, its operation is defined by the step relation, which is a subset of $S^a \times S$. When we add a definition of the step function or relation to the basic descriptive language, we have an equivalent of what is known as an *applicative* language. Our passive expressions correspond to constants, active to applications.

Supercompilation is based on the analysis of computation histories in a generalized form. To do this analysis, we must extend our basic language to include means to describe generalized states of the computing system (i.e., certain sets of precise states). So the extended descriptive language will include expressions that represent *configurations* of the system. This is an important concept of the theory. A configuration is a set of precise states of the computing system or its subsystem. Not any set of states, however, but only a set that can be represented by an expression in the extended descriptive language we have chosen. The concept of a configuration is language-dependent. Consider, for instance, a subsystem D represented by one decimal digit. Its possible precise states are $0, 1, \ldots, 9$. We would probably want to have variables whose possible values are exactly digits. Let d be such a variable, and assume that variables are allowed as structural components in the extended descriptive language. Then the set of precise states $0, 1, \ldots, 9$ of D is a configuration because it can be represented by the expression d. Consider the generalized state of D that includes the states 0 and 5 only. If we have a variable in the language that has $\langle 0, 5 \rangle$ as its domain, then it is a configuration; if we have no such variable, it is not. We have to treat it as a union of the configurations 0 and 5.

The expressions of the basic descriptive language describing precise states of the machine are referred to as *ground* expressions; the expressions of the extended language, representing sets of precise states, are called *nonground*. Passive ground expressions are referred to as *object* expressions.

Supercompilation includes a metasystem transition: we have a computer system, and create a metasystem for which the original system is an object of study. Moreover, we want the metasystem to be the same computing system as the object system (i.e., an identical copy of it). Nonground and active expressions representing configurations of the object system must become objects for the metasystem. Hence we need a mapping from the set of general (including active and nonground) expressions to the set of object expressions. We call such a mapping, M, a *metacode*, provided that it satisfies these two requirements:

(1) M is homomorphic with respect to concatenation: $M(E_1E_2) = M(E_1)M(E_2)$.
(2) M is injective: $E_1 \neq E_2 \rightarrow M(E_1) \neq M(E_2)$.

The metacode of E is denoted μE. Because of (2), metacoding has an inverse operation, *demetacoding*, denoted $\mu^{-}E$.

It would be ideal if the metacode, while transforming nonground and active expressions into object expressions, did not change object expressions at all. Unfortunately, this is impossible, owing to the following simple theorem: there is no metacode that transforms all object expressions into themselves. To prove it, suppose that we have such a metacode μ. Let E represent a nonground expression. The μE is an object expression, and $\mu\mu E = \mu E$. Hence, E and μE, which are unequal, have the same image; this violates (2). Thus, any metacode,

though introduced to transform general expressions into object expressions, will have the (undesirable) effect of transforming object expressions also. The only thing we can do is to minimize the effect of metacoding on object expressions.

From now on we limit ourselves to deterministic machines. Consider a ground configuration C_1. Suppose it is active. The step function uniquely determines the next state of the system, which is another ground configuration C_2. If it is active again, the next stage follows, and so on. Thus a sequence of states $C_1, C_2, \ldots$, correponds to every precise state. If one of these is passive, we take it as the result of the computation, and the history of computation becomes finite. Otherwise it is infinite, and there is no result.

Now consider an arbitrary (nonground) configuration C_1. It defines a generalized state of our computing system. We want to construct the generalized history of computation which starts with C_1. It will not be, in the general case, linear, as for a ground configuration, but will be represented by *the graph of states and transitions*, normally infinite. Its nodes are configurations. An arc from C_i to C_j represents a transition from C_i to C_j; it carries the condition under which this transition occurs.

Supercompilation transforms an infinite graph of states into a finite one. This is achieved by looping back at certain points in the construction of the graph of states, and declaring some configurations *basic*. An easy case of looping back is when a configuration C_j is met which is a specialization of one of the previous configurations C_i (we say that C_j is a specialization of C_i if the set of precise states represented by C_j is a subset of that of C_i). This is the case of C_4 and C_2^2 in Figure 1. When this happens, we reduce C_j to C_i, which is shown by a dashed line in Figure 1. Unlike transitions between states, which are shown by solid lines, the reduction arcs do not stand for a real step in the computing system, but only for a change in the way the state is represented.

The above is not the only case when we have to loop back. The other case is exemplified in Figure 1 by C_5 and C_2^3. Although the former is not a specialization of the latter, we can decide that it is "close enough" to it to loop back; or, rather, it is "too close" to go on and be sure that the graph will be finite. Hence, we generalize C_2^3 and C_5, that is, construct another configuration, C_6, such that both C_2^3 and C_5 are its specializations. Then we discard the subgraph originated from C_2^3, reduce C_2^3 to C_6, and go on with supercompilation from the node C_6.

A graph of states is *self-sufficient* if every configuration in it is either passive or reduced to another configuration and there are no circuits composed of reduction arcs only. The configurations found in a self-sufficient graph are *basic*. Such a graph is essentially a program applicable to all precise states covered by the basic configurations. A set of basic configurations (*basis*) can be communicated to the supercompiler at the outset, or it can be defined, in part or in full, in the process of supercompilation itself.

The *strategy* of supercompilation is the algorithm that at every moment in the construction of the graph of states decides which of the following actions must be taken (this does not imply that *any* of these actions can be taken at every moment).

(1) Reduce some configuration to one of the basic configurations.
(2) Go on constructing the subgraph for some configuration by applying to it the step function of the computing system.

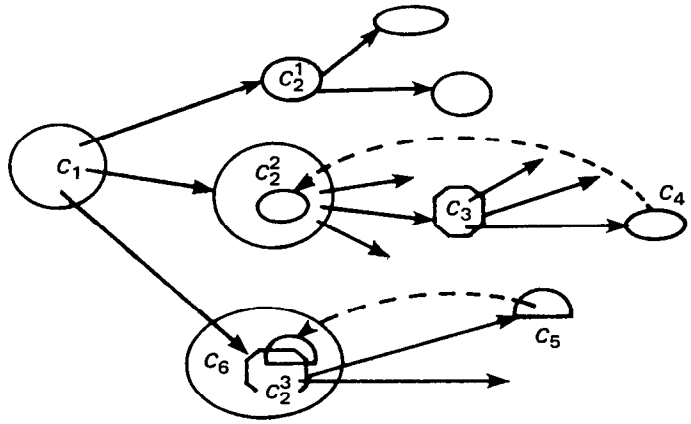

Fig. 1. Graph of states and transitions.

(3) Declare some configuration basic.
(4) Reduce one configuration to another.
(5) Generalize two configurations.
(6) Stop and output the graph.

When designing a strategy for supercompilation, the following questions require immediate answers: In what order should we consider configurations? Depth first or width first? Should we try to loop back after every step producing a new configuration? For a given configuration, what previous configurations must be considered as possible looping targets? Specifically, should we try only the ancestors, or must the configurations on the parallel branches also be included? How can we guarantee that the strategy leads to a finite graph?

As we go into the details of the supercompilation process, dealing with an aribtrary computing system becomes progressively more awkward; and there is always the need for examples. Thus, from the next section on, we start using a specific applicative language, Refal.

4. THE LANGUAGE REFAL

Definition. In the present paper we define and use Refal in a mathematical-style syntax; the actual format required by the existing implementations is of no concern here.

The elementary syntax units of Refal are of two kinds: special signs and object symbols (or just *symbols*). The special signs include:

—structure brackets "(" and ")";
—activation brackets "⟨" and "⟩";
—free variables, which are represented by subscripted "s" (a symbol variable) or "e" (an expression variable), for example, s_1, s_x, e_5.

The object symbols used in Refal are supposed to belong to a finite alphabet, which may, however, vary from one use of Refal to another. We use the following as object signs: characters distinct from the special signs, subscripted and superscripted capital letters, Algol identifiers (sometimes underlined so as to

stand out). In computer implementation, we also allow the use of whole numbers as object signs (in which case, of course, the alphabet of object signs becomes infinite).

We use capital italic letters $A, B, \ldots E, E_1, \ldots$, etc., as metasymbols to denote Refal objects. We agree that the subscripts i, j, k of variables, as in e_i, may mean *some* subscripts, when this is clear from the context.

A Refal *expression* is one of the following:

—an empty string, which we may represent by nothing or by the metasymbol empty;
—a symbol (i.e., an object symbol, not a special sign);
—a variable;
—E_1E_2, or (E_1), or $\langle E_1 \rangle$, where E_1 and E_2 are expressions.

An expression that is a symbol, a variable, (E_1), or $\langle E_1 \rangle$ is referred to as a *term*. An expression is *passive* if it does not include activation brackets; it is *active* otherwise. An expression without free variables is a *ground* expression; otherwise it is a *nonground* expression. A *pattern* is a passive, and generally nonground, expresssion. A passive ground expression is an *object* expression.

An *L-expression* is a pattern which (a) contains no more than one occurrence of every expression variable (we say *e*-variable for short), and (b) contains no more than one *e*-variable on the top level in every subexpression (i.e., none of its subexpression can be represented as $E_1e_iE_2e_jE_3$, where E_1, E_2, E_3 are some expressions). Examples of *L*-expressions follow:

$$Ae_1, \quad BC(DE), e_1 + (e_2)(e_3), \quad s_1e_xs_1, \quad (e_x)ABCe_y.$$

Examples of pattern expressions that are not *L*-expressions are the following:

$$(e_x)ABCe_x, \quad e_1 + e_2, \quad e_1s_2((e_1 + e_3)).$$

A Refal sentence has the form $\langle L \rangle = R$, where L is an *L*-expression and R is an arbitrary general expression. The equality sign is just a symbol (not a special sign) used for visual convenience. The right side R can include only those variables that appear in the left side $\langle L \rangle$. The pattern L usually starts with a symbol that is referred to as the function symbol. A Refal program is a (ordered) list of sentences.

