A Partial Evaluator, and its Use as a Programming Tool[1]

Lennart Beckman, Anders Haraldson, Östen Oskarsson and Erik Sandewall

Department of Mathematics, Linköping University, Linköping, Sweden

Recommended by R. Burstall

ABSTRACT

Programs which perform partial evaluation, beta-expansion, and certain optimizations on programs, are studied with respect to implementation and application. Two implementations are described, one "interpretive" partial evaluator, which operates directly on the program to be partially evaluated, and a "compiling" system, where the program to be partially evaluated is used to generate a specialized program, which in its turn is executed to do the partial evaluation. Three applications with different requirements on these programs are described. Proofs are given for the equivalence of the use of the interpretive system and the compiling system in two of the three cases. The general use of the partial evaluator as a tool for the programmer in conjunction with certain programming techniques is discussed.

1. Introduction

Partial evaluation has traditionally been considered as a toy, to be used only as a subroutine in very experimental research programs. The present paper argues that if properly developed and utilized, partial evaluation can also be very useful as a practical working tool for the serious programmer.

The paper describes a set of programs for partial evaluation (Sections 3 and 8), and experiments which have been done with them (Sections 4–6), as well as of how such programs can be put to use (the remaining sections). Some figures from the experiments are to be found in the appendix.

In the simplest case, a partial evaluation program is a program which takes a procedure P of some arguments $(x_1, x_2, \ldots, x_n)$ together with values $(c_1, \ldots, c_m)$ for the first m of these n arguments, and which generates a new procedure P' such that $P'(x_{m+1}, \ldots, x_n) = P(c_1, \ldots, c_m, x_{m+1}, \ldots, x_n)$ for all x. In doing so, it

[1] This research was sponsored by the Swedish Board for Technical Development (contract 4732) and by the Swedish Natural Science Research Council (contract Dnr 2654).

Artificial Intelligence 7 (1976), 319–357

should of course insert the constants c_i for the variables x_i in the definition of P, and apply functions for which all arguments are constants (as long as the functions do not have or depend on side effects).

Various other operations should also be included in the partial evaluator in order to make it practically useful. However, we can immediately make one observation, namely that the partial evaluator is a tool for increasing the efficiency of programs. In a sense which will be further articulated, it is thereby a tool for generating programs.

Being a program which operates on other programs, partial evaluators are more easily written in programming languages that have program/data equivalence, such as LISP. In what follows, we shall only be concerned with their use as a tool for a LISP programmer, for example in Artificial Intelligence research. Most of the discussion does carry over to more conventional programming languages, although implementing a partial evaluator is not as easy there.

In order to go into how a LISP programmer can make use of a partial evaluator, let us back up one step and consider a few of the problems that he has to face and will have to face in his daily work.

1.1. Many small programs

One significant problem is the increasing requirement for non-trivial data bases. Programs for language understanding, problem solving etc. have traditionally been run with minimal data bases, in the precise sense of minimal; only such data were loaded, as were deemed necessary to run one particular test or set of tests. Restrictions on memory size was one of several reasons for this practice. However, it is clear that coming systems must accommodate larger data bases for dictionaries and (other) sets of knowledge.

The larger data bases will still be small compared to their namesakes in the field of administrative data processing (ADP), but some of the attitudes that have developed for ADP data bases should carry over. In particular, the maintenance and care of the data base will no longer be a trivial activity, but instead an activity that is analogous in many ways to programming. The programmer will have to take up a second role as data base hacker.

"Hacking" a LISP data base is in many ways similar to hacking a LISP program. Both have to be edited, which can be done either with a text editor, or with a LISP editor which operates immediately on the list structures. But as with programs, there are various other utility operations on the data base which it is only reasonable to program in LISP (or whatever language is being used for the application), such as:

(a) printing; present a prescribed subset of a data base in an easily readable form,

(b) checking; check a prescribed subset of a data base for compliance with a certain structure specification (e.g. the specification of what structure a given program assumes of the data base),

(c) reorganization; scan a part of a data base and change its structure to fit a new application, or to conform to changes in the program that uses the data base.

The list could be continued, and it is not the topic of this paper to develop it further. The point we want to make is that the programmer's activity will not only consist of building and maintaining one large program; it will to an increasing extent also involve running miscellaneous small programs which operate on his data base.

An important consideration is then that there should be reasonable utility programs around, so that he does not have to write a new, specialized program each time he wants to make a routine operation on the data base. There are certainly situations where only a hand-tailored program will do the job, but there should also be considerable use for a relatively general utility program. Such program must then be parameter-driven, i.e. some of their arguments would specify the application (e.g. the structure of the data base segment that is to be processed), while other arguments would specify what segment of the data base is to be processed.

The claim we make is not merely that general procedures are good, and that data base operations is one of the places where they should be used. We claim in addition that a fairly small number of structuring methods (such as recursive property lists and internalization of lists) are sufficient for describing on low level most of the data base structures that people use in practice, and that the typical utility operations are fairly stereotype. This together implies that, unlike many other parts of LISP-type programming, data base hacking is an area where general procedures have a good chance of being truly useful.

The role of the partial evaluator is then straightforward; if he has a fairly general utility program U and a specification of an application, a programmer uses the partial evaluator to produce a tailored version of U that can be used repeatedly within that application.

1.2. One big program

Of course the programmer will not only be a data base hacker. Like in previous years, his primary concern will be one large LISP program which grinds his favorite axe. As his ambition increases and A.I. methodology develops he will want to develop even larger and more complex programs.

Methods for maintaining and extending large and complex programs are then of crucial importance. The currently fashionable structured programming approach is of no great help, because usually it is not possible, in the type of programming that we are concerned with here, to make a detailed plan of the program before embarking on the actual programming. In fact, the major purpose of the research is to develop the techniques that are needed for making such plans.

If the structured programming technique has little to say about that experimental programming situation, it is equally true that experimental programmers have said

little about how they structure their programs. In some cases, this may reflect a lack of structure, but more often the existing structure has been disregarded for being subjective, or a matter of "style"; "I will program in my own way, but I will not force it on you". Attention has centred on other issues, such as very-high-level languages, and more recently on programmer's-assistant-type systems which support the programmer in his work.

The fact that little has been written about programming methods in Artificial Intelligence research does not mean that there has been little progress. On the contrary, the technique has developed considerably during the last few years, as a result of both the work on very-high-level languages, and the various large programs that have been written. A discussion and a synthesis of the techniques that have actually been used, would now be instrumental both for developing the technique further, and as a basis for the design of appropriate tools and assistants for the programmer.

Such a discussion of techniques is of course beyond the scope of this paper. However, since the use and usefulness of a partial evaluater depends on the programming style or programming technique, we will outline one expected environment as an introduction to the proposed tool.

Experimental programs often start out as quite small, and then grow continuously to become many times their original size. After some time, they grow old and fat, and have to be re-written. Of course one wants to delay the aging, or in other words, make it maximally easy to add code while retaining the structure of the program. One wide-spread method for accomplishing this is to organize the program in a "data-driven" or "programmable" fashion. It is divided into two parts: a small although not trivially small structure part S, which contains schedules, interpreters and other executives, and a larger procedure part P, which consists of a set of procedures. These procedures are not called from S in the trivial way (which is to give each procedure a name and to use that name in S for writing calls to the procedure). Instead, the procedures are associated in a data base with various entities which may appear as data to the whole program. S reads (or accepts, in some manner) input data; does some work of its own but also looks up the code that is associated with input data, and executes that code (by interpretation or by calling *eval*).

We shall refer to this well-known programming technique as programming with procedures referenced from data items (PRD). We will mainly use the term "procedure" for such data-driven procedures. The abbreviations "P" and "S" that were introduced above will refer to a set of PRD's, and to a program that utilizes a set of PRD's, respectively.

Interpreters for high-level languages are of course organized using PRD's, with S as the interpreter and P as the interpreted program. Several aspects of high-level A.I. languages, such as new control structures and pattern-directed invocation, can be characterized as new mechanisms in the S part. Another straightforward example of this method is offered by programmable pretty-printers, which allow

the user to define how certain classes of expressions shall be pretty-printed. The user may for example associate a printing procedure with some function names, and the procedure would be called whenever a form with that name as the leading function is to be printed. In this case of course the argument for the printing program is a new program, and the fact that the procedure is associated with a function (=procedure) name should not obscure the observation that the procedure name is a data entity.

A final simple example is offered by a part of the SHRDLU program [15] where English text is read, and each English word may have a piece of code, a "semantic definition" for that word, associated with it.

The simple scheme of having exactly one procedure associated with each "word" in input is often extended in several ways. A system can be data-driven in several layers. For example, in language analysis, a top-level parser may go through names of syntactic classes to specialized procedures for each class, which again if necessary may go through individual words in the language to procedures that characterize that word. Also, it is sometimes useful to associate procedures with pairs, tuples or other composite structures, and not merely with single "words". For example, consider a relation-oriented data base system which for each relation maintains a number of procedures which shall be used when the relation is stored, looked up, deleted or updated. Such a system associates programs with pairs such as $\langle r$, STORE$\rangle$, $\langle r$, RETRIEVE$\rangle$, where r is the name of the relation. Finally, as this last example suggests, one will often want to associate several semi-independent procedures with a single tuple (or other structure), for example to specify several IF-NEEDED methods for retrieving the relation.

The intention of the PRD technique is of course that it should be possible to extend a program simply by adding more procedures to its P part, without changing its S part. The degree to which this is possible depends of course on how judiciously the S part was designed. But we believe that the use of this technique implies an alternative way of looking at highlevel languages.

Auxiliary A.I. languages have traditionally been implemented like all other high-level languages, i.e. as black-box systems. The user is supposed to believe that he is executing on a PLANNER machine, or a QA4 machine. Since he is not supposed to know about the inside of the black box, it is difficult for him to modify it, or combine several systems. This is no problem if the high-level language is considered as a panacea that will satisfy all programming needs, but is otherwise a serious disadvantage. However, if the interpreter for the programming system is viewed as a candidate program for the S part of various applications, then a number of interesting things follow:

(a) The programming system appears as modular. In some applications one might choose to use only some of the modules, e.g. the subsystem for data base management. In another application one might choose to exchange the pattern matcher for another one that fits the application.

(b) This in turn enables us to separate two purposes of a high-level language,

namely to provide useful control structures, and to support a human notation in the program (sometimes referred to as syntactic sugar). A control structure is implemented as a program for use on the S level, and supports certain structures on the P level. For each application, one can choose whatever S level tools are needed. A notation is implemented as a translator which accepts the notation, and deposits the appropriate code in a P level structure. Again, for each application, one can choose one or several translators that seem appropriate, or choose to program the data-driven procedures directly in the host language (e.g. LISP S-notation).

