Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 305 May 1988

Program Improvement by Automatic Redistribution of Intermediate Results

A Thesis Proposal

Robert J. Hall

Abstract

The problem of automatically improving the performance of computer programs has many facets. A common source of program inefficiency is the use of abstraction techniques in program design: general tools used in a specific context often do unnecessary or redundant work. Examples include needless copy operations, redundant subexpressions, multiple traversals of the same datastructure and maintenance of overly complex data invariants. I propose to focus on one broadly applicable way of improving a program's performance: redistributing intermediate results so that computation can be avoided. I hope to demonstrate that this is a basic principle of optimization from which many of the current approaches to optimization may be derived. I propose to implement a system that automatically finds and exploits opportunities for redistribution in a given program. In addition to the program source, the system will accept an explanation of correctness and purpose of the code.

Beyond the specific task of program improvement, I anticipate that the research will contribute to our understanding of the design and explanatory structure of programs. Major results will include (1) definition and manipulation of a representation of correctness and purpose of a program's implementation, and (2) definition, construction, and use of a representation of a program's dynamic behavior.

1 Introduction

Computer programs generally call many subroutines and perform many state changing operations in the course of a computation. On completion, a subroutine returns one or more values and possibly produces some changes in the program's state (via side effects). I will term any such value or change an *intermediate result* (or simply a *result*). These are used as input to and to satisfy preconditions for other program modules; however, the computation of intermediate results generally requires the use of resources (time and space). One cause of inefficiency in a program is when it wastes resources computing some intermediate result even though it has previously computed another result which could be used instead. A program that does this can be improved by changing it so that it arranges to use the other result instead of computing the extraneous result. I term this type of improvement a *redistribution of intermediate results.* In this research, I propose to focus on just this type of improvement.

Redistribution can be implemented in many ways. Examples include storing for later retrieval (using anything from a local variable up to a global, special-purpose datastructure), passing extra parameters and return values among subroutines, and moving client code into the local context of the result's initial computation. Redistribution has the effect of eliminating useless and redundant computation; computation is useless if its input could be redistributed as its output[1]; computation is redundant if its output has been computed previously and could have been stored and retrieved.

Opportunities for this type of improvement arise frequently because it is much easier to write clear, correct programs using general, well-understood software components than it is to generate specialized code for each task. However, a general tool used in a specific context is likely to do more work than strictly necessary. Another source of redundant computation is that designers find it easier to think about operations one at a time, rather than interleaved for maximum efficiency. This can result in multiple datastructure traversals and multiple copies of a given subexpression, for example.

The task of my system will be to automatically identify opportunities for redistribution in the given program and to automatically install appropriate code to do so. A key idea behind my approach to solving this problem is the hypothesis that, while a given redistribution may require major changes in the program's structure, the program's explanation (proof) of correctness will not change much. Thus, given an initial explanation of correctness, it should be relatively easy to produce not only the optimized program but also a modified correctness explanation that justifies it. This is one way in which I anticipate this work differing markedly from traditional approaches to code improvement: the system will be

[1] ...or if its output is never used at all, of course.

given an explicit representation of the program's explanatory structure in addition to its module structure (source code). Determining the information required in such a representation, and guidelines for its construction, will be a major research contribution.

I distinguish redistribution from optimizations which change the basic algorithmic strategy of the program. For example, changing the implementation of a sorting module from bubblesort to heapsort involves more than simply eliminating computations; it involves significantly changing the temporal order of comparisons, with a concomitant change in the explanation (proof) of correctness. That is, the program comes to be correct for fundamentally different reasons. This distinction is not completely precise as yet; however, I anticipate that the research will clarify it.

1.1 Why Redistribution Is Easier Than Optimization in General

Even though redistribution is a remarkably powerful optimization principle (see Section 2 for examples), I believe the problem to be significantly easier than the entire problem of program optimization primarily because far less *domain-specific* creativity and synthesis knowledge is required. As an example, suppose we wish to calculate, for any nonnegative integer n, the nth Fibonacci number, defined by

$$F(n) = \begin{cases} 1, & n = 0, 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

Starting with the program obtained from straight-forward transcription of the defining equation (which calls F recursively an exponential number of times) it is relatively easy to derive a linear-time version that simply redistributes intermediate results instead of recomputing them. The only domain knowledge required is that needed to show that $(n-1)-1 = n-2$. (See Section 2.)