Given an object expression E and an *L*-expression L, the *matching operation* $E:L$ is defined as finding such a substitution S for the variables in L that applying S to L yields E. The values assigned to *s*-variables in substitutions must be single symbols, while *e*-variables can take any expressins as their values. If there is such a substitution, we say that the matching succeeds, and E is *recognized* as (a special case of) L. If there is no such substitution, the matching fails.

Let E be an object expression and L an *L*-expression. The following algorithm performs the matching operation $E:L$ and shows that if there is a substitution transforming L into E, it is unique (this holds as long as L is an *L*-expression, but may not hold for other patterns). We refer to E as the *target* and L as the *pattern* in matching. The pair $E:L$ itself is referred to as a *clash*. Substitutions for the variables in L are *assignments*; they are written in the form $A \leftarrow V$, where V is a variable and A an object expression. A list of assignments is a *partial*

environment, PENV; if the list includes assignments for all the variables in L, it is a *total* environment.

The Matching Algorithm

begin
Set PENV empty; set STACK-OF-CLASHES to include one member $E:L$.
NEXT-CLASH:
Take CLASH from STACK-OF-CLASHES.
CLASH-LOOP: Use the rule for any of the applicable cases.
case 1. CLASH is *empty*:*empty*. Go to END-CLASH.
case 2. CLASH is $E:e_i$. Add $E \leftarrow e_i$ to PENV. Go to END-CLASH.
case 3. CLASH is $SE:SL$ or $ES:LS$, where S is a symbol; put CLASH = $E:L$; go to CLASH-LOOP.
case 4a. CLASH is $SE:s_iL$ or $ES:Ls_i$, and there is no assignment for s_i in PENV. Add $S \leftarrow s_i$ to PENV; put CLASH = $E:L$; go to CLASH-LOOP.
case 4b. CLASH is $SE:s_i$L or $ES:Ls_i$, and there is the assignment $S \leftarrow s_i$ in PENV. Put CLASH = $E:L$; go to CLASH-LOOP.
case 5. CLASH is $(E_2)E_1:(L_2)L_1$ or $E_1(E_2):L_1(L_2)$. Add $(E_2:L_2)$ to STACK-OF-CLASHES; put CLASH = $E_1:L_1$; go to CLASH-LOOP.
case 6. If none of the above is applicable, the matching fails (recognition impossible).
END-CLASH: If STACK-OF-CLASHES is empty, the matching succeeds, and PENV is the full environment. Otherwise go to NEXT-CLASH.
end of the Matching Algorithm.

The semantics of Refal are defined operationally by the *Refal machine* which executes algorithms written in Refal. The Refal machine has two potentially infinite information storages: the *program-field* and the *view-field*. The program field contains a Refal program, which is loaded into the machine before the run and does not change during the run. The view-field contains a ground expression which changes in time as the machine works; this expression is often referred to simply as the view-field.

The Refal machine works by *steps*. Each step is executed as follows. If the expression in the view-field is passive, the Refal machine comes to a normal stop. Otherwise it picks up one of the pairs of activation brackets in the view-field and declares the term it delimits the *leading* active term. It then compares the leading term, say $\langle E \rangle$, with the consecutive sentences in the program field, starting with the first one, in search of an *applicable* sentence. A sentence is applicable if E can be recognized as the pattern L in its left side (i.e., the matching $E:L$ is successful). On finding the first applicable sentence, the Refal machine copies its right side and applies to it the substitution resulting from the matching $E:L$. The ground expression thus formed is then substituted for the leading active term in the view-field. This ends the execution of the current step, and the machine proceeds to execute the next step. If there is no applicable sentence, the Refal machine comes to an abnormal stop.

The definition of the leading active term may vary, so that we can have several variants of the Refal machine. Originally, the Refal machine was defined as evaluating subexpressions according to the rule "inside-out, from left to right" (known as applicative order). Then the leading active term is defined as the leftmost active term $\langle E \rangle$ with a passive E. This is also the way it works in the existing implementations.

However, an outside-in left-to-right Refal machine can also be used (the normal evaluation order). It will start by trying to apply the program sentences to the outermost activation brackets, first on the left. Then the expression E in the leading active term $\langle E \rangle$ is not, generally, an object expression, but may include some activation brackets. The matching process $E:L$ must then be generalized as follows. We say that the active subexpression $\langle E_1 \rangle$ of E *does not prevent* the matching $E:L$ if the substitution of any expression for $\langle E_1 \rangle$ has no effect on the success or failure of the matching; otherwise $\langle E_1 \rangle$ *prevents* the matching. If the current matching is not prevented by some active subexpression, the outside-in Refal machine goes on and may complete a step with some values in the substitution resulting from the successful matching being active. If it finds that some subexpression prevents the matching, this subexpression becomes the next attempted leading active term.

We can also construct a Refal machine with many step-executing processors. Such a machine will attach one processor to every activation brackets pair in the view-field. Parallel activations as in $(\langle E_1 \rangle)\langle E_2 \rangle$ will be executed in parallel. As for nested activations, in the situation of prevention, the outer activation will wait until the preventing configuration is—partially or completely—computed.

If the inside-out evaluation process is finite, the outside-in process will also be finite and yield exactly the same result. It may happen, however, that the inside-out evaluation never stops, while the outside-in evaluation results in a finite process (this situation is well known from the lambda calculus). The order of parallel activations does not effect the results. So we take as the basic Refal machine the inside-out left-to-right kind. This is referred to simply as the Refal machine. If a computation process is finite with this machine, all other kinds will produce the same result. But we can also write a Refal program meant specifically for outside-in execution.

A function is defined by specifying (a) a general Refal expression F called the *format* of the function and (b) a Refal program which is its *definition*. Substituting some values for the variables in F, we put it in the view-field of the Refal machine which is loaded with the definition. If after a finite number of steps the Refal machine comes to a normal stop, the resulting object expression in the view-field is the value of the function.

Examples. In the unary number system, zero is represented by 0, one by 01, two by 011, and so on. We want to define the function of addition for these numbers. Let the format be $\langle +(e_x)e_y \rangle$. Then the definition is

$$\langle +(e_x)0 \rangle = e_x$$

$$\langle +(e_x)e_y 1 \rangle = \langle +(e_x)e_y \rangle 1$$

With the input values 01 for e_x and 011 for e_y, the Refal machine will exhibit the following computation process: $\langle +(01)011 \rangle$, $\langle +(01)01 \rangle$ 1, $\langle +(01)0 \rangle$ 11, 0111. We could have chosen a different format (e.g., $\langle +(e_1)(e_2) \rangle$ or $\langle$add e_x, $e_y \rangle$, etc.)

The function reversing a string of symbols can be defined as follows:

$$\langle \text{reverse } s_1 e_2 \rangle = \langle \text{reverse } e_2 \rangle s_1$$

$$\langle \text{reverse} \rangle =$$

As an example of the use of nested activation brackets in the right side, we define the adding machine for binary numbers:

$$\langle \text{addb}(e_x 0) e_y s_1 \rangle = \langle \text{addb}(e_x) e_y \rangle s_1$$

$$\langle \text{addb}(e_x 1) e_y 0 \rangle = \langle \text{addb}(e_x) e_y \rangle 1$$

$$\langle \text{addb}(e_x 1) e_y 1 \rangle = \langle \text{addb}(\langle \text{addb}(e_x) 1 \rangle) e_y \rangle 0$$

$$\langle \text{addb}(e_x) e_y \rangle = e_x e_y$$

The format is $\langle \text{addb}(e_1) e_2 \rangle$. Note that the variables we choose for formats have nothing to do with the variables used in programs. Also, the variables in different sentences are in no way related (though we usually keep the same variables as a matter of convenience). The last sentence of the program for *addb* may not be understood immediately. It will work correctly because it will be used only in the situation where at least one of the two arguments e_x and e_y is empty. The program would be more readable if instead of that sentence we used these two:

$$\langle \text{addb}(e_x) \rangle = e_x$$

$$\langle \text{addb}(\ ) e_y \rangle = e_y$$

The language we define above is referred to as the *strict* Refal. It is the basis for equivalent transformation and automatic generation of programs. For convenience of programming, however, we introduce some natural extensions of the strict language. The interpretive implementation of Refal allows the extensions, but the supercompiler requires strict Refal on the input. A special Refal program translates programs written in extended Refal into strict Refal.

The first step to extend strict Refal is to remove the restriction on the left sides of sentences. This version of the language is referred to as *basic* Refal. It allows any pattern expressions in the left sides of sentences, not just L-expressions. When the pattern P in the matching $E:P$ is not an L-expression, there may be more than one substitution transforming P to E. So a rule is necessary that would tell us which of the substitutions must be used. Our rule corresponds to matching from left to right: of all substitutions the one chosen is that which assigns the least (with respect to the number of constitutent terms) value to the leftmost e-variable; if this does not eliminate ambiguity, the same selection is made for the second e-variable from the left, and so on.

Using arbitrary patterns, we can define the function *chpm* that changes every "+" into "−" in a string, as follows:

$$\langle \text{chpm } e_1 + e_2 \rangle = e_1 - \langle \text{chpm } e_2 \rangle$$

$$\langle \text{chpm } e_1 \rangle = e_1$$

When the argument E is successfully matched against $e_1 + e_2$, the character "+" in the pattern is associated with the first "+" from the left in E. So we can take e_1 out of the activation brackets and apply *chpm* recursively to e_2. To define this

function in strict Refal, we need three sentences:

$$\langle \text{chpm} + e_1 \rangle = -\langle \text{chpm } e_1 \rangle$$
$$\langle \text{chpm } s_2 e_1 \rangle = s_2 \langle \text{chpm } e_1 \rangle$$
$$\langle \text{chpm} \rangle =$$

The metacode transformation we use in Refal singles out one symbol, let it be the asterisk "*", to build up the images of variables and activation brackets. The metacodes of s_i and e_i are *Si and *Ei, respectively. The pair of activation brackets $\langle\ \rangle$ becomes *( ). Parentheses and symbols distinct from * remain as they are, while * becomes *V. For instance, the active nonground expression $\langle Fe_1(\langle Gs_x \rangle)\rangle$ becomes, when metacoded, the object expression *$(F{*}E1(*(G{*}SX)))$.