A third purpose of some high-level languages is not served if this viewpoint is taken, namely, the purpose to restrict the programmer's freedom so that he can not perform certain types of errors. That kind of support for the programmer may be appropriate for routine programming in well-known domains, but certainly not for experimental programming, where new methods are being developed. PRD programming emphasizes another purpose of high-level languages, namely to provide an application-oriented framework for the program. High-level control structures are of course one aspect of the application-oriented framework, and another aspect is the possibility to associate procedures with concepts that arise naturally in the application, or in its description.

In programs that perform a complex task such as language understanding in some domain, it is often appropriate to use several sub-systems for various aspects of the problem, e.g. a grammar sub-system, a deduction sub-system, a sub-system for psychological modelling, etc. Traditionally, each such sub-system has been a language system, i.e. a set of programs that support a special-purpose language. We prefer to view it as a PRD program system, consisting of conventions for the P level, one or more S level executives, and one or more translators. The alternative viewpoint should make it easier to modify existing sub-systems, to integrate several sub-systems, and to gradually extend the translators that generate code into the P part of sub-system. Such extensions could gradually take more and more application knowledge into account (from a data base of such knowledge).

Some of the difficulties with PRD programming should also be mentioned. The requirement to maintain the data-driven program structure while meeting demands for extending the program, may be a burden. New ideas for how to organize S, and for communication between procedures (e.g. message-sending between actor-like processes) can be expected to resolve that problem. The work of developing and maintaining several sub-systems, and the effort to remember their respective idiosyncracies may be very great. This probably has to be solved by a combination of several methods; on the one hand, an AI laboratory should maintain a library of such tools which have been developed so that they are truly user-friendly and easy to work with; and on the other hand, one should make it as easy as possible for the programmer to design his own high-level languages and the interpreters for them.

Since procedures in a PRD system may come from a variety of sources, (hand-

written or generated from one of several high-level languages, or from declarative information), code entry requires a program which can put pieces of code in the right position in a fairly complex structure. This calls for an "advising" technique which should be reminiscent of the one used in INTERLISP [13]. However, the INTERLISP advise has to edit arbitrary user code, whereas the advising into a PRD system could have a substantial knowledge about the program structure that it advises into.

Finally, if procedures are explicitly associated with entities in a description of the application, one simplifies the task of setting up a semantic description of the program, which specifies how the program is related to the application. Such semantic descriptions will be a necessary part of future programmer's-apprentice-type systems.

In summary, there are plenty of concrete and fairly short-term actions which can be taken to support the programmer who uses PRD methods, and also some interesting longer-term lines of development.

Programs for generating programs and for increasing their efficiency, can be useful in several ways in the context of data-driven procedures. The work of writing translators for specialized languages, can be significantly reduced if the programmer merely has to write a simple translator, which generates crude and inefficient code, and if he can draw on an existing program optimizer for cleaning up the code. Sometimes the desired high-level language consists partly or wholly of declarative information about the concepts in the chosen domain. Such declarative information would then be used by an interpreter. A partial evaluator operating on the interpreter and the declarative information may speed up execution considerably.

Finally, the efficiency of the entire PRD system might be improved by an operation which at least in principle is fairly simple. Namely, the idea to reference all code in P, and thus most code in the system, from high-level concepts, is very useful while the program is being developed and extended. It does however have a price in performance; operations which are in themselves fairly simple, may invoke several levels of referencing in order to find the pertinent code. It would then be worthwhile to open up the calls, i.e. insert the code in P into these places in S where that code may be called from. We shall refer to this operation as consolidation of the program. Consolidation is not altogether trivial in practice, and some simplifications of the resulting code will be needed. This again is an application of a partial evaluation program.

In summary, we have suggested that a partial evaluator can be a useful tool for a programmer, both as a way of making utility programs more economical and therefore useful, and as a tool in working on a large program using several subsystems. Utility programs, interpreters, translators, etc. are of course a set of tools for the programmer; he uses them to develop his program. The partial evaluator is a tool for maintaining the tools, or in other words a second-order tool.

2. Principles and Problems of Partial Evaluation

Partial evaluation has been used by several researchers and for a variety of purposes. We have identified the following applications (in an attempted chronological order):

Lombardi and Raphael used it as a modified LISP evaluation which attempted to do as much as it could with incomplete data [9].

Futamura studied such programs for the purpose of defining compilers from interpreters [6].

Dixon wrote such a program and used it to speed up resolution [2, 5].

Sandewall reports the use of such a program for code generation in [10].

Deutsch uses partial evaluation in his thesis [4], but had written a partial evaluation program much earlier (personal communication, December 1969). His idea at that time was to use it as an alternative to the ordinary *eval*, in order to speed up some computations.

Boyer and Strother Moore use it in a specialized prover for theorems about LISP functions [1]. Somewhat strangely, they call it an "eval" function.

Hardy uses it in a system for automatic induction of LISP functions [8].

Darlington and Burstall use it in a system for optimization of procedures, including transformations on list-processing procedures that reduce the amount of generated garbage [3].

Finally, the language ECL [14] can evaluate, during compilation, a procedure call only containing "frozen" or constant arguments. Most of these authors seem to have arrived at the idea independently of each other, and at least numbers 2 through 5 at roughly the same time.

Partial evaluation is a simple and powerful concept, and it is not surprising that so many authors have used it. It is perhaps somewhat more surprising that so few of the above-mentioned works contain references to each other. The basic partial evaluation idea is of course quite trivial, but the possibility of combining it with other optimizing operations on programs, the variety of applications, and the prospect of using it as a fairly practical tool constitute good reasons for studying it in detail.

There is one relatively abstract formulation of partial evaluation which is useful for describing some of its properties in a concise way, for example the possibility to use it to speed up itself. We simplify the description without loss of generality by considering functions of two arguments only. Let P(Q,A) be a program or procedure of two arguments, where the first argument Q is relatively fixed, and characterizes the user's present application, and A is more variable. Thus typically the user will run P repeatedly with the same Q but different A.

The "arguments" Q and A might occur as ordinary arguments or vectors of arguments to a procedure, but Q might also be stored globally in the data base and A might be read from some input medium. Such choices are not essential.

The purpose of P could also be served by a hand-tailored program for the particular application characterized by Q, i,e., a program S which satisfies.

$$S(A) = P(Q,A)$$

for all A. The potential efficiency of S and the generality of P can be simultaneously achieved by a compiler C which satisfies

$$C(Q) = S' \cong S, \qquad S'(A) = P(Q,A),$$

where the symbol $\cong$ is used for strong program equivalence, i.e., $P \cong P'$ iff $P(X) = P'(X)$ for all X,

$$C(Q)(A) = P(Q,A).$$

Writing the compiler C has one disadvantage compared to writing the general program P; it is in most cases easier to write P. The purpose of the partial evaluator is to generate S′ from P and Q, i.e., it should satisfy

$$R(P,Q)(A) = P(Q,A). \qquad (*)$$

A program equivalent to R, called REDFUN, is described in a succeeding section of this paper. Such a program has a drawback compared to C. The computation of R(P,Q) could be expected to take longer time than C(Q). This does not matter if P is only used once, but if it is used very often (as happened in some of our applications) it can be a problem. We are then computing R(P,Q) repeatedly with the same first argument and varying second arguments. This is exactly the situation that R was designed for, so it is natural to apply it to itself, computing

$$R(R,P).$$

From (*) it follows by simple substitution

$$R(R,P)(Q) = R(P,Q),$$

and therefore

$$R(R,P)(Q)(A) = R(P,Q)(A) = P(Q,A) = C(Q)(A),$$
$$R(R,P)(Q) \cong C(Q),$$

which means that in a weaker sense R(R,P) is equivalent to C. We write this as

$$R(R,P) \simeq C,$$

where $\simeq$ has the obvious definition

$$P(A) \cong P'(A) \Leftrightarrow P \simeq P'.$$

The partial evaluation with respect to R can also be carried one step further: if several P are to be treated, one might consider computing

$$R(R,R)(P)(Q)(A) = P(Q,A),$$

where R(R,R) need only be computed once, and the continued expressions (i.e., R(R,R)(P), etc.) are computed with increasing frequency.

There is a catch with this argument: it is difficult to achieve a program R which accepts arbitrary programs as its first argument, and some restrictions on the program must be assumed. In order to do R(R,P), R must be sufficiently powerful to accept itself as first argument, and at the same time R must be written in the restricted programming language so that it is an acceptable first argument to itself.

It is not obvious that this double restriction can be satisfied. In our case with the program REDFUN it was not. Rather than revise REDFUN, we decided to use the straightforward compiler method on the meta-level, i.e., to write a second program K for which

$$K(P)(Q)(A) = P(Q,A),$$

which means that

$$R(R,R)(P) = R(R,P) \simeq K(P)$$ (i.e., K can be considered as a handcoded version of R(R,R))

Such a partial evaluation compiler, REDCOMPILE, is described in Sections 7–8 of this paper.

There are some interesting problems in the context of the partial evaluator R, namely:

(a) the choice of operations that are to be performed on the argument-program P. The operations need to be both theoretically correct and practically useful.

(b) the proper ordering and interplay between these operations. They can be organized in various ways which give the same final result from R(P,Q) but with widely differing execution times in that computation.

Efficiency considerations make it very desirable in practice to have a compiler like K above. This however raises some new problems. The program R and the program generated by K (or the form K(P)) have to be able to call each other: some simplifications cannot be handled when K(P) is computed, so K has to put code into K(P) which will attempt to make that simplification when K(P)(Q) is later computed. That code can be a call to the general-purpose optimizer R. On the other hand, since R may call itself recursively, it may sometimes find itself wanting to compute R(P′,Q′), where K(P′) already exists and can be used. It is then important that the information the user provides about this program (declarations about functions, procedures for simplifying special forms, etc.) can be used by both R and K, so that the user does not have to duplicate them. In this section, we shall therefore describe how the user-provided information in REDFUN is organized.

The call to the program R has to be made slightly differently in different applications, as will become apparent from the examples in later sections. However, it is usually easy to establish the correctness of a proposed call to R. Often the user sets up a function or form, and then turns loose an optimizer which improves it. When the compiler K is used, the situation is quite different. The two-step generation process makes it much more difficult to think clearly about what operation happens when, and what information is needed when. One way to handle this problem is to first program the application using R, where one can easily convince oneself of the correctness, then re-write it using K and finally prove that the new program is equivalent to the old one, using equations which relate R and K. We shall therefore give attention to the formal characterization of R and of K.