Discussion. Although almost any program can be written in almost any language, and evaluation of languages is very subjective, we believe that Refal is the best choice for the supercompiler project. Supercompilation deals with generalized histories of computation. Refal's definition through a mathematical machine, the mode of operation of this machine, Refal's use of pattern expressions—all this is essential for successful supercompilation. Also, it is a historical fact that the very concept of supercompilation was suggested by the use of Refal.

When compared with Lisp, Refal is different in data structure, in treatment of variables, and in the use of patterns. In Lisp there is only one way to construct data: forming a binary tree. A list is only a special binary tree written in a special notation. When we write something like (sum 16 25 36) in Lisp, we actually mean what in Refal would be represented as (sum(16(25(36( ))))). But we also have in Refal a simple concatenation of objects: sum 16 25 36, which can, unlike Lisp's lists, be processed in both directions. The vertical and horizontal dimensions of Refal trees can be varied independently.

While the data structure in Refal is more sophisticated than in Lisp, the use of variables is much simplified, which makes it easier to analyze the computation process. There are no global variables in Refal; neither are there variables which stay valid over a whole function definition. Free variables in Refal can be described as local to sentences. In a sense they are not variables at all, but simply details of the Refal machine used to express patterns. When we write, say, $ABCe_xZ$, e_x is a physical object, which occupies a certain geometric place in an expression and is used according to certain rules in pattern matching. This suggests the concepts of a configuration and its generalizations.

The use of patterns is crucial for supercompilation. It also makes programs shorter and more readable, as is well known. Take the textbook example of the function ISECT that computes the intersection of two sets represented by lists. As written in Lisp it is

```
(DEFINE (MEMBER A SET)
        (COND ((NULL SET) NIL)
          ((EQUAL A (CAR SET)) T)
          (T (MEMBER A (CDR SET))) ))
(DEFINE (ISECT SET1 SET2)
        (COND ((NULL SET1) NIL)
          ((MEMBER (CAR SET1) SET2) (CONS (CAR SET1)
            (ISECT (CDR SET1) (SET2)))
          (T (ISECT (CDR SET1) SET2)) ))
```

The same function defined in Refal:

$$\begin{aligned}\langle \text{ISECT}(\ )(e_2)\rangle &= \\ \langle \text{ISECT}(s_1e_2)(e_3s_1e_4)\rangle &= s_1\langle \text{ISECT}(e_2)(e_3e_4)\rangle \\ \langle \text{ISECT}(s_1e_2)(e_3)\rangle &= \langle \text{ISECT}(e_2)(e_3)\rangle\end{aligned}$$

It is worth mentioning that, as a rule, Lisp programmers habitually use the PROG feature, which is rather a retreat from the functional philosophy of the language. Although it is easy to define an analogue of the PROG feature in Refal, one does not really want to. Programming in Refal is strict functional programming.

Prolog, like Refal, takes pattern matching as the main operation of the language. In many cases, a Prolog program will resemble the corresponding Refal program. For instance, to trim leading blanks in a list using Prolog, we could write the axioms:

$$\begin{aligned}&\text{trim}(' '.L,T) \leftarrow \text{trim}(L,\ T).\\ &\text{trim}(L,\ L)\end{aligned}$$

and set the goal trim(L, T), where L is the list to be trimmed and T is the variable to which the result must be assigned. To do the same in Refal, we write the sentences:

$$\begin{aligned}\langle \text{trim}\ '\ 'e_1\rangle &= \langle \text{trim}\ e_1\rangle \\ \langle \text{trim}\ e_1\rangle &= e_1\end{aligned}$$

and put $\langle$trim $L\rangle$ into the view-field.

The differences between Refal and Prolog are of two kinds. First, Prolog uses the same data structures as Lisp (i.e., binary trees). Second, there are significant differences in the process of matching and its use. Expressions matched in Refal describe the state of affairs—the stages of certain processes. Unlike patterns, they include no free variables. The results of matching are always unique, and after a successful matching the function call is replaced by the right side of the sentence used. After the replacement has taken place, there is no way back. A Refal program applied to a function call describes a *process*, that is, a sequence of completely defined stages. In Prolog, the expressions matched are goals that describe desired properties and relations, not the situations already achieved. The results of matching are not, generally, final: we may have to return and match a goal to the "head" of another axiom. Both parties to matching may include variables (cross-binding). The operation of the Prolog machine is not a linear process, but a walk around a tree with repeated backtrackings. It is much more complicated and difficult to grasp than the operation of the Refal machine. The way a set of Prolog axioms works, even in very simple cases, is often far from obvious. Consider the function *trim* above. When writing and mentally testing it, one is likely to apply *trim* to some argument as an isolated goal. One will be convinced that it works correctly. And it will come as a surprise that, in a conjunction with another goal, one may get a wrong answer (see [18]).

Prolog is very successful as the language of a rather intelligent database. In our view, however, Prolog is not at its best when used to define *algorithms* proper. Take the textbook example below [14]. The program of formal differentiation in

Prolog starts as follows:

$d(X, X, 1)$
$d(C, X, 0) \leftarrow \text{atomic}(C).$
$d(U + V, X, A + B) \leftarrow d(U, X, A)\ \&\ d(V, X, B).$
$d(U - V, X, A - B) \leftarrow d(U, X, A)\ \&\ d(V, X, B).$
... etc.

Compare it with the analogous program in Refal:

$\langle Ds_x/s_x \rangle = 1$
$\langle Ds_c/s_x \rangle = 0$
$\langle De_u + e_v/s_x \rangle = \langle De_u/s_x \rangle + \langle De_v/s_x \rangle$
$\langle De_u - e_v/s_x \rangle = \langle De_u/s_x \rangle - \langle De_v/s_x \rangle$
... etc.

We think of D as a *procedure* of differentiation. In Refal, we simply list the possible forms of the argument and write out the answers, as in a calculus textbook. Using Prolog, we have to take two additional steps in order to write the program. First we translate the procedure of differentiation into a relation between the input and output; then we interpret the list of relations as a procedure. Both steps are absolutely unnecessary. One of the results of this doing and undoing is the introduction of auxilliary variables, which we have no need for when programming in Refal.

Since, in a supercompilation, we do an analysis of the algorithmic processes in a computer, Prolog is hardly the best choice.

A number of functional programming languages have been proposed during the last few years, for example, OBJ2, Hope, KRC, and others. New functional languages keep appearing. They are similar to Refal in many respects; sometimes they are *very* similar. For instance, the language used in [16] is essentially a subset of Refal. To the best of the author's knowledge, Refal, implemented efficiently in 1968, was the first in this family of languages. It is on the simple side of the spectrum, which is dictated by its use for program transformation.

5. DRIVING

The operation of the Refal machine can conveniently be described in terms of elementary operations that are essentially certain types of substitutions. We introduce two kinds of substitutional operations: *assignments* and *contractions*.

An assignment is represented as $E \leftarrow V$, where E is an expression and V is a variable. (Here, and in the following, it should be clear from the context whether a letter is used as a metasymbol for a Refal expression or just as a Refal symbol). The execution of this assignment results in the association of the value E with the variable V. To apply an assignment as a substitution to an expression, it must be put on the left side: $(E \leftarrow V)E_1$; this stands for the result of replacing every occurrence of V in E_1 by E.

A contraction is represented as $V \rightarrow L$, where V is a variable and L is an L-expression. If the current value of V is an object expression E^o, then the execution of the contraction is the matching $E^o : L$. If this matching fails, we say that the contraction cannot be applied to V. If it succeeds, the resulting total environment contains an assignment for every variable in L, and we interpret these assignments as giving the new values of these variables. After the execution

of the assignment, the contracted variable V becomes undefined, unless it also appears in the right side of the contraction L. So the contraction can be read as "break down the value of V according to the pattern L." For instance, if the current value of e_1 is $AB(X + Y)A$, then the execution of the contraction $e_1 \rightarrow s_x e_1 s_x$ will succeed and result in the value A for s_x and the new value $B(X + Y)$ for e_1. To apply a contraction as a substitution, we put it on the right of the expression, so $E_1(V \rightarrow E) = (E \leftarrow V)E_1$.

Our notation, though unusual, is consistent and natural. It implements the following two principles: (a) when the formal object is seen as a substitution, the arrow is directed from the variable to its replacement, and the variable stands close to the transformed expression; (b) when it is seen as an operation in an environment, the old variables, the values of which are defined, are on the left, while the new variables being defined are on the right.

As is well known, the effect of simultaneous substitutions is generally different from that of their sequential execution. Let $V_1, \ldots, V_n$ be the free variables of a configuration C. Then $(V_1) \ldots (V_n)$ is referred to as the *varlist* of C, and denoted as *var* C. When we deal with simultaneous contractions or assignments, it is convenient to deal with one object, the varlist, instead of sets of variables. Suppose we have a set of simultaneous contractions $(v_1 \rightarrow L_1) \ldots (v_k \rightarrow L_k)$, where v_1 to v_k are some variables from C. Take *var* C and apply all the contractions to it. The result, L, may not be an L-expression only, if some of the L_i's in the contractions had used the same e-variable. In such a case we rename the variables in the L_i's to avoid conflicts. So we assume that L is an L-expression. It gives a full account of the contractions applied, as well as of variables *not* affected by the contractions. If, for example, the varlist is $(e_1)(s_2)(e_3)$ and the contractions are $(s_2 \rightarrow A)(e_3 \rightarrow Be_3)$, then L is $(e_1)(A)(Be_3)$, which reminds us that there is also the variable e_1 in the varlist that was not affected by contractions. Such *list contractions* are represented as $V \rightarrow L$. In our example:

$$(e_1)(s_2)(e_3) \rightarrow (e_1)(A)(Be_3).$$

We also write assignments in the full form: $E \leftarrow V$, e.g.,

$$(e_1)(B)(e_2 + ABCe_1) \leftarrow (e_1)(s_2)(e_3).$$

We often treat varlists as unordered sets. We write $V_1 \leq V_2$ to mean that every variable from V_1 is also in V_2. If $V_1 \leq V_2$ and $V_2 \leq V_1$, we say that V_1 and V_2 are equal as sets. At the same time, we must remember that a varlist is a definite Refal expression, and when taken in isolation its terms cannot be reordered.

When only one variable from the full varlist, say e_1, is affected by a contraction, we may represent it by a single contraction term $(e_1 \rightarrow L_1)$. We can then find that we want a composition of several such terms. In fact, this is exactly how the generalized matching algorithm, to be discussed shortly, works. One should keep in mind, however, that the meaning of an individual contraction may depend on the full list of variables. Take the contraction $e_1 \rightarrow s_x e_1$, for instance. If s_x is not in the varlist, then this contraction succeeds whenever the value of e_1 starts with any symbol; s_x takes on this symbol as its value. If s_x is in the varlist, then our individual contraction is actually a part of the contraction $(e_1)(s_x) \rightarrow (s_x e_1)(s_x)$. For it to succeed, the value of e_1 must start with the symbol that is the current value of s_x, not just any symbol.

In full contractions, we can rename the variables on the right side in any (consistent) way; the meaning of the operation will not be changed. For example, instead of the contraction above, we could write $(e_1)(s_x) \rightarrow (s_y e_1)(s_y)$. The only difference would be that what we called s_x before is now called s_y. When we have an individual contraction, we must make it clear, with respect to every variable in the right side, whether the variable is *old* (i.e., belongs to the current varlist) or *new* (i.e., was not used before). Repeated e-variables are not allowed in L-expressions. So we agree, in order to avoid unnecessary renamings, that an e-variable with a subscript already used in the varlist can be used in the right side of contractions only in the contraction itself (and only once, of course). Since in such a use the variable is redefined and not compared with another value, we do not call it an old variable: e-variables cannot be old. It is only s-variables that must be categorized in individual contractions as new or old.

The rule for the composition (*folding*) of full contractions follows from our definitions:

$$\text{If } V_2 \leq \mathit{var}\ L_1, \text{ then } (V_1 \rightarrow L_1)(V_2 \rightarrow L_2) = (V_1 \rightarrow L_1(V_2 \rightarrow L_2)).$$

If we have a contraction $V \rightarrow L$, the variables in L are said to be the *derivatives* of the variables in V. Our use of contractions is such that a contraction never appears for a variable that is not a derivative of the preceding varlist; such a situation would be senseless.

The rule for folding assignments is

$$\text{If } V_2 \leq \mathit{var}\ E_1, \text{ then } (E_2 \leftarrow V_2)(E_1 \leftarrow V_1) = ((E_2 \leftarrow V_2)E_1 \leftarrow V_1).$$

Now consider the sequence $(E \leftarrow V)(V \rightarrow L)$. It represents a situation when the varlist V is assigned the value E, after which we ask that it be restructured according to the pattern L. The assignment *clashes* with the contraction. To resolve the clash we must match $E:L$. In fact, the matching operation is the only operation we use; contractions and assignments, for individual variables and for varlists, are only special cases.

If the target E in the clash $E:L$ is an object expression, its resolution is given by the matching algorithm which is part of the definition of the Refal machine. Now we are interested in a situation where E may be a nonground, although still passive, expression. Thus both operands in the clash represent sets of ground (object) expressions. A ground expression E^g is an element of the set represented by E iff the matching $E^g:E$ succeeds. The union of nonground expressions E_1 and E_2 considered as sets is represented as the sum $E_1 + E_2$; their intersection as $E_1{}^*E_2$. For the matching of two nonground expressions, the following formula holds:

$$E:L = \textstyle\sum_k (\mathit{var}\ E \rightarrow L^k)(E^k \leftarrow \mathit{var}\ L), \quad 1 \leq k \leq N, \tag{1}$$

where the left side is a clash and the right side its *resolution*. For every additive term in the resolution, $\mathit{var}\ E^k$ is equal (as a set) to $\mathit{var}\ L^k$, and

$$E(\mathit{var}\ E \rightarrow L^k) = (E^k \leftarrow \mathit{var}\ L)L, \qquad 1 \leq k \leq N \tag{2}$$

is a subset of the intersection E^*L. These subsets are disjoint, and their sum in (1) is the full intersection E^*L. The algorithm to resolve a given clash follows.

The Generalized Matching Algorithm, GMA. Let the clash be $E:L$. Let $W_e = var\ E$ and $W_l = var\ L$. In the following, E and L will be used as variables, but W_e and W_l are fixed, referring to the initial values of E and L.

A *partial resolution term*, PRT, is a list contraction followed by assignments for a subset W_l' of W_l. A CPRT (current Clash and PRT) is a clash and a PRT. The GMA is operating on a sum of CPRTs, referred to as STATE. Every term in STATE is processed independently. In the processing, a term may be eliminated, or give rise to more than one term. A term can be *closed*, which means that the clash in it disappears (being resolved), and the partial resolution becomes complete. In the end, the closed CPRTs (which become PRTs after closing) make up the sum in (1). If no terms are left, ($k = 0$), E^*L is empty. We denote this result of the resolution as Z. It is the unity of the summing operation: $X + Z = X$.

The *update* of the CPRT $(E:L)(W_e \to L^k)(E^k \leftarrow W_l')$ by the PRT $(var\ L^k \to L^i)(E^i \leftarrow W_l'')$ is the result of the following transformation of the CPRT:

1. Replace E by $E(var\ L^k \to L^i)$;
2. Replace E^k by $E^k\ (var\ L^k \to L^i)$;
3. Replace L^k by $L^k\ (var\ L^k \to L^i)$;
4. Add $(E^i \leftarrow W_l'')$ to $(E^k \leftarrow W_l')$.

To update a CPRT by a sum of PRTs, we take one copy of the CPRT for each PRT, update it and sum the results.

An *internal s-clash* is a clash $S:S'$, where S and S' are either specific symbols or s-variables from the same varlist $var\ L^k$ of a CPRT. It is resolved according to these rules, where *id* is the identity contraction and A is an arbitrary symbol:

1. $S:S = id$
2. $s_i:S = (s_i \to S)$
3. $A:s_i = (s_i \to A)$
4. If none of the above, Z.

The main procedure follows:

begin
Put STATE = $(E:L)$ *id*.
Until all terms in STATE are closed, do:
begin Pick any of the CPRTs in STATE. Let C be the clash in CPRT, and PRT the partial resolution term. Use any applicable rule of the following:

case 1. C is *empty*:*empty*. Close CPRT (by eliminating C).
case 2. C is $E:e_i$. Update CPRT by $E \leftarrow e_i$ and close it.
case 3. C is $S'E:SL$, or $ES':LS$. Here and in the following, S is either a symbol or a symbol variable and so is S'. If S is a variable for which there is no assignment in CPRT, put CPRT equal to $(E:L)$ PRT and update it by $S' \leftarrow S$. If S is a symbol, say A, or a variable whose assignment value in PRT is A, resolve the internal clash $S':A$, let the resolution be R, put CPRT = $(E:L)$PRT, and update it by R.
case 4. C is $(E_1)E_2:(L_1)L_2$ or $E_2(E_1):L_2(L_1)$. Using the GMA recursively, resolve the clash $E_1:L_1$ starting with STATE = $(E_1:L_1)$PRT, and let the result be R. Put CPRT = $(E_2:L_2)$PRT and update it by R.
case 5L. C is $e_jE:SL$. Update CPRT by $(e_j \to) + (e_j \to s_{j'}e_j)$. Here and in the following, j' stands for a *new* variable index (i.e., one that has not yet been used in *var* E or its derivatives).

case 5R. C is $Ee_j{:}LS$. Update CPRT by $(e_j \rightarrow) + (e_j \rightarrow e_j s_{j'})$.
case 6L. C is $e_j E{:}(L_1)L$. Update CPRT by $(e_j \rightarrow) + (e_j \rightarrow (e_{j'})e_j)$.
case 6R. C is $Ee_j{:}L(L_1)$. Update CPRT by $(e_j \rightarrow) + (e_j \rightarrow e_j(e_{j'}))$.
case 7. If none of the above is applicable, the current CPRT is eliminated.

end processing terms.
Collect all closed resolution terms as the output. If none, output Z (matching impossible).
end of the algorithm.

It is easy to prove that the contractions $var\ E \rightarrow L^k$ in (1) are pairwise incompatible (i.e., the intersection $L^{k*}L^{k'}$ for k not equal k' is empty). The idea: all the branchings in cases 5L to 6R are such that one branch produces object expressions that have at least one term more than those produced by the other branch in comparable subexpressions.

The GMA is a generalization of the well-known concept of *unification* in term-rewriting systems. The data structure in Refal is more general than the structure of terms formed by constructors. When we limit Refal expressions to that subset, the GMA is reduced to unification.

To construct the graph of states and transitions of the Refal machine, we use *driving*. The idea of driving is to execute one or more steps of the Refal machine in the situation where the contents of the view-field are not completely defined, but are described by a nonground configuration which includes unknown subexpressions represented by free variables. The Refal machine is not meant to deal with free variables in the view-field; we "drive" free variables forcefully through the sentences of the program.

Let the leading active subexpression in C_1 be $\langle FE \rangle$. Let the sentences for F be

$$\langle FL_1 \rangle = R_1$$

$$\ldots$$

$$\langle FL_n \rangle = R_n$$

For those values of the free variables in E with which E matches L_1, the Refal machine will use the first sentence. To find this subset, we resolve, using the GMA, the clash:

$$E{:}L_1 = \sum_k (var\ E \rightarrow L_1^k)(E_1^k \leftarrow var\ L_1), \quad 1 \le k \le N_1.$$

Under each contraction in the sum, the Refal machine will take the first sentence and replace the expression under concretization by $(E_1^k \leftarrow var\ L_1)R_1$, because the assignment part of the resolution gives us the values to be taken by the variables in L_1 in the process of matching. It is only the variables from L_1 that are allowed to be used in R_1, hence after the substitution we have an expression which depends only on the variables in E and its derivatives.