3. The REDFUN Program

The view of the partial evaluator as a function of two arguments characterizes its overall effect, but leaves out some important details. In the actual implementation, the partial evaluator consists of a set of functions which call each other recursively, and which are similar to LISP's *eval*, *apply*, etc. Thus the first argument is a form (=evaluable piece of program), or a part of a form, the second argument is an association list of variable bindings. The central functions we call *redform* (for "reduce form"), and it is characterized by

$$eval[redform[form,l],ll] = eval[form,append[l,ll]].$$

If each binding on l is also a binding on ll, which is the typical usage, we obtain

$$eval[redform[form,]l,ll] = eval[form,ll].$$

The complementary operation on functions come in a number of different versions. Sometimes it is desirable to let the association list be a partial description of the expected binding environment of the function, i.e., contain bindings of free variables. This is done by the function *redfun*, characterized by[2]

$$redfun[\langle \text{LAMBDA}, vl, form\rangle,l] = \\ = \langle \text{LAMBDA}, vl, redform[form,rebind[vl,l]]\rangle,$$

where *rebind*[*vl*,*l*] returns a modified version of l where all bindings of values to variables in *vl*, have been deleted. Another version allows the second argument to contain information about the values of some of the lambda variables. In its very simplest version it is characterized by

$$redapply[\langle \text{LAMBDA}, vl, form\rangle l,] = \\ = \langle \text{LAMBDA}, vl, redform[form,l]\rangle,$$

where the known values of some of the arguments can be used to simplify *form*, but the opportunity to reduce the number of lambda variables is not used. Modifications which do reduce the number of arguments are trivial to implement, and the function R discussed in the previous section is one of them.

The function *redform* is the central one in the following sense; applications tend to call operations on functions, such as *redfun* or *redapply*. These functions call *redform*, and *redform* calls itself recursively. When it sees a non-atomic form, it usually applies itself to each argument in the form. In rare circumstances, such as when it encounters a free lambda expression, *redform* will call back on *redfun*-type function. Thus while *redform* is the core of the program, *redfun*-like functions are mostly used as interfaces to the applications.

The value returned from *redform* is trivial if the first argument *form* is an atom; if *form* is bound to a value v on the second argument, then a form which evaluates to v shall be returned. Usually this is $\langle \text{QUOTE}, v\rangle$ with exceptions made for special atoms such as NIL, and for numbers. If *form* is not bound in the second argument it is returned unchanged.

[2] $\langle x, y, \ldots\rangle$ is synonymous with the LISP expression *list*$[x, y, \ldots]$.

If the first argument to *redform* is a non-atomic form

$$\langle fn, arg_1, arg_2, \ldots, arg_n \rangle,$$

then a variety of operations may be contemplated.

(i) *Application of function*: If the values of all arguments arg_i and all free variables in *fn* are known (constants, or known by virtue of the second argument to *redform*), then the function *fn* could be applied to the arguments, and a constant returned as value. This assumes that the function does not have and does not depend on side-effects *during execution.* For example, the property-list access function *getp* is not in general free of side-effects, but if some properties are fixed during execution, e.g. properties which describe the application area, then for such arguments it can be considered as free of side-effects. This is important for consolidation of PRD programs. Side-effects also include input/output.

(ii) *Beta-expansion*: Suppose *fn* has a definition ⟨LAMBDA, $\langle v_1, v_2, \ldots \rangle$, *form*⟩, let *redform* evaluate to *form'*, where *form'* has been obtained from *form* by replacing v_i with arg_i for all *i*. Here arg_i is the argument expression, not the value of the argument.

This operation is always safe if the arguments arg_i are free of side-effects. It might still be safe if they have side-effects, for example if each argument is only used once in the definition of *fn*, and they occur in the right order, and there is no interaction with side-effects in the body of *fn*'s definition. On the other hand, even if there are no side-effects, there might be efficiency problems if the same, expensive computation has to be re-done several times after beta-expansion. Thus the beta-expansion has to be preceded by a non-trivial analysis and/or controlled by the user. Beta-expansion may enable simplifications of the program which would not otherwise be possible, and it is therefore sometimes important that it is done.

(iii) *Specialization of function*: Suppose some but not all arguments are constants, so that application of the function is not possible, and that considerations of side-effects or of efficiency prevent beta-expansion. It might then still be worthwhile to create a specialized version of *fn* where the available knowledge about the arguments has been incorporated. The specialized version must sometimes be given its own name, namely if it is called recursively, or if the same specialized code will be used repeatedly. Otherwise it can be inserted in the surrounding code, for example, as a free lambda expression. We shall distinguish these as *closed* and *open* specialization, respectively. Open specialization is in practice implemented to return a *prog*-expression with initialized variables.

(iv) *Functions with non-standard argument structure:* FEXPR or FSUBR functions which do not follow normal evaluation rules must be treated individually, so that *redform* is recursively applied exactly to the sub-forms, and so that variable bindings are handled properly, etc., COND, PROG, QUOTE, SETQ and SELECTQ need such special treatment, as well as user functions with non-standard argument structure.

(v) *Simplifications:* The operations performed during partial evaluation may

often result in expressions which are non-optimal in crude ways, and need to be tidied up. Two types of simplifications are easy to handle:

(a) function/argument interactions, e.g.,

(APPEND NIL *X*) → *X*

(OR *X* (QUOTE *A*) *Y*) → (OR *X* (QUOTE *A*))

(PLUS 3 (TIMES *X Y*) 4) → (PLUS 7 (TIMES *X Y*))

(b) function/function interactions, e.g.,

(CAR (LIST *X Y* (QUOTE *B*))) → *X*

(CONS *X* (LIST *Y Z*)) → (LIST *X Y Z*)

(EVAL (QUOTE *form*) AL) → *form* if AL is appropriate

Although it is easy to write a simplifier which just performs these operations, such a simplifier would not be of interest in itself, since no programmer would write the code that it knows to simplify. However, such expressions do result from partial evaluation operations, and the simplifier is therefore desirable as an integral part of the partial evaluation system.

These are the basic operations. The intricacies of side-effects make it desirable to have some additional options, such as,

(vi) *Incomplete application of functions*: If *fn* has side-effects, it may still be possible to determine the value of the form from a knowledge of some of the arguments. For example, the form

(PRINT (QUOTE (*A B C*)))

certainly could not be simplified to

(QUOTE (*A B C*))

but the environment of the form might still make use of the information that the value of the form will be (*A B C*). This could be handled by a separate case in the partial evaluator, or by letting such a form "simplify" into

(PROG2 (PRINT (QUOTE (*A B C*)))
(QUOTE (*A B C*)))

where then the procedure, which checks whether all arguments to a function are "known" will be generous enought to admit even such an argument. Incomplete application is desirable not only for system functions like *print*, but also for user functions.

It is important to realize that the operations that have been mentioned are closely inter-dependent. Each of them would be fairly useless in isolation, but together they can make fairly complex operations on a program. One reason is that some of them tend to "grow" the program that is operated upon (notably beta-expansion), some others tend to "shrink" the program (notably application of functions, and some of the operations on functions with special argument structure, such as COND); and some of them rearrange the program so that other operations can apply.

A comprehensive partial evaluation program has to make decisions about which of these operations to perform, and several of the operations require additional information about the function *fn*, in procedural or declarative form. This means that the PRD programming technique that was described in the preceding section, is appropriate here. The domain of the partial evaluator is LISP programs, and it therefore sometimes assumes that some of its code is associated with entities which occur in LISP programs, such as function names and perhaps variable names.

The REDFUN program is written in this way. It assumes that function names carry a declaration about what operations shall be performed on it. Each function shall be declared to belong to either of a number of classes (although defaults are of course available):

PURE	Basic functions without side-effects. Application of function performed if possible.
OPEN	User-defined functions (usually) for which beta-expansion is performed if arguments look good enough, and open specialization if they look less good.
REDUCED	Functions for which specialization will be performed if some argument or free variable is known. The specialization is made closed or open, depending on the circumstances.
SPECIAL	Functions with special argument structure. Such functions should have associated with them a property under the indicator REDUCER which describes how that function shall be processed by REDFUN.
LAMBDA	Functions on which no operation is performed. Functions with side-effects such as *put* and *rplaca*, and functions which depend on side-effects, such as *getp*, would typically fall in this group.

When *redform* encounters a *form* $\langle fn, arg_1, arg_2, \ldots \rangle$ and the class of *fn* is anything else than SPECIAL, *redform* will first recursively operate on each arg_i, and then operate on the whole form according to the classification. If the class of the function is SPECIAL, then the unmodified argument list will be given to the REDUCER property of the function. These five classes correspond to the first four operations mentioned above, plus the no-op case. Simplifications are handled in a second step; after beta-expansion, specialization, etc., has taken place, the resulting expression

$$\langle fn, arg_1, arg_2, \ldots, arg_n \rangle$$

is checked for possible simplifications. In principle, this is a pattern match, but for efficiency reasons it is implemented as

$$apply[getp[fn,\text{COLLAPSER}], \langle arg_1, \ldots, arg_n \rangle],$$

where the COLLAPSER property is a procedure (lambda-expression) which attempts the various simplifications. We have here an opportunity to grind our favorite thesis about PRD program organization; COLLAPSER properties are

organized in a stereotype manner, so that additional simplification rules can be stated independently to the system, and can be inserted into the correct place by a function which knows about this pre-determined structure.

Thus the user can control the operation of the REDFUN package by adequate declarations of function classes, and by writing or generating REDUCER and COLLAPSER properties for his function names. The need for interaction between the various operations performed by the program, causes some non-trivial efficiency problems. These problems also have a bearing on how the user's procedural information (such as REDUCER and COLLAPSER properties) should best be organized.

The result of beta-expansion or simplification is often a form which again may be submitted to *redform*. It might contain new calls to functions which have been declared OPEN, or forms that can be partially evaluated or simplified. Let us first consider the interaction between beta-expansion and immediate application (including the simplification of conditional expressions when the value of the if-clause is known). There are several ways of interrelating them, each of which has its drawbacks:

(a) First make a simple substitution, and then apply *redform* to the result. This has the disadvantage that code in the arguments of the expanded function, which has already been simplified as far as possible, will be repeatedly overhauled by *redform*. The problem is aggravated when we have several OPEN declared functions which call each other.

(b) First do beta-expansion as far as we can get, i.e., if one OPEN function calls another, open up all of them and then partially evaluate the resulting expression. This has the disadvantage that the opened-up expression may be very large, so that large chunks of code are generated by beta-expansion and then thrown away by the recursive call to *redform*. This happens particularly when several of the OPEN functions contain conditional expressions, where all but one of the branches are supposed to go away because the variables in the *if* clause have known values.

The obvious solution is to combine substitution and partial evaluation into one procedure, i.e., to do the substitution involved in beta-expansion together with immediate application, in one single scan through the form. With this method, we would need a *super*-redform function which takes three arguments: a form to be simplified, an association list which binds variables to known values, and an association list which binds variables to forms which shall be substituted for them. (One could of course incorporate the second argument in the third, at the cost of some extra processings.)

We preferred another method, which is a slight modification of alternative (b), namely to have a beta-expander which scans a function definition and does the following operations:

(i) substitutes argument forms ("actual parameters") for variables ("formal parameters");

(ii) calls itself recursively when it encounters functions which have been declared OPEN;

(iii) remembers which variables have which known values on entry to the beta-expansion, and which of these bindings are hidden further down because the variable has been re-bound;

(iv) whenever it encounters a conditional expression, calls *redform* recursively on the *if*-clauses, in an attempt to throw away either the *then*-clause or the *else*-clause already during beta-expansion.