Thus the first part of the graph of states for C_1, corresponding to the first sentence in the definition of F, will consist of N_1 branches:

$$(var\ E \rightarrow L_1^k)C_2^k, \quad 1 \le k \le N_1,$$

where at the end of each branch we have the new configuration:

$$C_2^k = (E_1^k \leftarrow var\ L_1)R_1^k.$$

For those members of the initial configuration C_1 which do not belong to any of the subclasses we separated, the first sentence will be found unapplicable. The Refal machine will then try to apply the second sentence, which we should take into account by separating another group of subclasses of E and adding it to the first group. Repeating this procedure for each sentence in the definition of F, and renumbering the contractions throughout the whole set, we come to the graph of states which can be represented by the expression:

$$C_1((var\ E \rightarrow L^1)C_2^1 + (var\ E \rightarrow L^2)C_2^2 + \ldots + (var\ E \rightarrow L^N)C_2^N)$$

where $N = N_1 + N_2 + \ldots + N_n$. The parenthesized sum of branches following C_1 is referred to as the *development* of the configuration C_1, and denoted as *dev* C_1. There is an obvious optimization that can be applied to the construction of the graph of states. If, for the ith sentence, the argument E is found to match L_i without contractions, which means that E is a subset of L_i, then the branches originating from all the sentences starting from the $i + 1$st can be omitted because they will never be used.

The graph of states has a double significance. First, it is a history of computation, and we can use it in this role for analysis of algorithms and equivalent transformation of functions. Second, since it is a *generalized* history, it is a ready program to execute one or more steps of the algorithmic processes described by the initial configuration of the graph. Indeed, let the values of the variables in C_1 be given. Then we can apply the contractions on the branches to these values, and use the first applicable branch to make the step from C_1 to C_2. If we have the graph of states for C_2, we can make one more step, and so on. The ordering of the arcs in the graph of states is important. The groups of branches originating from different sentences must be ordered in the same way as the sentences in the original definitions. The ordering of branches within groups, though, is immaterial, because the corresponding contractions are, as we know, incompatible.

If the initial configuration C_1 is $\langle Fe_1 \rangle$, the resulting graph will have exactly one branch, $(e_i \rightarrow L_i)R_i$, for each sentence, where L_i is the left and R_i the right side of the sentence. Thus, the Refal program is nothing else but the collection of transition graphs for the configurations of the form $\langle Fe_1 \rangle$, where F runs over all the functions used. We can combine all these graphs into one graph, which we denote by G_{tot}, by introducing the special variable e_0 which stands for the contents of the view-field. So the contraction $e_0 \rightarrow \langle F^m e_1 \rangle$ should read: "if the configuration $\langle F^m e_1 \rangle$ is in the view-field, then." The total graph is

$$G_{tot} = ((e_0 \rightarrow \langle F^1 e_1 \rangle)\ dev\ \langle F^1 e_1 \rangle + (e_0 \rightarrow \langle F^2 e_1 \rangle)\ dev\ \langle F^2 e_1 \rangle + \ldots + (e_0 \rightarrow \langle F^n e_1 \rangle)\ dev\ \langle F^n e_1 \rangle)$$

It is also convenient to end each branch by assigning the resulting configuration to e_0, which is read: "put C_2 into the view-field." Now every branch in the total graph, as well as in configuration developments, consists of contractions and assignments only. It starts with a contraction for e_0, which specifices the configuration we put in the view-field, and ends with the assignment to e_0, which specifies what will appear in the view-field as the result. The development of a configuration C is different in that its branches start with contractions for the variables of C. The total graph may be viewed as *dev* e_0.

In the development of C_1 we can apply driving to every active configuration C_2^k, replacing it with its development. We shall then have the history of two steps of computation starting with C_1. Now we can drive all active configurations in the developments of all C_2^k, and so on. At every stage of this process we have a tree where the walks represent generalized computation histories. Every walk ends either with a passive expression (terminated walk) or with a call of some configuration.

If we use the breadth-first principle and drive indefinitely long, we construct an infinite tree without active configurations, which includes all possible computation histories. Some walks in this tree may terminate, while others may be infinite. A walk that terminates in n steps has the form:

$$(var\ C_1 \rightarrow L^1)(var\ L^1 \rightarrow L^2) \ldots (var\ L^{n-1} \rightarrow L^n)(E^n \leftarrow e_0),$$

where E^n is a passive expression which can include only the free variables from L^n. We can fold all n contractions into one $(var\ C_1 \rightarrow L)$. Recalling that we deal with the development of C_1, we can write the formula of a terminated walk as

$$(e_0 \rightarrow C_1)(var\ C_1 \rightarrow L)(E^n \leftarrow e_0).$$

This is essentially a formula for one step of the Refal machine. A configuration can be seen as a function of its free variables. Each terminated walk in the infinite driving of C_1 gives a subset of the domain of C_1 and the algorithm of computing C_1 on this subset by one Refal step, that is, by simply restructuring C_1 into E^n. We call these subdomains *the ultimate neighborhoods* in the computation of C_1. Infinite driving breaks down the domain of the initial configuration into ultimate neighborhoods. It is analogous to the enumeration of the pairs argument-value in the theory of recursive functions. In our case, each pair consists of C_1 $(var\ C_1 \rightarrow L)$ and E^n (i.e., a pair of sets of expressions, not individual expressions); the rule of transforming the argument into the value goes with the pair.

6. EXAMPLES OF SUPERCOMPILATION

In Section 3 we defined how supercompilation is different from the construction of the infinite graph of states through driving. Space limitations do not allow us to go into detail of possible strategies of supercompilation. We simply discuss some examples of supercompilation as performed by the CCNY supercompiler.

The final outcome of supercompilation will again be in Refal, which allows to evaluate the optimizing effect of the transformation. It is easy to convert a graph resulting from supercompilation into a standard Refal program. To every basic configuration, C_i, a function is put in correspondence with the format

$\langle C_i\ var\ C_i\rangle$. The walks in the graph have one of the two forms:

passive end: $(var\ C_i \rightarrow L^1)(var\ L^1 \rightarrow L^2) \ldots (var\ L^{n-1} \rightarrow L^n)(E^n \leftarrow e_0)$
active end: $(var\ C_i \rightarrow L^1)(var\ L^1 \rightarrow L^2) \ldots (var\ L^{n-1} \rightarrow L^n)(E^n \leftarrow var\ C_j)(C_j \leftarrow e_0)$

In each walk we fold the contractions; let the result be $(var\ C_i \rightarrow L)$. Then we form the sentence $\langle C_i L\rangle = E^n$ in the case of a passive walk-end, and the sentence $\langle C_i L\rangle = \langle C_j E^n\rangle$ if the walk-end is active. Taking all sentences in their order, we have the definition of the function C_i. Taking the definitions for all basic configurations, we have the complete Refal program equivalent to the original program as far as the computation of the initial configuration C_1 is concerned.

Let us consider a very simple example of supercompilation. Take the following definitions:

$$\begin{aligned}
\langle F^a A e_1\rangle &= B\langle F^a e_1\rangle \\
\langle F^a s_2 e_1\rangle &= s_2\langle F^a e_1\rangle \\
\langle F^a\rangle &= \\
\langle F^b B e_1\rangle &= C\langle F^b e_1\rangle \\
\langle F^b s_2 e_1\rangle &= s_2\langle F^b e_1\rangle \\
\langle F^b\rangle &= \\
\langle F e_1\rangle &= \langle F^b\langle F^a e_1\rangle\rangle
\end{aligned}$$

Let the initial configuration C_1 be $\langle F e_1\rangle$. After the first step of driving, it becomes, without any branchings and contractions, $\langle F^b\langle F^a e_1\rangle\rangle$. We call such configurations as $\langle F e_1\rangle$ *transient*. There is no need to keep them in the memory of the supercompiler. We simply redefine C_1 as $\langle F^b\langle F^a e_1\rangle\rangle$. According to the inside-out semantics of the standard Refal machine, the evaluation of these nested function calls requires two passes of the argument e_1. However, there is nothing to prevent us from using the outside-in principle during the driving as an optimization technique. Whereas on the programming level we can choose the inside-out semantics for its simplicity, or the outside-in semantics for its sophistication, the supercompiler should always use the outside-in evaluation in order to implement a more efficient algorithm. The only risk we run is that the domain of the function will be extended, but this is hardly a risk at all—and in this specific case even this does not happen. So we start from the outside, trying to drive the call of F^b. We immediately find, however, that the driving is prevented by the inner call of F^a. So we go inside and drive this call. This results in the graph:

$$\begin{aligned}
(e_0 \rightarrow C_1)((e_1 \rightarrow A e_1)\langle F^b B\langle F^a e_1\rangle\rangle \leftarrow e_0 \\
+\ (e_1 \rightarrow s_2 e_1)\langle F^b s_2\langle F^a e_1\rangle\rangle \leftarrow e_0 \qquad (1) \\
+\ (e_1 \rightarrow empty)empty \leftarrow e_0)
\end{aligned}$$

What should we do now? At every step of supercompilation we must decide whether each active configuration should be driven further, be declared basic, or

reduced to a previous configuration. Even though the CCNY supercompiler will make the required choice automatically, we have to present its choice here without explanation, since we do not go into a formal definition of the strategy we use. So we decide to go on with the first active configuration we see in (1), that is, $\langle F^b B \langle F^a e_1 \rangle \rangle$. We again start from the outside, but this time we find that the inner active expression does not prevent us from successful matching and driving. The development is a very simple graph:

$$C\langle F^b \langle F^a e_1 \rangle \rangle \leftarrow e_0 \tag{2}$$

The active configuration here is identical to C_1, which is basic by definition, so we do not drive it further.