This was considerably easier to implement, and eliminated most of the waste that occurred when method (b) was used.

Consider next the values returned by the user-provided COLLAPSER properties, which do some simplifications. Sometimes the returned value can be further simplified. For example, in order to simplify

(APPEND (APPEND *X Y*) (APPEND *V W*))

into

(APPEND *X Y V W*)

the COLLAPSER of APPEND may have to act twice. Also, after the transformation

(APPLY* (FUNCTION FOO) *X Y Z*) → (FOO *X Y Z*),

redform should check what function class *foo* is in, and what operations may be appropriate on the new form. In both of these examples, the desired result could be achieved by re-applying *redform* to the value of the COLLAPSER function, whenever it has done some simplification. However, this solution is very expensive, since the arguments (*X*, *Y*, etc.) would then recursively be given to *redform*, and they may be big expressions. In fact, if *redform* is systematically re-applied to the values from all COLLAPSER'S and REDUCER's (where the problem is similar), then the execution time will grow very quickly with the depth of *form*, in the manner which is characteristic of double recursion. What is needed, of course, is a way to apply some but not all of the operations of *redform* to the values from REDUCER's and COLLAPSER's.

One immediate problem then is to what extent the REDFUN program can determine which operations have to be done in what situations, and to what extent the user has to specify this (e.g. by including explicit requests for post-processing in the simplification procedures he provides). Regardless of how that problem is solved, the problem remains of how the right subset of operations can be accessed, if they are integrated and buried in procedures. In particular, it is possible that the present user procedures in REDFUN, i.e., REDUCER's and COLLAPSER's, are too big units, and that they should be broken down into several smaller procedures for each function name, where each procedure performs one specific operation, or else that these user procedures shall be parametrized with respect to which operations they shall perform in each call. However, such mechanisms also cost a substantial overhead.

There is another solution to the problem which at first seems wasteful, but which in fact has some advantages. One can leave the values from REDUCER's and COLLAPSER's as they are, which means that sometimes the simplification is not finished, but instead make a loop on the top level of *redfun*, of the form

$$fn := redfun[fn,l]$$

and iterate until *fn* converges. This method is guaranteed not to miss any simplifications, if it converges; it avoids the double recursion, and it also avoids the need for and cost of complex control of the various operations. This is the method we have used. We typically needed about three cycles for fully optimizing the compiled PCDB axioms (as described in Section 5), and less in the other experiments.

Another place where the efficiency of *redform* could be increased was in partial evaluation of conditional expressions. Given an expression

$$(\text{COND } (p_1\ e_1) \ldots).$$

redform should of course first partially evaluate p_1, and if it comes back as a constant, decide to partially evaluate only e_1, or only the rest of the expression, depending on what the constant is. Actually, it is the REDUCER property of COND which does this. Suppose now that each of the p_i are well-sized expressions, and that all the variables in them are known. *Redform* will then be doing an operation which *eval* could have done just as well, and much quicker.

One way to tip *redform* about this is to modify the conditional expression to read

$$(\text{CONDVAR } (v_1\ v_2 \ldots)(p_1\ e_1) \ldots),$$

where the first "argument" to CONDVAR is a list of all the variables which occur in any of the p_i. The REDUCER property of CONDVAR first checks if all the variables in the first "argument" have known values. If so, it runs down the list of conditions doing ordinary *eval*'s, if not, it calls the REDUCER of COND with the appropriate arguments. The transformation from a COND-expression to a CONDVAR-expression could be done by the programmer who wants to put in some work in order to speed up his program, but it would be easy to write a program which makes the transformation automatically when it is safe to do so (i.e., when it can understand all auxiliary functions called from the p_i). The beta-expander that was described above also knows how to handle CONDVAR expressions efficiently. This simple trick speeded up execution times considerably.

A minor observation was that performances improved significantly when expressions of the form

$$(\text{QUOTE } xxx)$$

which of course shall not be changed, were checked for directly in the body of *redform* rather than treated by a REDUCER property for QUOTE.

In one of the experiments, the one with the GIP/GUP program, some problems occurred, which were of a more general nature. They will be discussed here, although the solutions to the problems were not made general, but were made to handle the difficulties, which arose in the GIP/GUP program.

3.1. Assignments to variables

Sometimes one argument to a procedure is a list or other structure of several parameters. It is then tempting for the programmer to write something like

```
(LAMBDA (X Y) (PROG (P1 P2 P3)
  (SETQ P1 (CAR X))
  (SETQ P2 (CADR X))
  (SETQ P3 (CADDR X))
  ...))
```

where the *Pi* should be mnemonic names for the various parameters. If the first argument *X* is known by virtue of the call, then of course the *Pi* shall be known also. This case is not covered by other rules about when variables are considered as "known".

If the assignment appears inside some conditional expression as COND, OR, AND, SELECTQ, there arises some problems. Consider for example

```
(AND Y (SETQ L (REVERSE Y)) (FOO L)).
```

If the value of *Y* is unknown to REDFUN, it cannot be sure of the value of *L* after encountering this expression. Inside the expression the value of *L* would still be known. So the essential problem is to keep track of entries and exits in conditional expressions. Of course, if the assignment occurs in the first condition in a conditional expression (or if there are conditionals inside conditionals with assignments on different levels) the task of keeping track of assignments becomes quite complicated.

One way to solve this problem would be to let REDFUN generate code for every branch. Since there are usually several branches in ordinary code, this solution was dropped because of the growth of the code.

We have therefore chosen to let REDFUN always consider the value of *L* as unknown *after* a conditional expression. *Inside* an *and*-expression, however, the value is known after the assignment, i.e., the call to *foo* could be replaced by either a simplified definition of *foo* or by a call to a generated function with this definition. This partial solution was sufficient for the experiments.

The implementation of these additional facilities involved updating the second argument in *redform*, i.e., the association-list of bindings of known variables. This was no problem. There did arise a problem, however, as to where the knowledge about *setq*'s in the argument of *and* (or other conditionals) should be located. It would seem natural to put this information in the REDUCER procedure of the function *setq*. (The REDUCER procedure if it exists, suppresses all actions normally taken by redform, and specifies how a form is to be treated by *redform*.) However, this is not possible, since a REDUCER procedure only has access to the local form that is to be simplified, not to its environment in the program. The alternative would be to put this operation into the REDUCER of the function *and*. However, this would force that REDUCER (and those of other conditionals) to make an extra search through their arguments on all levels, looking for assign-

ments. What we really need, of course, is a way for the *setq*-reducer to send a message to the reducer of any conditional function which appears earlier in the recursion. In the present experiment, this was done with a fix involving free variables, but in revised versions of REDFUN a more systematic method should be found.

3.2. Prog-expressions

The trivial way to partially evaluate a *prog*-expression is to partially evaluate the forms in the expression separately. If there are labels and *goto*'s in the *prog*, then one might desire to optimize the *goto*-structure. Also, the analysis of situations where variables become known by virtue of assignments, as discussed above, are fairly simple in straight code, but much more difficult to analyze when *goto*'s are involved. In the present experiment, it was sufficient to partially evaluate each form in the *prog*-expression separately, and to restrict the part of the *prog*-expression which preceded the first label or *goto*.

The REDFUN program is quite straightforward and does only fairly simple operations on its argument program. It is certainly possible with todays A.I. methodology to write more sophisticated programs, which can optimize more, and are less dependent on control information from the user. However, smarter programs will not necessarily be more useful, namely if they take longer to run. REDFUN is already almost too slow to be used, and it was only after RED-COMPILE was implemented (see Sections 7–8) that the running time became acceptable to the user. The intelligence that can be incorporated in a program of this type, does not seem to be worth its cost yet.

4. Generating Code From Parameters in the PCDB Application

PCDB [7, 10, 12] is a program generator which uses first-order predicate calculus (with some procedure-oriented extensions) as its high-level language. The generated code consists mainly of storage and retrieval procedures for relations and functions in predicate calculus. When the generated code is executed, it is located in the "P" part of an PRD-type system, so the PCDB code generator knows where to store the code. The structure of the whole system is characterized by Fig. 1. Single arrows indicate procedure calls; double arrows data flow.

The REDFUN program, described in the previous section, was developed in conjunction with PCDB. We found that the problem of specializing and optimizing code was a general problem, so instead of including this part in PCDB it was separated as the REDFUN package.

The data base in PCDB consists of assertions, represented as ground-unit clauses in PC, for example

OLDER (JOHN, FATHER(MARY))
PRIME (5)
DISTANCE (STOCKHOLM, NEW-YORK, 2000*KILOMETERS)

Every relation will now have a number of procedures associated with it. They fall into two groups, based on how they were generated. The *direct* storage and retrieval procedures are procedures which can store a relation explicitly in the data base, or look up a relation which has been thus stored. They implement conventions about where and how each relation shall be stored, and can be generated from declarative formulations of those conventions. The *deductive* procedures, on the other hand, perform deduction when a relation is stored in the data base, searched for in the data base, etc. The deductive procedures typically consist of a collection of "methods", each of which has been generated from one axiom provided by the user. The supervisor for the search implied by these methods resides in the "S" part of the system. Functions in predicate calculus are handled in much the same way as relations.

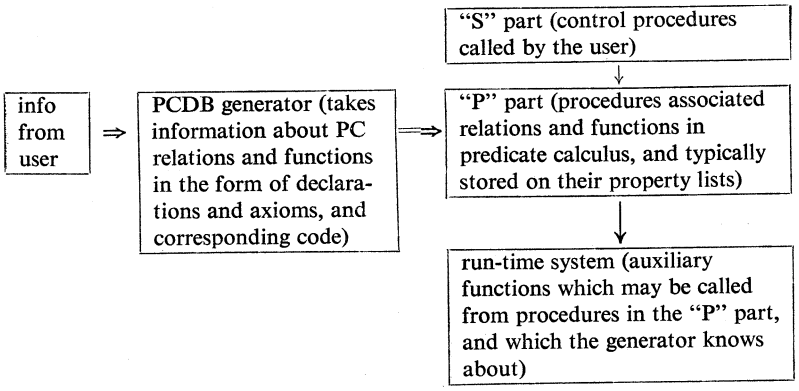

FIG. 1.

The partial evaluation programs are used both for generating the direct procedures from declarative information, and for compiling axioms, but in fairly different ways for the two purposes. We shall devote one section to each.

Each relation R is associated with a number of features $d_1(R)$, $d_2(R)$, . . ., where the possible d_i include the number of arguments, number of proper arguments,[3] the datatype(s) of the argument, the "one-manyness" between the arguments etc. If this information is not given by the user, it is calculated by default from some of the other d_i and/or from axioms that the user has provided.

At run-time, each relation R must be associated with a number of direct procedures $f_1(R)$, $f_2(R)$ for immediate storage, retrieval, etc. The program generation task is then to generate the f_i from the d_i's. In principle there is one generation function F_i for each f_i. These generators make use of a technique which seems to be generally applicable when parameters are to be mapped into code, namely to go through FUNARG-expressions.

[3] To be able to answer an open question at least one proper argument must be given. For example, if the relation Age(x,y) has x as a proper argument, the system can answer Age(x,?), but not Age(?,y).

Let us take a very trivial example. Suppose we want to write a program (LAMBDA (*R*) . . .) which generates the appropriate search procedure for the binary relation *R*, where *R* might or might not be transitive, depending on the presence of a flag on the relation's property list. (In PCDB, a direct retrieval routine would not normally do a search on transitivity, but that is not important for the example.) If *R* has been labelled as transitive, then the function definition for some *R* e.g., PARTOF should be

```
(LAMBDA (A B)
  (SOME (GETP A 'PARTOF)
    (FUNCTION (LAMBDA (X) (PARTOF X B]
```

which will continue the search according to the transitivity; otherwise it should be

```
(LAMBDA (A B) NIL].
```

The functions *some* and *getp*, are used as in INTERLISP [13]. The generator could then be written in a straightforward way as follows:

```
(LAMBDA (R)
  (PUTD R (COND [(GETP R 'TRANSITIVE)
                 (LIST 'LAMBDA
                       (LIST 'A 'B)
                       (LIST 'SOME
                             (LIST 'GETP 'A (KWOTE R))
                             (LIST 'FUNCTION
                                   (LIST 'LAMBDA
                                         (LIST 'X)
                                         (LIST R 'X 'B]
                (T (LIST 'LAMBDA
                         (LIST 'A 'B)
                         NIL]
```

This is not very legible, and nobody would use this method for long. One way out is to use a sub-system for patterns; this is certainly sufficient for this simple example. However, there is an interesting method, where one writes simply (in INTERLISP)

```
(LAMBDA (R)
  (PUTD R (OPTIMIZE
            [FUNCTION (LAMBDA (A B)
                        (COND [(GETP R 'TRANSITIVE)
                               (SOME (GETP A R)
                                     (FUNCTION (LAMBDA (X)
                                                 (APPLY* R X B]
                              (T NIL)))
                        (R]
            )))
```

thus maximally mixing the generation-time variable R and the execution-time variables A, B and X. The outer *function* expression has two arguments. The first argument is a *general procedure* which should be specialized in order to obtain the desired procedure for the relation R. The second argument is the list of *transfer variables*, i.e., those variables which were bound at generation time, and whose values shall be retained until execution time. In this example R is the only transfer variable. Its value is kept in a FUNARG-expression; the outer *function* expression evaluates into an expression of the form

(FUNARG [LAMBDA (A B) . . .] ((R . PARTOF)))

if the generator has been called with the relation name PARTOF as its argument. The third element in the FUNARG-expression is an association list or an equivalent array. This method of handling the FUNARG problem was proposed in [11] and has subsequently been incorporated into INTERLISP.

The function *optimize* may be the identity function, and the relation R will then be associated with a piece of code which is a FUNARG-expression. Some LISP interpreters can handle such functions, and in the others it can be simulated. However, the code will of course run slowly, since it has to look up the TRANSITIVE property each time. If there are multiple parameters d_i, and some of them have to be computed by default, the overhead will increase. It is then appropriate to define *optimize* so that it will go into its argument, insert the known value of R in all places, look up the TRANSITIVE property (since it is presumably to be considered as fixed), make the right choice in the conditional expression, etc. All of this is typical *redfun* operations, and it is sufficient to define

optimize[*fn*] = *redfun*[*fn*,NIL],

redfun[⟨FUNARG, *fn*, *l*⟩,*ll*] = *redfun*[*fn*,*append*[*l*,*ll*]].

In the very elementary example that was given here, the strength of the FUNARG method is not fully needed. In more complex examples, its advantages become more evident. Basically, the advantage is that the user does not have to handle generation-time and evaluation-time variables differently; the distinction is taken care of by the subsequent optimization.

The fact that the user's code goes through a partial evaluator which also is an optimizer, is beneficial in several ways. It relieves the user of most of the concern for generating smart code. The amount of optimization can be adapted to the situation; in the testing stage one can choose to do little or no optimization (i.e., keep the original FUNARG-expressions); in a latter stage one can optimize more. An additional advantage became apparent when the PCDB system was translated from LISP1.5 to the INTERLISP dialect of LISP. It is usually difficult to make dialect conversation of program generators, since it is not sufficient to modify the generator; one also has to change it so that the generated code comes out in the new dialect. But since the generator operated through optimization of FUNARG-expressions, that consideration went away. As soon as all LISP code had been converted, including the general procedures in the generators, both the generator and the

generated code ran in the new dialect. (The task of converting *redfun* to the new dialect was of course less trivial, but just had to be done once.)

The straightforward scheme that has been outlined here, was compromised in two ways in the actual PCDB system. First, it was noticed that relations could be divided into groups, where the appropriate procedures for relations in different groups differed radically, whereas relations in the same group had procedures which were fairly similar, although they should still differ in their details depending on the remaining parameters d_i. One parameter d_0 was therefore chosen as relation type, and one set of generators was associated with each relation type. Thus each generator F_i has the form

(LAMBDA (*R*) (APPLY* (GETDFLT (GETDFLT *R* 'D_0) 'DEFINER$_j$) *R*))

where *getdflt* is like *getp* except that it can also look up default values and such. Thus we could have

PARTOF: D_0 = BINREL (a hypothetical relation group for binary relations),

BINREL:DEFINER$_j$ = (LAMBDA (*R*) (PUTD *R* (OPTIMIZE . . .))),

i.e., BINREL carries the real program generator. This organization makes it easy to add new classes of relations to the generators, which has turned out to be very useful. The other modifications to the original scheme was that, since it was somewhat inconvenient to loop up the declared properties during the partial evaluation, we did not actually do

```
(PUTD R (OPTIMIZE (FUNCTION (LAMBDA (X Y) . . . )
                            (R) )))
```

in the generator, but rather

```
(PROG (D1 D2 . . .)
      (SETQ D1 . . .)
      (SETQ D2 . . .)
      . . .
      (PUTD R (OPTIMIZE (FUNCTION (LAMBDA (X Y) . . . )
                                  (R D1 D2 . . .) ))))),
```

where the body of the LAMBDA-expression in the latter case uses the variables *D*1, *D*2, etc., whenever the original LAMBDA-body contains code that will compute those parameters. (In practice, mnemonic names are of course used.)

Apart from these superficial deviations, the general method has been directly applied, and worked well. The general procedures typically consist of calls that lead to a hierarchy of auxiliary functions. Some of them are opened up by *redfun*, and a few are allowed to remain and form a run-time system for the generated code. However, it is striking how much expansion and simplification can be made by the partial evaluator, and how big the difference is between the general procedure in the program generator, and the generated code.

5. Compilation of Axioms in the PCDB Application

The axiom compiler takes as input a clause written in implication form, for example,

$$R(x,y) \wedge S(z,y) \wedge P(x) \supset T(x,z),$$
$$R(x,y) \wedge R(y,z) \supset R(x,z),$$
$$R(o,w(c,a),s) \supset S(g(c,o,a),s),$$
$$R(o,f(p,o'),s) \wedge M(p) \wedge X(o,o') \supset R(o',f(n(p),o),s).$$

The latter two axioms are taken from an application. The result of the compiled axiom shall be a procedure which accepts a sub-question and generates a set of sub-sub-questions. For example, the transitivity axiom should compile into a piece of code of the form

(LAMBDA (X Z) (. . . retrieve possible y and return sub-sub-question $\langle R\ y\ z\rangle$ for every such y . . .)).

In practice, the procedure should take additional arguments which control the depth and direction of search, and which specify the procedure which is to be applied to sub-sub-questions. Also, one and the same axiom can be compiled in several ways, depending on whether it is to be used for forward (IF-USED) or backward (IF-NEEDED) deduction, and in the latter case also depending on whether it is used for closed questions (answerable by truthvalue) or open questions (of the form "which z satisfy $R(x, z)$?"). All of this has been described in the above mentioned references and is not significant for the present discussion.

The obvious way to compile axioms when a partial evaluator is available, is to write an axiom interpreter P(axiom,sub-question), and partially evaluate it with respect to its first argument. The resulting procedure takes one argument which is the sub-question. One then has to modify its lambda-variables so that they correspond to the arguments of the relation. But more important, such an interpreter P could not simply make a loop over the literals to the left of the implication sign and process them one at a time, since the operations on each literal depend on which variables end expressions occur in surrounding literals. For example, the literal $R(o,f(p,o'),s)$ should be processed differently, depending not only on which of the variables o, p, o', and s have occurred in previous literals or not, but also depending on whether $f(p, o')$ has previously occurred. Also, if this expression has not occurred in previous literals, its value shall be computed, and the value shall or shall not be saved, depending on whether the expression occurs in a later literal.

Although it would certainly be possible to write an axiom interpreter which handled all of this correctly and which could be partially evaluated, such an interpreter would not necessarily be transparent. We therefore preferred to have one program which converts the part of the axiom to the left of the implication sign, to a *prog*-expression consisting of calls to certain procedures called *bind* and *cont*. They are two general procedures for accessing the data base and for maintaining the result(s) from the access. The arguments of the procedures will have

known values, so the whole *prog*-expression is then given to *redform* which does beta-expansion and partial evaluation.

After having written the compiler in this fashion, we considered again whether there would be any gain in re-writing it as an axiom interpreter. For practical purposes, the answer seems to be no; it would not become significantly more transparent, and it would run slower. Re-writing it as an interpreter might however be useful if one wants to prove its correctness. But even with the present compiler, partial evaluation plays an important part; a compiler whose latter stages do explicit code generation, instead of calling *redform* with a *prog*-expression as argument, would be much more difficult to maintain than the present program.

To complete the picture, let us mention where the code from the compiled axiom goes. At run-time, each relation symbol is associated with several deductive procedures, each of which may contain the compiled code from several axioms. There is of course an insertion procedure which knows how the deductive procedures are organized, and which can insert the compiled axiom in the right place. The insertion procedure is sensible to advise from the user.

Since REDFUN was developed in conjunction with PCDB, one would expect REDFUN to be fairly much tailored to its specific needs. Let us however mention one point where early versions of REDFUN were not sufficient, and had to be improved: It used to be that the OPEN declaration merely implied that beta-expansion should be done if possible, and open specialization was not made. This led to problems in several cases.

In the simplest example, the general procedures for storage, call an auxiliary function defined as

```
(LAMBDA (ROOT B)
  (COND  ((NULL (CAR ROOT))(RPLACA ROOT B))
         ((EQ (CAR ROOT) B) T)
         (T NIL)))
```

which attempts to put a new fact in the data base, and checks that nothing which contradicts the new fact was stored there before. The auxiliary function is called with a first argument on the form

(GETR *S A*)

where *getr* is a property list access function characterized by

$$car[getr[a,i]] = getp[a,i].$$

In this case it is not satisfactory to make a beta-expansion on the auxiliary function, since the property list access will then always be performed twice. The problem was temporarily solved by re-writing the auxiliary function as

```
(LAMBDA (ROOT B)
  (PROG  (RT)
         (SETQ RT ROOT)
         (RETURN (COND ((NULL (CAR RT)) . . . )
                       ((EQ . . . ) . . . )
                       (T . . . ) ))))).
```

This is only a provisional solution since in this way the user again has to think about code generation problems. The definitive solution was to improve *redfun* so that functions which are declared OPEN, like the auxiliary function in this example, are treated with open specialization when they get this kind of argument.