The driving of the second active configuration in (1), $\langle F^b s_2 \langle F^a e_1 \rangle \rangle$, yields

$$\begin{aligned} &(s_2 \rightarrow B)C\langle F^b \langle F^a e_1 \rangle \rangle \leftarrow e_0 \\ &+ s_2 \langle F^b \langle F^a e_1 \rangle \rangle \leftarrow e_0 \end{aligned} \tag{3}$$

Again, the end configurations are all identical to C_1. Substituting (2) and (3) in (1), and reducing the end configurations to C_1 (which in this case is trivial), we have the final graph:

$$\begin{aligned} &(e_0 \rightarrow C_1)((e_1 \rightarrow Ae_1)C\langle C_1(e_1) \rangle \leftarrow e_0) \\ &\quad + (e_1 \rightarrow s_2 e_1)((s_2 \rightarrow B)C\langle C_1(e_1) \rangle \leftarrow e_0 + s_2 \langle C_1(e_1) \rangle \leftarrow e_0) \\ &\quad + (e_1 \rightarrow empty)empty \leftarrow e_0) \end{aligned}$$

Thus the only function in the basis is C_1. Folding contractions in this graph, we have the Refal program:

$$\begin{aligned} \langle C_1(Ae_1) \rangle &= C\langle C_1(e_1) \rangle \\ \langle C_1(Be_1) \rangle &= C\langle C_1(e_1) \rangle \\ \langle C_1(s_2 e_1) \rangle &= s_2 \langle C_1(e_1) \rangle \\ \langle C_1(\ ) \rangle &= \end{aligned}$$

If the outside-in execution of a Refal program results in exactly the same steps as the inside-out execution, the program is called *bidirectional.* The program for $\langle C_1(e_1) \rangle$ is bidirectional because there is always only one active term in the view-field. The original program for $\langle Fe_1 \rangle$ is not bidirectional. Its inside-out execution requires $2n$ loops if the length of the input string is n. The outside-in (lazy) evaluation involves only n loops. It is well known, however, that lazy evaluation entails certain overheads, because of the necessity of analyzing at every step which of the activation brackets must be developed first. The use of the super-compiler gives the best solution to the problem. It implements the same efficient algorithm as the lazy evaluator, but executes the overhead operations at compile time. This results in a bidirectional program that reflects the semantics of the outside-in evaluation but can be directly executed on the simple inside-out machine. The supercompiler transforms a two-pass algorithm into a one-pass.

It is not always the case, though, that the outside-in evaluation is algorithmically better than the inside-out evaluation. Consider the initial configuration:

$$\langle \text{rep3}\langle F^a e_1\rangle\rangle \tag{4}$$

where the function *rep*3 (repeat three times) is defined as

$$\langle \text{rep3 } e_x\rangle = (e_x)(e_x)(e_x)$$

With the inside-out rule, we compute F^a and then make two copies of the result and form a list of the three identical subexpressions. With the outside-in strategy, we find that the step execution for the function *rep*3 is not prevented, so we make the step, which results in the configuration:

$$(\langle F^a e_1\rangle)(\langle F^a e_1\rangle)(\langle F^a e_1\rangle)$$

When this configuration is evaluated, the function call $\langle F^a e_1\rangle$ is evaluated three times—an obvious waste.

The outside-in evaluation strategy can be modified so as to bar this effect. In Refal it is easy to keep track of the copying of the values of variables. We call a variable *duplicated* if in the right side of the sentence there are more occurrences of this variable than in the left side. Duplicated variables can be marked in the left sides of a program's sentences by way of preprocessing. Now, when the supercompiler (or a direct outside-in evaluator) establishes that a Refal step is not prevented by any of the inner active terms, it takes the additional step of checking that none of the duplicated variables (at this moment they have already been assigned definite values) has active subexpressions. If this is not the case, the step execution must be delayed, and the active subexpressions of the duplicated variables developed first, in some order.

It does not always happen that every active configuration called in the graph of states can either be driven further or can be recognized as one of the configurations already declared basic. For example, redefine the function F^a above as follows:

$$\begin{aligned}
\langle F^a e_1\rangle &= \langle F^1(\ )e_1\rangle \\
\langle F^1(e_1)Ae_2\rangle &= \langle F^1(e_1B)e_2\rangle \\
\langle F^1(e_1)s_x e_2\rangle &= \langle F^1(e_1 s_x)e_2\rangle \\
\langle F^1(e_1)\rangle &= e_1
\end{aligned}$$

Let the initial configuration be $\langle Fe_1\rangle$ again. After the obvious two steps, we have $\langle F^b\langle F^1(\ )e_1\rangle\rangle$ as the new C_1. Driving from the outside, we find that we cannot make a step in F^b, we therefore make a step in F^1:

$$\begin{aligned}
(e_0 \rightarrow C_1)((e_1 \rightarrow Ae_1)\langle F^b\langle F^1(B)e_1\rangle\rangle \leftarrow e_0 \\
+\ (e_1 \rightarrow s_2 e_1)\langle F^b\langle F^1(s_2)e_1\rangle\rangle \leftarrow e_0 \\
+\ (e_1 \rightarrow \textit{empty})\langle F^b\rangle \leftarrow e_0)
\end{aligned} \tag{5}$$

Neither of the two active configurations here coincides with or is a special case of C_1. We try to drive on. We can see that the F^b call will not be ready for

development again, and will never be ready as long as F^1 is called recursively, because F^1 puts each new symbol in its own "pouch," not outside, as F^a does. Since the recursive calls reproduce themselves at each step, the driving could go on infinitely. This is the situation when a new configuration C_2 is "dangerously close" to the old C_1 (the same function F^1 is being developed), so that we cannot simply drive it on, yet we cannot reduce C_2 to C_1. We have to construct a generalization of C_1 and C_2, (i.e., a configuration C_g such that both $C_1 : C_g$ and $C_2 : C_g$ succeed). Then we reduce the old configuration C_1 to C_g, declare C_g basic, and develop it in the hope that this time we will be able to close the graph.

The possible algorithms of generalization turn out to be rather complex in the full Refal. If we limit ourselves to constructor-formed trees, generalization simplifies. We cannot go into detail here, but in our example the simple technique known as left-to-right L-generalization leads to success. Going from left to right in both configurations, we factor out those structural components of the matching process that are the same and, when they are not the same, we replace the whole remaining subexpression by an e-variable. So the generalization of *empty* with any nonempty expression is an e-variable; different symbols generalize to a symbol variable, and so on.

In supercompilation, we use the depth-first principle as the basic approach. The generalization of the first new configuration $\langle F^b\langle F^1(B)e_1\rangle\rangle$ with the old one $\langle F^b\langle F^1(\ )e_1\rangle\rangle$ yields $\langle F^b\langle F^1(e_2)e_1\rangle\rangle$. Taking this configuration as the new basic $\langle C_2(e_1)(e_2)\rangle$, and redriving it from the moment of generalization, we transform (5) as follows:

$$
\begin{aligned}
&(e_0 \to C_1)(\mathit{empty} \leftarrow e_2)\langle C_2(e_1)(e_2)\rangle \leftarrow e_0 \qquad (6)\\
&(e_0 \to C_2)((e_1 \to Ae_1)\langle F^b\langle F^1(e_2B)e_1\rangle\rangle \leftarrow e_0\\
&\qquad + (e_1 \to s_3e_1)\langle F^b\langle F^1(e_2s_3)e_1\rangle\rangle \leftarrow e_0\\
&\qquad + (e_1 \to \mathit{empty})\langle F^be_2\rangle \leftarrow e_0)
\end{aligned}
$$

Now the first two active configurations in the development of C_2 can be recognized as C_2. The third one will be found basic, and the program for it will be identical, except for format differences, to that for F^b. In the end, we have the program:

$$
\begin{aligned}
\langle C_1(e_1)\rangle &= \langle C_2(\ )(e_1)\rangle\\
\langle C_2(Ae_1)(e_2)\rangle &= \langle C_2(e_1)(e_2B)\rangle\\
\langle C_2(s_3e_1)(e_2)\rangle &= \langle C_2(e_1)(e_2e_3)\rangle\\
\langle C_2(\ )(e_2)\rangle &= \langle C_3(e_2)\rangle\\
\langle C_3(Be_1)\rangle &= C\langle C_3(e_1)\rangle\\
\langle C_3(s_2e_1)\rangle &= s_2\langle C_3(e_1)\rangle\\
\langle C_3(\ )\rangle &=
\end{aligned}
$$

This program differs from the original one in that there are no nested calls in the right sides. This is the result of changing the *basis* of the program to include a nested configuration of the original functions. It does not make the new program significantly more efficient, though, only a bit easier to implement.

In the preceding we have tacitly assumed that there are no nested configurations in the graph resulting from supercompilation. This is not always the case,

however; instead of taking the combination $\langle F^b\langle F^1(e_2)e_1\rangle\rangle$ as a basic configuration, we could *decompose* it as

$$(\langle F^1(e_2)e_1\rangle \leftarrow h_1)\langle F^b h_1\rangle \leftarrow e_0$$

Here the variable h_1 stands for the "hole" effected by the removal of a subexpression. It is basically the same as an e-variable, since the removed subexpression may evaluate to anything. But it is convenient to have the holes be syntactically different. Both the inner configuration $\langle F^1(e_2)e_1\rangle$ in this case, and the remaining outer configuration (with holes replaced by e-variables), are declared basic in decomposition.