The reader will already have noticed that the same problem is also faced by an ordinary LISP compiler, and that some compilers perform services like open specialization. In the particular and very simple example that was mentioned here, such compiler facilities would suffice. In more complex cases, specialization enables other operations on the program, and then it is useful to have it done by a separate LISP-level optimizer.

In summary, PCDB was the first application of REDFUN and provided most of the ideas for how it should be designed. The only problem which could not be solved within the REDFUN framework was that of execution time. That problem has however been solved with the REDCOMPILE program which is described in Sections 7–8.

6. Specialization of a Utility Program in the GIP/GUP Application

Although the REDFUN program worked quite well for the PCDB application, the conclusions that one could draw from that were limited since PCDB had been able to influence REDFUN. In order to gain better experience with REDFUN, a second experiment was designed as follows:

One of the authors (Oskarsson) wrote a program of the "general procedure" type for input and output of property list information. This program was written without thinking of the restrictions of REDFUN, and in fact without knowledge of that program. After the program was completed we attempted to apply REDFUN to it as a "general compiler". This experiment shall now be described.

The GIP/GUP package is intended for parametized input and output of property lists. There is one function (GIP) for input and one (GUP) for output. The functions read and print sets of property list properties. They can also recursively operate on atoms which are given or found as properties under given indicators.

For example, if the function *gip* is called with one argument, which should be a free property list, it could read input on the form

```
CAIN: FATHER = ADAM;
        MOTHER = EVE: HUSBAND = ADAM;
BROTHERS = ABEL /.
```

This input from the user will assign the following property list:

```
to CAIN:  (FATHER ADAM MOTHER EVE BROTHERS (ABEL)),
to EVE:   (HUSBAND ADAM SONS (CAIN ABEL)).
```

Similarly, if the function *gup* is called as

```
(GUP free-property-list '(CAIN))
```

it will print out

```
CAIN:
  FATHER = ADAM
  MOTHER =
    EVE:
      HUSBAND = ADAM;
      SONS = CAIN, ABEL;;
  BROTHERS = ABEL;
```

Usually of course the second argument to *gup* is a long list of atoms, and *gip* is used to read larger sets of properties. The free property lists which are arguments to GIP and GUP contain information about how to store and retrieve properties and what layout is desired. There are 28 different indicators which can be used in this list. Almost all have default values attached to them. Normally the user is expected to give just a few of these indicators himself, but he has the possibility of giving different specifications for almost every place in the recursive input or output.

Such highly parametized programs are of course quite flexible, but have the disadvantage that they must often consult the free property list. Almost every time they will find nothing and must therefore go out and search the list of default values. If one uses the program several times with the same free property list as argument, he will find the program unduly slow. One would like to partially evaluate the program with respect to this parameter. This was considered a typical problem for REDFUN.

In a preliminary test, one function in GUP was simplified manually, and a 50% reduction in execution time was obtained. This encouraged us to apply REDFUN to the whole package. This problem is an archetype specialization, i.e., with knowledge of the parameter property list, *gip* can be reduced from one to zero arguments, and *gup* from two to one arguments, and their definitions can be specialized. The function *specialize* for doing this is defined by

specialize[⟨LAMBDA, *vl*, *form*⟩, *arg*] =
= ⟨LAMBDA, *cdr*[*vl*], *redform*[*form*,*list*[*cons*[*car*[*vl*],*arg*]]]⟩.

6.1. Simplification through several levels of functions

Like the "general procedures" in PCDE, the function *gup* calls a number of auxiliary functions. In the PCDB case, such functions should be beta-expanded, i.e., they were declared OPEN. In the GUP case, this was not appropriate because of side-effects, and closed reduction was desired instead.

Fig. 2 indicates the calling structure of *gup*. Marked boxes indicate functions where the most can be gained by reduction. In order to simplify an auxiliary function, REDFUN must first analyze the functions that call it, in order to determine what arguments are known and what their values are. It was only when the need for closed reduction was experienced in the GIP/GUP experiment, that

the REDUCED option in REDFUN was introduced. A function with this declaration will be simplified when a call to it occurs, and the simplified version of its definition is stored separately as a new function, and a call to it is inserted in the code. Also, REDFUN keeps track of what specialized functions it creates, so that if the same specialization is later used again, it does not have to create a new copy, but can make a second call to the existing specialization.

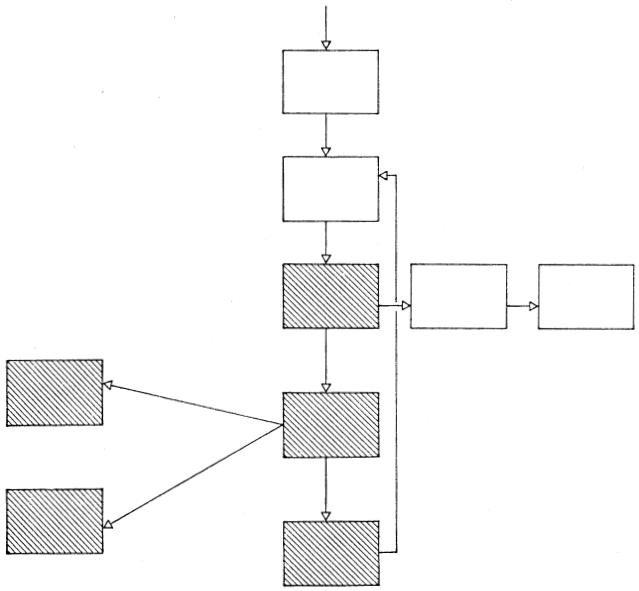

FIG. 2.

6.2. The effect of REDFUN on the function XTPR in the GIP/GUP-package

XTPR is declared REDUCED, REALIND and REALPROP are declared OPEN and TAKEIN, TAKENDL and TAKEFL are declared PURE. XTPR/F1 is the new version of XTPR generated by REDFUN. The function OUTIND has been reduced to OUTIND/F1, but is not shown here. It uses however IDTI, IDTP and CARRTERM as free variables, which are bound in XTPR/F1.

Before:

```
(REALIND [LAMBDA (I)
                (COND ((GETL ILI (QUOTE INTNAME)))
                      (T I)),
(REALPROP
  [LAMBDA (I)
    (COND ((TAKEFL FLI (QUOTE FUNC))
           (APPLY* I A))
          (T (GETP A I)),
```

```
(XTPR
  [LAMBDA (IX)
    (PROG (P ILI FLI CARRTERM I SAVE OUTFUNC IDTI IDTP)
          (SETQ ILI (CDR IX))
          (SETQ IX (CAR IX))
          [COND ((NULL (ATOM (CAR ILI)))
                 (SETQ FLI (CAR ILI))
                 (SETQ ILI (CDR ILI]
          (SETQ OUTFUNC (TAKEIN ILI (QUOTE OUTFUNC)))
          (SETQ I (REALIND IX))
          (SETQ P (REALPROP I))
          [COND (OUTFUNC (SETQ P (APPLY* OUTFUNC P]
          (COND ((NULL P) (RETURN)))
          (SETQ IDTI (PLUS (TAKEIN ILI (QUOTE INDENTI))
                           IDTC))
          (SETQ IDTP (PLUS(TAKEIN ILI (QUOTE INDENTP))
                          IDTI))
          (SETQ CARRTERM (TAKENDL ILI (QUOTE CARRTERM)))
          (OUTIND I P (TAKEFL FLI (QUOTE VERT)))
          [COND ((SETQ SAVE (TAKEIN ILI (QUOTE SAVE)))
                 (RPLACA SAVE (APPEND (CAR SAVE)
                                      (DTST P]
          (RETURN]).
```

After:

```
(XTPR/F1
  [LAMBDA NIL
    (PROG (P CARRTERM IDTI IDTP)
          (SETQ P (GETP A (QUOTE RELATION)))
          (COND ((NULL P) (RETURN)))
          (SETQ IDTI 2)
          (SETQ IDTP 4)
          (SETQ CARRTERM (QUOTE =))
          (OUTIND/F1 NIL P)
          (RETURN]).
```

7. Modes of Using the Partial-Evaluation Compiler

The purpose of the REDCOMPILE program was specified in Section 2, namely, to speed up the optimization process when R(P,Q) has to be computed repeatedly for the same P but different Q. In principle, REDCOMPILE is a program K such that

$$K(P)(Q) \cong R(P,Q).$$

We shall henceforth refer to the computation of K(P) as *compilation*, to the computation of P′(*Q*) where P′ = K(P) as *compiled generation*, and to the computation of R(P,Q) as *interpreted generation*. In all examples that we have run, compiled generation has been considerably faster than interpreted generation. Compilation takes roughly as much time as interpreted generation, but only has to be performed once for each P.

The operator K which has been specified above describes the principle of the REDCOMPILE program, but the actual program deviates on some points. First, we actually need REDCOMPILE for two different though related purposes. One is for specializing functions, as K does and the other is for beta-expansion. One needs a function L which takes the definition of a function *foo* and generates a function *foo′* which can be applied to the actual argument list of *foo*, to generate an appropriate form to be inserted. L should satisfy

$$P \circ l \cong L(P) * l,$$

where $\cong$ in this case is an equivalence between *forms*, where l is an argument list, and where $\circ$ is an infix operator for LISP's *cons* function and * for the *apply* function.

Although it is intuitively clear that the functions K and L are related, it is not easy to express one in terms of the other. This can be done better if we first introduce the exact conventions for the compiler. Just like *redfun*, the function *redcompile* must actually be provided with two arguments. The first argument is a lambda-expression (or an atom whose function definition is a lambda-expression); the second argument is a list of atoms, each of which should occur in the first argument, either as a lambda-variable, or as a free variable. The second argument informs *redcompile* that this variable is to be considered as known in the following sense: if it is a lambda-variable, then the corresponding argument in 1 will be a constant or can be reduced to a constant at generation time (it may be a variable whose value is known, or an expression consisting only of constants and of functions which can be evaluated at simplification time). If it is a free variable, its value in the given environment will be known at generation time. (This applies both to compiled and interpreted generation. We define those concepts similarly for L as for K.)

The list of parameter variables is particularly useful in cases like the following: Suppose L is applied to the simple function definition

(LAMBDA (*X Y*) (COND (*X* (CONS *X Y*)) (*T Y*))).

If *X* is not a parameter, L could generate

```
(LAMBDA . . . (LIST 'COND
                    (LIST (GETARG X)
                          (LIST 'CONS (GETARG X) (GETARG Y)))
                    (LIST T (GETARG Y]
```

whereas if it is a parameter, L could generate

```
(LAMBDA . . . (COND
                ((ARGVAL X)(LIST 'CONS (GETARG X)(GETARG Y)))
                (T (GETARG Y].
```

In the former case, the execution of the conditional is postponed to execution time (of the generated program), in the latter case the test is performed at generation time.

There exist some special cases, however, where we cannot perform the simplifications until the second step of the process, i.e., when we know the values of the variables. We can consider a conditional expression with both known and unknown conditions. The strategy mentioned above will not be powerful enough to deal with this, since we want the (LIST 'COND-version, but may want to omit some or other branches, depending on the values of the known variables. This type of simplification is significantly more difficult to handle in *redcompile* than in *redfun*. In practical use, we have found only very few cases where it is necessary, and handled them by using *redform* as a post-processor to code generated through *redcompile*.

Consider the case where the definition of a function G calls another function H, and L is applied to both of these. When L(G) is computed, L can analyze the argument list for H in each call, and determine which arguments will be "known". This information need therefore be specified by the user on the top level only.

In such a case, L(G) must contain a call to the saved value of L(H). Some efficiency is gained by adopting the following somewhat complex calling conventions for L(H): if H has a definition (LAMBDA (*v*1 *v*2 . . . *vk*) *a*), then L(H) shall have the form (LAMBDA (*v*1 *v*2 . . . *vk*) *a'*), where the same variable names are used. L(H) assumes that all arguments which are provided for known lambda variables should come as their known values, whereas other arguments will come as forms. For example, if the definition of G contains the sub-expression

(H *A B*)

and neither *A* nor *B* are known in G, then L(G) will in the corresponding position contain

(H* '*A* '*B*),

where H* is defined as L(H). If *A* is known, L(G) will instead contain

(H* *A* '*B*).

This convention facilitates particularly the case where known values are passed on from one auxiliary function to the next, and all of them are to be compiled. L(H) must of course be designed to treat its arguments appropriately and according to this convention. Using it, the relationship between REDFUN and REDCOMPILE is:

$$\begin{aligned} &\mathit{redform}[\mathit{lx} \circ \mathit{argl},\mathit{al}] = \\ &\quad = \mathit{redcompile}[\mathit{lx},\mathit{vl}] * \mathit{prepev}[\mathit{argl},\mathit{al},\mathit{mask}[\mathit{cadr}[\mathit{lx}],\mathit{vl}]] \end{aligned} \tag{1}$$

if all variables in the list *vl* are bound in *al*. Here *redcompile* is the function L with the second argument *vl* added, and with the special calling convention for the generated code implied by *prepev*. The asterisk stands for *apply* in an arbitrary binding environment, and the auxiliary functions *prepev* and *mask* are defined by:

$$mask[ll,vl] = \textit{if } null[ll] \textit{ then } \text{NIL} \quad \textit{else } cons[memb[car[ll],vl], mask[cdr[ll],vl]], \tag{1a}$$

$$prepev[argl,al,m] = \textit{if } null[argl] \textit{ then } \text{NIL} \quad \textit{else } cons[\textit{if } car[m] \textit{ then } eval[car[argl],al] \textit{ else } car[argl]], prepev[cdr[argl],al,cdr[m]]]. \tag{1b}$$

It has been our experience that in each application of REDCOMPILE, it was necessary to hand-tailor the interface from the generator to REDCOMPILE. Furthermore, the two levels of program generation involved make it very easy to make mistakes in using *redcompile*. It is therefore desirable to prove that the usage is correct. The proofs can be worked out using the known (or at least assumed) relation between *redfun* and *redcompile*. We shall illustrate this with two examples.

The first example is for the generation of direct storage and retrieval routines in PCDB, as described in Section 4. Each generator takes a relation R as argument, and essentially has the form

⟨LAMBDA, ⟨R⟩, ⟨REDFUN, ⟨FUNCTION, ⟨LAMBDA, *ll*, *a*⟩, *vl*⟩⟩⟩

with different *ll*, *a*, and *vl* in different generators. The function body *a* typically contains calls to other functions. REDFUN beta expands these functions several levels down. When REDCOMPILE is used, it must be applied both to the auxiliary functions, and to the explicit argument of REDFUN in the generator. The former operation is the archetype use of *redcompile*, and is trivial once the list of known variables is available as a side-effect of the latter mentioned operation. In order to speed-up the generator, it is changed into the following expression:

⟨LAMBDA, ⟨R⟩, ⟨LIST, 'LAMBDA, '*ll*, *ac*⟩⟩,

where *ac* is a form which is computed as

$$ac = redcompform[a,vl],$$

where *redcompform* is an auxiliary function to *redcompile*, and is characterized by:

$$redcompile[\langle \text{LAMBDA}, ll, a\rangle, vl] = \langle \text{LAMBDA}, ll, redcompform[a,vl]\rangle. \tag{2}$$

The equivalence between the old and the new generator is proven as follows. Let the generation-time binding environments be the association-list *e*. Introduce a function *funct*[*fn*,*vl*,*al*] which corresponds to the LISP pseudo-function FUNCTION but where *funct* is really a function of its arguments, and where its third argument explicitly specifies its binding environment. We have

$$funct[fn,vl,al] = \langle \text{FUNARG}, fn, pair[vl,evlis[vl,al]]\rangle. \tag{3}$$

We have (1), (1a), (1b) and (2) as premises, and also need a part of the definition of *redfun*:

$$redfun[\langle \text{LAMBDA}, ll, a\rangle, al] = \langle \text{LAMBDA}, ll, redform[a,al]\rangle \tag{4}$$

and the rule for introducing LAMBDA-variables (which is a consequence of the definition of *eval*):

if *ll* is a list of variables, *a* is a form, and *al* is an association list, and each variable in *ll* is bound in *al*, we have

$$eval[a,al] = \langle \text{LAMBDA}, ll, a\rangle * evlis[ll,al], \tag{5a}$$

where the application indicated by * is to be done in *al*.

It follows that under the same assumptions

$$a \cong_{al} \langle \text{LAMBDA}, ll, a\rangle \circ ll, \tag{5b}$$

where the index on the equivalence sign indicates the environment in which the equivalence holds.

Finally we need the following:

LEMMA. *If all bindings in the association list al also occur in the association list e, and*

$$a \cong_e a',$$

then it follows that

$$redform[a,al] \cong_e redform[a',al].$$

The lemma follows immediately from the property of *redform* that was specified at the beginning of Section 3.

With this, we are ready for:

PROPOSITION

$$redfun[funct[\langle \text{LAMBDA}, ll, a\rangle,vl,e],\text{NIL}] \cong_e$$
$$\cong_e \langle \text{LAMBDA}, ll, eval[redcompform[a,vl],e]\rangle,$$

where $\cong_e$ *means that they are equivalent in the generation environment e, for all choices of a, vl, and ll.*

Proof. By (3), the left-hand side of the expression equals

$$redfun[\langle \text{FUNARG}, \langle \text{LAMBDA}, ll, a\rangle, pair[vl,vv]\rangle,\text{NIL}],$$

where $vv = evlis[vl,e]$, i.e., the list of values of the variable-list variables in the generation-time environment. Here *ll*, *a*, and *vl*, but not *vv* are known at compile time. By the FUNARG case of *redfun* (see Section 3), this equals

$$redfun[\langle \text{LAMBDA}, ll, a\rangle,pair[vl,vv]].$$

According to (4) this expression equals

$$\langle \text{LAMBDA}, ll, redform[a,pair[vl,vv]]\rangle.$$

By (5b) and the lemma, this is equivalent in *e* to

$$\langle \text{LAMBDA}, ll, redform[\langle \text{LAMBDA}, \text{NIL}, a\rangle \circ \text{NIL},pair[vl,vv]]\rangle$$

which by (1) is equal to

$$\langle \text{LAMBDA}, ll, redcompile[\langle \text{LAMBDA}, \text{NIL}, a\rangle,vl] *$$
$$prepev[\text{NIL},pair[vl,vv],mask[\text{NIL},vl]]\rangle.$$

By the definition of *prepev*, this equals

$$\langle \text{LAMBDA}, ll, redcompile[\langle \text{LAMBDA}, \text{NIL}, a\rangle,vl] * \text{NIL}\rangle$$

which is equal to

$$\langle \text{LAMBDA}, ll, \langle \text{LAMBDA}, \text{NIL}, redcompform[a,vl]\rangle * \text{NIL}\rangle$$

according to (2), this equals

$$\langle \text{LAMBDA}, ll, eval[redcompform[a,vl],e]\rangle$$

using (5a). In summary we have

$$redfun[funct[\langle \text{LAMBDA}, ll, a\rangle,vl,e],\text{NIL}] \cong_e$$
$$\cong_e \langle \text{LAMBDA}, ll, eval[redcompform[a,vl],e]\rangle.$$

This completes the proof.

Only one step in the proof is an equivalence, and the others are equalities. The equivalence really arises because of a technicality, and is not essential. If *redform* is appropriately defined, it goes away. We should therefore have equality rather than merely equivalence in this result.

It is straightforward to write a small generator-compiler which changes the old generator into the new one, and which uses *redcompile* as an auxiliary function. However, since this is an equivalence transformation, it is tempting to use *redfun* itself for speeding up the generator. This is done as follows: The function *redcomp* is defined through

$$redcomp[lx,vl] = list[\text{LAMBDA},cadr[lx],$$
$$eval[redcompform[caddr[lx],vl]]]$$

and is declared OPEN. Also, the function *redcompform* is declared PURE, which is appropriate since it is free of side-effects. Finally, the result above is implemented as a rule

$$\langle \text{REDFUN}, \langle \text{FUNCTION}, lx, vl\rangle, \text{NIL}\rangle \rightarrow$$
$$\rightarrow \langle \text{REDCOMP}, \langle \text{QUOTE}, lx\rangle, \langle \text{QUOTE}, vl\rangle\rangle$$

in the COLLAPSER property of the atom REDFUN. With this, *redform* will change the original generator, first to the form

$$\langle \text{LAMBDA}, \langle R\rangle,$$
$$\langle \text{LIST}, \text{'LAMBDA}, \text{'}ll, \langle \text{EVAL}, \langle \text{QUOTE}, ac\rangle\rangle\rangle\rangle,$$

where *ac* is the value from *redcompform*, and then (using another simplification rule) change (EVAL (QUOTE *ac*)) to *ac*. This is the desired result for improving the equivalence.