In the preceding example, we could, in the very beginning, decompose

$$C_1 = \langle F^b\langle F^1(\ )e_1\rangle\rangle = (\langle F^1(\ )e_1\rangle \leftarrow h_1)\langle F^b h_1\rangle \leftarrow e_0$$

Then, after a generalization, $\langle F^1(e_2)e_1\rangle$ and $\langle F^b e_1\rangle$ would be declared basic, and the transformed program would reproduce the original one. While in this case we could decompose or not, in other cases decomposition may be necessary to construct a finite graph. Take the recursive definition of the factorial in the unary system:

$$\begin{aligned}\langle \text{fact } 01\rangle &= 01\\ \langle \text{fact } e_n1\rangle &= \langle \text{mult}(e_n)\langle \text{fact } e_n\rangle\rangle\end{aligned}$$

For $C_1 = \langle \text{fact } e_n\rangle$, we have, after the first step of driving:

$$\begin{aligned}(e_0 \rightarrow C_1)((e_n \rightarrow 01)01 &\leftarrow e_0\\ + (e_n \rightarrow e_n1)\langle \text{mult}(e_n)\langle \text{fact } e_n\rangle\rangle &\leftarrow e_0)\end{aligned}$$

We must recognize here that the leading active subexpression in the second branch is a match to C_1 (i.e., basic), and take it out by decomposition. This leads to declaring $\langle \text{mult}(e_n)e_1\rangle$ basic, too. If we do not do so, but simply drive on, we face an infinite sequence of nested calls:

$$\begin{aligned}&\langle \text{mult}(e_n)\langle \text{fact } e_n\rangle\rangle\\ &\langle \text{mult}(e_n1)\langle \text{mult}(e_n)\langle \text{fact } e_n\rangle\rangle\rangle\\ &\langle \text{mult}(e_n11)\langle \text{mult}(e_n1)\langle \text{mult}(e_n)\langle \text{fact } e_n\rangle\rangle\rangle\rangle \ldots\end{aligned}$$

Another reason why decomposition may be necessary is the use of built-in and, in Refal, undefined functions. Such function calls must either be immediately computed, if it happens that all their arguments are known, or treated as basic configurations. Using the host computer's numbers, we can define the factorial as follows:

$$\begin{aligned}\langle \text{fact } 1\rangle &= 1\\ \langle \text{fact } e_n\rangle &= \langle \text{mult}(e_n)\langle \text{fact}\langle \text{prec } e_n\rangle\rangle\rangle\end{aligned}$$

where the functions *mult* and *prec* are built-in. When the nested call appears in the graph and is developed, the call of $\langle \text{prec } e_n\rangle$ must be taken out. Then *fact* is taken out as basic, which results in a complete decomposition.

Consider an example that shows how a supercompiler can deal with another type of redundancy, the occurence of the same variables more than once in the initial expression. The problem we want to solve can be formulated as a theorem,

namely: If $*S = S*$, where S is a string of symbols, then S consists of asterisks * only. In algorithmic terms, we define equality by

$$\langle=(s_1e_2)(s_1e_3)\rangle = \langle=(e_2)(e_3)\rangle \quad \text{(E1)}$$
$$\langle=(\ )(\ )\rangle = T \quad \text{(E2)}$$
$$\langle=e_z\rangle = F \quad \text{(E3)}$$

and try to transform the program for the configuration $C_1 = \langle=(*e_s)(e_s^*)\rangle$ into a program that simply checks that all the symbols in e_s are asterisks.

So we start with driving the equality call in C_1. Since it is not quite trivial, let us trace the use of the GMA. We match C_1 to the left side of (E1). First we have case 4 (i.e., two subproblems: $*e_s : s_1e_2$ and $e_s\ * : s_1e_3$). The first one is resolved easily and results in one PRT, which is $(* \leftarrow s_1)(e_s \leftarrow e_2)$. Resolving the second, we face case 6L, which produces a pair of contractions $(e_s \rightarrow)$ and $(e_s \rightarrow s_4e_s)$. (Note that we could use s_1 instead of s_4, since the variable s_j in the rule must be *new* only with regard to the varlist it belongs to, and the actual implementation of the supercompiler will do so in this situation. However, for the convenience of the reader, we pick up an entirely new variable.) Applying the contractions, we have two CPRTs: $* : s_1e_3$ and $s_4e_s^* : s_1e_3$. Since s_1 has taken the value *, we have case 3 with internal clashes in both CPRTs. The clash $* : *$ is resolved trivially in the first one, while in the second we have $s_4 \rightarrow *$. In both terms, the matching then succeeds. The result of matching with the left side of (E1) is two contractions, $e_s \rightarrow$ *empty* and $e_s \rightarrow *e_s$. Thus the driving through (E1) produces two branches:

$$(e_s \rightarrow)\ \langle=(*)(*)\rangle \leftarrow e_0 \quad \text{(B1)}$$
$$+\ (e_s \rightarrow *e_s)\ \langle=(**e_s)(*e_s\ *)\rangle \leftarrow e_0 \quad \text{(B2)}$$

The second sentence, (E2), is not applicable; driving through (E3) results in the third branch:

$$F \leftarrow e_0 \quad \text{(B3)}$$

The end configurations in (B1) and (B2) are transient. In one more step of driving we come to the graph:

$$\begin{aligned}(e_0 \rightarrow C_1)((e_s \rightarrow)\ T &\leftarrow e_0\\ + (e_s \rightarrow *e_s)\ \langle=(*e_s)e_s\ *)\rangle &\leftarrow e_0\\ + F &\leftarrow e_0)\end{aligned}$$

The only active configuration here is reduced (identical) to C_1, and we come to the program:

$$\langle C_1(\ )\rangle = T$$
$$\langle C_1(*e_s)\rangle = \langle C_1(e_s)\rangle$$
$$\langle C_1(e_s)\rangle = F$$

which is exactly what we want.

More examples of the work of the CCNY supercompiler can be found in [23]. The examples illustrate the following applications of supercompilation: program specialization, program optimization, the use of interpretive definitions of

programming languages to produce efficient compiled programs, problem solving of different kinds, and theorem proving.

7. THE SUPERCOMPILER AS A COMPILER

We now give a more substantial example of the work of the CCNY supercompiler. In accordance with the procedure described in Section 1(a), we define the semantics of a programming language in Refal and write a program in that language. The supercompiler then translates it into a target language. The object language we choose is a simplified structured FORTRAN. There are no explicit declarations and no subroutines. There is only one type of data: integers. The usual arithmetic and logical expressions are allowed, as well as the input/output procedures READ and OUT. Control transfer is managed by *if-* and *while-*statements. We call this language EXSEQ (for "execute sequentially"). The program in EXSEQ we are going to translate is as follows:

```
(1) READ N;
    IF N < 0 THEN OUT('ERROR') ELSE
    IF N = 0 THEN OUT(1) ELSE
              (F := N;
               WHILE N > 2 DO(N := N - 1; F := F*N);
               OUT(F))
```

The complete semantical definition of EXSEQ is too long (about 70 lines in Refal) to reproduce in full, but we would like to give an idea of what it is like and how it works. The main semantic function has the format ⟨exec P on (S)⟩, "execute the program P on the state S". The state of the computing system is defined by a list of terms (*Identifier* = *Value*). Two standard identifiers, INPUT and OUTPUT, for the input and output streams, must always be present in the state. When P is interpreted, we put in the view-field ⟨exec P on ((INPUT = ⟨input⟩)(OUTPUT =))⟩. Here ⟨input⟩ is the call of the input function of the Refal system, which provides the input for P. The original value of OUTPUT is empty. When we want to translate P, we supercompile the call ⟨exec P on ((INPUT = e_i)(OUTPUT =))⟩, and, in the supercompiled program, interpret e_i as the input stream.

The function *exec* is defined as follows:

$$\langle \text{exec } e_1; e_2 \text{ on } (e_s)\rangle = \langle \text{exec } e_2 \text{ on } (\langle \text{exec } e_1 \text{ on } (e_s)\rangle)\rangle$$
$$\langle \text{exec } e_1 \text{ on } (e_s)\rangle = \langle \text{exstat } e_1 \text{ on } (e_s)\rangle$$

which expresses the semantics of sequential execution of statements separated by semicolons. In the definition of the function *exstat*, "execute statement," we introduce, when necessary, special functions for executing special kinds of statements:

$$\langle \text{exstat } s_x := e_e \text{ on } (e_s)\rangle = \langle \text{exass}(s_x)\langle \text{eval } e_e \text{ on } (e_s)\rangle\rangle$$
$$\langle \text{exstat IF } e_c \text{ THEN } e_1 \text{ ELSE } e_2 \text{ on } (e_s)\rangle = \langle \text{exif}(e_1)(e_2)\langle \text{eval } e_c \text{ on } (e_s)\rangle\rangle$$
$$\langle \text{exstat WHILE } e_c \text{ DO } e_p \text{ on } (e_s)\rangle = \langle \text{exwhile}(e_c)(e_p)\langle \text{eval } e_c \text{ on } (e_s)\rangle\rangle$$
$$\langle \text{exstat READ } s_x \text{ on } (e_1(\text{INPUT}=(e_v)e_w)e_2) = \langle \text{exass}(s_x)(e_v)(e_1(\text{INPUT}=e_w)e_2)\rangle$$
$$\langle \text{exstat OUT}(e_e) \text{ on } (e_s)\rangle = \langle \text{out}\langle \text{eval } e_e \text{ on } (e_s)\rangle\rangle$$
$$\langle \text{exstat}(e_1) \text{ on } (e_s)\rangle = \langle \text{exec } e_1 \text{ on } (e_s)\rangle$$
$$\langle \text{exstat on } (e_s)\rangle = e_s$$

In these statements, we use the simple character of the syntax of EXSEQ; otherwise we would have separated the syntax analysis from the interpretation of the resulting tree. Identifiers are symbols (atoms) of Refal, thus we use s_x for an identifier while parsing the assignment statement (the first sentence); the value of e_e is then defined uniquely by the rules of Refal matching. The evaluation function *eval* returns $(V)(S)$, where V is the value of the expression and S is the state after evaluation. The function *exass*:

$$\langle \text{exass}(s_x)(e_v)(e_1(s_x = e_e)e_2)\rangle = (e_1(s_x = e_v)e_2)$$
$$\langle \text{exass}(s_x)(e_v)(e_s)\rangle = ((s_x = e_v)e_s)$$

changes the value of s_x, if it had one before, or adds a new term to the state if it did not.

When *if*- and *while*-statements are processed by *exstat*, the condition e_c is evaluated first. The function *exif* is defined in the obvious way:

$$\langle \text{exif}(e_1)(e_2)(T)(e_s) = \langle \text{exstat } e_1 \text{ on } (e_s)\rangle$$
$$\langle \text{exif}(e_1)(e_2)(F)(e_s) = \langle \text{exstat } e_2 \text{ on } (e_s)\rangle$$

and *exwhile* is defined in a similar manner. The *read* statement expects that the INPUT list is not empty (otherwise, an abnormal stop would occur). It chops off the first element of the list and assigns it to s_x. OUT evaluates its argument and calls the function *out*, which adds it to the OUTPUT list and prints it out. Parentheses can be used as begin–end brackets. The empty statement leaves the state unchanged. The function *eval* is defined in the same style as *exstat*.