The second example is taken from the compilation and optimization of an axiom in PCDB. The original code for this was essentially

$$redform[\langle \text{PROG}, \ldots, makbind[\ldots],$$
$$makcont[\ldots],$$
$$makbind[\ldots],$$
$$makcont[\ldots],$$
$$\ldots\rangle,\text{NIL}]$$

with the addition of some labels etc., in the *prog*-expression. The function *makbind*[*l*,*n*] is defined as

$$\langle \text{BIND}, kwote[car[l]], kwote[cadr[l]] \ldots\rangle,$$

where *l* contains information of the conditions in the axiom. *Makcont* is analogous. The code generation will be speeded-up by redcompiling *bind* and *cont*, and their auxiliaries. In this case $ll = vl$, in the equation (2), i.e., all the known variables

will be arguments and there will be no known variables which are not arguments. This is just the opposite of the previous example. REDCOMPILE can be introduced by omitting the call to REDFUN and changing the definition of *makbind* into

$$redcompile[bind,ll] * \langle car[l], cadr[l], \ldots \rangle,$$

where the call to *redcompile* can be evaluated once and for all. *Makcont* is redefined similarly. The equivalence is proved as follows:

$$redform[\langle \text{PROG}, \ldots makbind[\ldots], \ldots \rangle, \text{NIL}] =$$
$$= \langle \text{PROG}, \ldots redform[makbind[\ldots], \text{NIL}], \ldots \rangle$$

where the essential element in the list equals

$$redform[\langle \text{BIND}, kwote[car[l]], kwote[cadr[l]], \ldots \rangle, \text{NIL}]$$

which, if the definition of *bind* is $lx = \langle \text{LAMBDA}, vl, a \rangle$, equals

$$redcompile[lx,vl] *$$
$$prepev[\langle kwote[car[l]], kwote[cadr[l]], \ldots \rangle,$$
$$\text{NIL}, mask[vl,vl]]$$

which, since $mask[vl,vl] = \langle T, T, \ldots \rangle$ equals

$$redcompile[lx,vl] * \langle car[l], cadr[l], \ldots \rangle.$$

Clearly equality in this case assumes that *redform* does not make any change to the top level of the *prog*-expression besides applying itself to each form in it.

In the completed application, the programs treated by REDCOMPILE were written in full LISP. It was easy to implement the new rules that were necessary to treat the special functions, and to expand the capability of simplification. Some restrictions had to be put on the programs, just as with REDFUN. The most serious of these are that assignment of new values to variables which are considered as known, creates problems. With REDCOMPILE it is easy to manage this for the majority of situations, but there exists cases that REDCOMPILE cannot manage.

8. The REDCOMPILE Program

REDCOMPILE is in many respects designed very similar to REDFUN, although it naturally uses somewhat different strategies.

REDCOMPILE expects every function to belong to the same function-classes as REDFUN does. If the same declarations are used, the finally resulting programs will be equivalent. The only exception is that the REDUCED class is not implemented as yet.

The functions are treated in a totally different way in REDCOMPILE, though, since at the time it works, compilation time, it does not have access to the values of variables considered as known. These values will not be known until generation time, i.e., when the resulting program is evaluated to give the optimized version of the original program.

The central function is *redcompform*, which was formally defined in Section 7, and is an analog of *redform*. It takes a form as argument, together with a list of known variables. As result, it gives a form which evaluates to a form that is equivalent (in the strong sense) to the input form.

If the input *form* given to *redcompform* is an atom, the result is almost trivial. If the atom is a variable that is expected to be known at generation time, the result is a form that shall evaluate (at generation time) to a form that evaluates (at execution time) to the value. The former is

$$\langle \text{LIST, QUOTE, } form \rangle$$

which evaluates to

$$\langle \text{QUOTE, } v \rangle$$

at generation time (if v is the value).

If the input form is non-atomic,

$$\langle fn, arg_1, arg_2, \ldots, arg_n \rangle$$

redcompform will proceed exactly as *redform*. If *fn* belongs to any class but SPECIAL, *redcompform* will first call itself recursively on arg_i until it encounters atomic forms. Depending upon the class of *fn*, it will then proceed in different ways.

If *fn* is PURE, *fn* is not immediately applied to the arguments even if they are considered as known, since we can not expect to have access to their values until generation time. We postpone the evaluation until then by returning

$$\langle \text{LIST, QUOTE, } \langle fn, \ldots \rangle \rangle,$$

where the dots stand for the result of *redcompform* on arg_i.

It is easier to find a strategy for *redcompform* when *fn* is OPEN than is the case with *redform* (defined in Section 3), since we do not know the values of the arguments. To do the proper compilation of *fn*, we need only to know which of its arguments will be known at generation time, i.e. consist only of constants and known variables and/or functions without side-effects of these. When *redcompform* is applied to the arguments it determines for which of these this is true, and therefore this is known when the beta-expansion of *fn* is done.

One way to proceed with this would be to substitute the lambda-variables for the arguments as we compile the function body of *fn*. This would not be so efficient, since the arguments have to be evaluated at generation time, and this would have to be done once every time they occur in the function body. An easier way to solve the problem, and prevent this, is the following:

(a) Make a call to *redcompile* with *fn*, where the list of known variables is generated by matching the list of lambda-variables with the list of arguments. This can be done without difficulties arising because the arguments that are to be known are easily distinguished from the others.

(b) Save the result of this call—which is a lambda expression—on the property list of *fn*.

(c) Insert a call to this lambda-expression in the place of the original call to *fn*.

(d) Repeat this recursively when a new OPEN function is encountered. Thus the insertion of the function body of *fn* is postponed until generation time.

One problem is not solved by the method described above, namely when the beta-expansion of a function creates a new sequence that can be simplified. This is not recognized at compilation time. The easiest method to solve this would be local calls to *redform* at generation time at the places where this might happen.

SPECIAL functions are treated in the same manner as *redform* does, i.e. *redcompform* is applied to the proper subforms by an analogue to the REDUCER property of the function name that *redform* uses.

The no-op case, i.e. the LAMBDA class, is not trivial for *redcompform*, since it can not return the form unchanged, with only the arguments properly treated. Instead it must return a form that evaluates to a form differing from the original only by having simplified arguments. This is done by returning

$$\langle \text{LIST}, \langle \text{QUOTE}, fn \rangle, \ldots \rangle.$$

The result of an application of *redcompile* to a program will be a program that is considerably more complex—but when evaluated gives the same nice optimized version of the original program as the one *redfun* makes.

Appendix

We will here give some figures, which we obtained from the various experiments. We will compare execution time for unspecialized and specialized code both when the generated code was interpreted and compiled (by the LISP compiler). Figures to compare the speed of REDFUN and REDCOMPILE with the LISP compiler are also given.

The first example is to generate code from parameters in the PCDB application. In a typical testrun we obtained the following results:

	time interpreted code	time compiled code
unspecialized code	3.62 sec.	2.29 sec.
specialized code	1.11 sec.	0.52 sec.

To get an idea of how slow REDFUN and REDCOMPILE are compared with the LISP compiler we have made the following testruns:

We use a generator as described in Section 4, to generate a storage function for a binary relation (in PCDB a STOREDEFINER for the relationtype CLEAN-LINK). In the first test we measure the time to generate the funarg-expression and to optimize it by REDFUN. Next is to measure the time it takes to run the generator through REDCOMPILE which also implies redcompilation of the generators auxiliary functions, and then measure how fast the retrieval function is generated by this redcompiled generator. At last we compile the code in the original generator and its auxiliary functions with the normal LISP compiler.

	time
(a) generation of a specialized storage function by optimizing with REDFUN (interpreted generation).	2.80 sec.
(b) redcompilation of the generator and its auxiliary functions (compilation).	7.00 sec.
(c) generation of a specialized storage function by the generator produced from step (b) (compiled generation).	0.68 sec.
(d) compilation of the original generator and its auxiliary functions by the LISP compiler.	8.34 sec.

In the next example we used the following forward axiom

$$S(x,y,z) \wedge S(x,y,T) \wedge S(x,v,z) \supset S(x,z,v).$$

We obtained the following figures when this axiom was used in an application:

	interpreted code
unspecialized code	12.19 sec.
specialized code	3.18 sec.

In one typical experiment, the following figures were obtained for the original GUP program and the specialized version generate REDFUN:

	original program	specialized program
size (approx. nr of cons-cells)	2150	1599
number of functions	29	22
time, uncompiled code	68.82 sec.	7.67 sec.
time, compiled code	5.16 sec.	2.22 sec.

ACKNOWLEDGMENT

The design and use of the REDFUN program has very much been a group effort in our lab. Responsible for PCDB and the application of REDFUN there was Anders Haraldson; for GIP/GUP, Östen Oskarsson and for REDCOMPILE, Lennart Beckman.

The first two sections of the report were written by Erik Sandewall. Each of the remaining sections was written by the author responsible for the program described there, in collaboration with Erik Sandewall.

We particularly want to thank Rene Reboh, Tore Risch, Arne Tengvald and Jaak Urmi in our lab, who have helped in various ways with this work, and Dave McDonald at the MIT A.I. lab who read parts of the manuscript and suggested numerous improvements.

REFERENCES

1. Boyer, R. and Strother Moore J. Proving theorems about LISP functions. Advance papers in Third International Joint Conference on Artificial Intelligence, Stanford Research Institute, Stanford, CA (1974).
2. Chang, C.-L. and Lee, R. *Symbolic Logic and Mechanical Theorem-Proving*. Academic Press, New York (1973).
3. Darlington, J. and Burstall, R. A system which automatically improves programs. *Proc. Third International Joint Conference on Artificial Intelligence*, Stanford (1975).
4. Deutch, P. An interactive program verifier. Ph.D. Thesis, Xerox Research Center, Palo Alto, CA (1973).
5. Dixon, J. The SPECIALIZER, a method of automatically writing computer programs. Div. of Computer Research and Technology, NIH, Bethesda, MD, unpublished report.

6. Futamura, Y. Partial evaluation of computer programs: An approach to a compiler–compiler. *J. Inst. Electronics and Communication Engineers* (1971).
7. Haraldson, A. PCDB—A procedure generator for a predicate calculus data base. *Information Processing* 74, J. L. Rosenfeld, ed., North-Holland, Amsterdam (1974), 575–579.
8. Hardy, S. Automatic induction of LISP functions. Essex University (December 1973).
9. Lombardi, L. A. and Raphael, B. LISP as the language for an incremental computer. E. Berkley and D. Bobrow, eds. *The Programming Language* LISP: *Its Operation and Application*, MIT Press, Cambridge, MA (1964).
10. Sandewall, E. A. Programming tool for management of predicate-calculus-oriented data bases. *Proc. Second International Joint Conference on Artificial Intelligence*, British Computer Society (1971).
11. Sandewall, E. A. A proposed solution to the FUNARG problem. *ACM SIGSAM Bull.* **17** (1971).
12. Sandewall, E. A. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. Advance papers of Third International Conference of Artificial Intelligence, Stanford Research Institute, Stanford, CA (1973).
13. Teitelman, W. INTERLISP *Reference Manual.* Xerox Research Center, Palo Alto, CA (1974).
14. Wagbreit, B. The treatment of data types in ELL. *Comm. ACM* **5** (1974), 251–264.
15. Winograd, T. Procedures as a representation for data in a computer program for understanding natural language. Artificial Intelligence Laboratory, MIT, Cambridge, MA (February 1971).

Received March 1975