Unlike the examples in the preceding section, where we used Refal as the target language, we express the graph of states resulting from supercompilation in terms of an assembler-like language. Both the Refal graph and its equivalent in the assembler-like language are essentially flowcharts; one could write a program that would do the conversion, even though in this case it was done manually. The configurations of the Refal graph become labels in the assembler program. The Refal variables are converted into symbolic names; we used the following coding: e_1 in the configuration C_4 becomes E41, e_2 in C_{28} beomces E282, and so on. When a variable is simply passed from one function to another, we retain its original notation. In our case, the structure of the Refal graph after supercompilation (unlike the structure before supercompilation) is very simple, so that no recursive procedures are required. We simply translate configuration calls as go-tos (and skip those that point to the next statement). We keep all configuration labels in the program, even though only a few are used in go-tos, in order to give an idea of the structure of the Refal graph. This is the supercompiled program:

```
C1:  equivalence E21 = input-stream;
C2:  E41 := head(E21); E42 := tail(E21);
C4:  E53 := E41 − 0;
C5:  if E53 < 0 then begin print('ERROR');
                           STATE := ('N=' E41)('INPUT=' E42)('OUTPUT = ERROR');
                           normal-stop
                      end
                 else go-to C7;
```

```
C7:  if E41 = 0 then begin print(1);
                         STATE := ('N=' 0)('INPUT=' E42)('OUTPUT=' 1);
                         normal-stop
                     end
                else go-to C9;
C9:  E281 := E41; E282 := E41; E283 := E42; E284 := 2 - E41;
C28: if E284 < 0 then go-to C38
                 else begin print(E281);
                         STATE := ('F=' E281)('N=' E282)('INPUT =' E283)
                           ('OUTPUT=' E281);
                           normal-stop
                      end;
C38: E282 := E282 - 1;
C42: E281 := E281*E282;
C44: E284 := 2 - E282; go-to C28;
```

The major difference between this program and the program one would have written as the translation of (1) is that the supercompiled program makes an assignment to STATE each time it comes to a normal stop. This is a consequence of our definition of the EXSEQ semantics, according to which the execution of every statement produces STATE. When it is passed from one active configuration to another, we do not see it, because it is only the variables in STATE that matter, not the actual form of the configuration. But in the end, when the whole view-field becomes one passive configuration, the supercompiler outputs it as the value of the overall semantic function. This effect is not difficult to eliminate.

The program is quite good in other respects. Such small things as the subtraction of 0 from E41 can be eliminated, as they usually are, in the order of optimization. Note that there are ten basic configurations in the final program, but in order to find them the supercompiler had to consider 44 configurations as possibly basic, and then to discard the majority in repeated generalizations.

To evaluate the effect of supercompilation as an optimizing procedure, we have also converted the graph of states into a Refal program. It runs about 40 times faster than before supercompilation (i.e., in the direct interpretation mode). We also experimented with Lisp as the object language. A semantic definition of Lisp was written in Refal, and a small program in Lisp was run both interpretively and after supercompilation. The speed-up factor from supercompilation was again about 40.

Even though supercompiled programs may be quite fast, the process of supercompilation itself is performed in the interpretation mode, and is very slow. The supercompilation of program (1) took slightly less than two minutes on an IBM/370. For the semantic definition of programming languages in Refal to be useful in practice, the process of supercompilation itself must be supercompiled. We have kept this in mind from the start of the supercompiler project. In [22], one can find a more detailed discussion, as well as the Refal expressions for the automatic creation of compilers and compiler generators. The present supercompiler model is too big and too slow to be applied to itself as a whole. We are now developing the techniques of bootstrapping a supercompiler in parts. To judge such translation times as the two minutes for program (1) properly, one must take into account that there are two independent slow-down factors now in force. The first is that the current implementation of Refal is essentially

interpretive. We hope that a good compiler can give a speed-up by factor of 10 to 20. The other factor is that of supercompilation; let us take 40 as the first approximation. We can then hope to gain three orders of magnitude, which should make automatic construction of compilers practicable.

In a significant development, N. Jones et al. [10] succeeded in applying a program for partial evaluation, called MIX, to itself, and producing thereby compilers and compiler generators for simple languages. The operation of MIX can be described as a very simple form of supercompilation where only one-function-call configurations are allowed and no generalization is made. In addition, MIX requires a hand-made annotation of the original program, which makes a distinction between *eliminable* and *residual* calls; this corresponds to a supercompilation with the whole set of basic configurations (residual calls) supplied by the user. Jones has shown that the annotation dramatically reduces the volume of the partial evaluator, which, in fact, makes self-application possible. This idea should also be tried with a supercompiler. In our work at CCNY we have probably overemphasized the automatism of supercompilation, which blew up the size of the programs and caused a delay in self-application.

On the other hand, because of the sheer volume and complexity of information we want to process, what we ultimately want is a completely autonomous supercompiler. The self-application of such a supercompiler is a hard technical problem. To perform sophisticated program transformation on its own, the supercompiler must be big enough. But the bigger it is, the more time and space is needed for the supercompiler to apply itself to itself. The problem is similar to that of constructing a flying machine: we must give it a powerful engine, but by trying to increase the power of the engine we also increase its weight. Even after the principles of a flying machine had been understood, considerable technical progress was required to actually start flying. This is the situation with a self-applicable supercompiler now. But we see no reason why it should not eventually "rise into the air."

ACKNOWLEDGMENTS

The author expresses his appreciation and gratitude to the members of the group working on the supercompiler: Robert Nirenberg, James Piccarello, and Dimitri Turchin. We have made some changes to the syntax of Refal following the suggestions made by John Backus, which we highly appreciate. We appreciate as well the discussion of the supercompiler. The final text of this paper was written at the University of Copenhagen, where the author, having been invited by Neil Jones, spent a semester. It is a pleasant duty to thank Neil Jones for the invitation, the many fruitful discussions, and, in particular, his suggestions on the semantics of Refal.

REFERENCES

1. BECKMAN, L., HARALDSON, A., OSKARSON, O., AND SANDEWALL, O. A partial evaluator and its use as a programming tool. *Artif. Intell. 7* (1974), 319–357.
2. BURSTALL, R. M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24* (1977), 44–67.
3. ERSHOV, A. P. On the essence of translation. In *Formal Description of Programming Concepts*, E. J. Neuhold, Ed., North-Holland, Amsterdam, 1977, 391–418.

4. ERSHOV, A. P. Mixed computation: Potential applications and problems for future studies. *Theor. Comput. Sci. 18* (1982), 41–67.
5. FRIEDMAN, D. P., AND WISE, D. S. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, Michaelson and Millner, Eds, Edinburgh University Press, 1967, 257–284.
6. FUTAMURA, Y. Partial evaluation of computation process—an approach to a compiler-compiler. *Syst. Comput. Control 2*, 5 (1971), 45–50.
7. FUTAMURA, Y. Partial computation of programs. In *Proceedings RIMS Symposium on Software Science and Engineering*, LNCS 147, Springer-Verlag, New York, 1983, 1–35.
8. HENDERSON, P., AND MORRIS, J. H., JR. A lazy evaluator. In *Proceedings 3rd Symposium on POPL* (1976), 95–103.
9. JONES, N., AND SCHMIDT, D. Compiler generation from denotational semantics. In *Semantics-Driven Compiler Generation*, LNCS 94, Springer-Verlag, New York, 1980, 70–93.
10. JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. An experiment in partial evaluation: The generation of a compiler generator. In *International Conference on Rewriting Techniques and Applications* (1985), Springer-Verlag, New York, (forthcoming).
11. JONES, N., AND TOFTE, M. Some principles and notation for the construction of compiler generators. DIKU, Internal Rep., Univ. of Copenhagen, 1983.
12. LOMBARDI, L. A. Incremental computation. In *Advances in Computers, 8*, Academic Press, New York, 1967.
13. NILLSON, N. J. Artificial intelligence prepares for 2001. *AI Mag. 4* (1983), 7–14.
14. O'SHEA, T., AND EISENSTADT, M. *Artificial Intelligence.* Harper and Row, New York, 1984.
15. PARTSCH, H., AND STEINBRUEGGEN, R. Program transformation systems. *ACM Comput. Surv. 15* (1983), 199–236.
16. REDDY, U. Narrowing as the operational semantics of functional languages. Submitted to the *Symposium on Logic Programming* (Boston, 1985).
17. *Basic Refal and its Implementation on Computers.* GOSSTROI SSSR, TsNIPIASS, Moscow, 1977. The authors are not indicated in the book. They are: V. F. Khoroshevski, And. V. Klimov, Ark. V. Klimov, A. G. Krasovski, S. A. Romanenko, I. B. Shchenkov, and V. F. Turchin. (In Russian)
18. SOWA, J. F. A prologue to Prolog. Draft, distributed with permission of the author, 1984.
19. TURCHIN, V. F. Equivalent transformation of recursive functions defined in Refal. In *Trudy Vsesoyuzn. Simpos. "Teoria Yazykov i metody progr,"* Alushta-Kiev, 1972, 31–42. (In Russian)
20. TURCHIN, V. F. A supercompiler system based on the language Refal. *SIGPLAN Not. 14* (1979), 46–54.
21. TURCHIN, V. F. The language Refal, the theory of compilation, and metasystem analysis. Courant Institute Rep. 20, 1980.
22. TURCHIN, V. F. Semantics definitions in Refal and automatic production of compilers. In *Semantics-Driven Compiler Generation*, LNCS 94, N. Jones, Ed., Springer-Verlag, New York, 1980, 443–474.
23. TURCHIN, V. F., NIRENBERG, R. N., AND TURCHIN, D. V. Experiments with a supercompiler. In *ACM Symposium on LISP and Functional Programming* (1982), ACM, New York, 47–55.
24. VUILLEMIN, J. Correct and optimal implementation of recursion in a simple programming language. *J. Comput. Syst. Stud. 9*, 3 (Dec. 1974).

Received December 1981; revised June 1984, January 1986; accepted February 1986