On the other hand, improving the program so that it only requires $O(\lg(n))$ recursive calls is much more difficult. One must derive the mathematical identities

$$\begin{aligned} F(2n+1) &= (F(n))^2 + 2F(n)F(n-1) \\ F(2n) &= (F(n))^2 + (F(n-1))^2 \end{aligned}$$

from the definition of F. In addition to programming skill, this requires mathematical creativity. I distinguish the domain-specific creativity required to synthesize these identities from the ability to prove domain-specific identities correct[2] (a skill more akin to analysis than synthesis). Clearly, to design a particular correct

[2] I believe that MACSYMA could prove the above identities correct; it could not, however, think of them in the first place!

implementation (and know that it is correct) is strictly harder than just knowing that it is correct. I hypothesize that solving the redistribution problem requires some ability to prove identities, but no ability to design them.

Another way in which redistribution is potentially easier than general optimization lies in determining whether a given optimization is worth carrying out. In general, knowing whether a program's efficiency will improve significantly after an optimization is very difficult, involving a deep understanding of the performance characteristics of the code. On the other hand, it is almost always beneficial to redistribute an intermediate result instead of recomputing it. There are exceptions to this rule: the optimization can be ill-advised if maintenance of an auxiliary datastructure required to store the values for later use dominates the cost of the re-computation[3]. My initial approach will be to ignore these cases, tacitly assuming that redistribution is always a win[4].

1.2 Why Redistribution is Difficult

I believe detecting and exploiting opportunities for redistribution is difficult for three reasons. First, it requires an ability to visualize and characterize abstractly the dynamic behavior of the program. This is because the source code for the program explicitly contains only the information necessary to execute it. With a few minor exceptions like variables' types, all other abstract properties are implicit, for example, "the first visit by this sort routine to any given sequence element happens in sequence order (i.e. before those of succeeding elements and after those of preceding elements)." It is not easy to deduce a fact like this from the source code alone. One goal of this research is to explore ways of visualizing and characterizing the dynamic behavior of a program. (See Section 4.)

Second, redistribution does require some ability to reason with the axioms governing the behavior of the abstract data types. Specifically, the system must be able to show that a subgoal follows from previously achieved subgoals. (This is the "analytic" ability mentioned earlier.) This occurs when the optimizer must infer that a previously computed value can be used to satisfy the purpose of a later computation. In the Fibonacci example, the optimizer must be able to deduce that $(n-1)-1 = n-2$ so that it may conclude that the nested recursive call to F computes a value which can be used to satisfy the purpose of the top-level call to $F(n-2)$.

[3]For example, if a Lisp function uses `(CAR L)` twice for each of a sequence of lists `L`, it will be faster to leave it as is than to store the values in a hash table (with keys `L`). Building and accessing a hash table is costly compared to the `CAR` operation.

[4]While this research is not intended to mimic human cognitive abilities, I have observed human hackers overdo the redistribution optimization, creating large datastructures to hold intermediate results rather than leaving in an elegant and cheap recomputation.

Accordingly, I will assume that in addition to the source code, the program is given axioms governing the behavior of the abstractions used in building the program. Of course, it is clear that the problem of proving theorems from these axioms rapidly becomes intractable. I take the view that the program will do as well as it can in trying to demonstrate needed equivalences, but may quit in failure after a certain amount of effort is expended. This can only result in the optimizer missing some opportunity; it will not result in an incorrect program. This approach is in accord with my intuition about how humans handle analogous difficulties in reasoning about programs. I expect that this reasoning will tend to be on the easy side for most of the naturally occurring problems associated with the use of libraries of general programming abstractions.

The third major difficulty is that the problem seems to require knowing the specifications of the program elements and an explanation of how the implementation satisfies the specifications. I intuit that I use this information when I optimize my own code, for example. To know that some code is useless, one must know the overall purpose of the program and how the structure of the program satisfies this specification. Traditional compilers and code improvers have no access to this information, even though the person who writes the program must know it to some degree. Most source languages are unequipped to capture anything beyond an operational description of the program code.

As this research is targeted at understanding a portion of the whole process of design, I will assume that the program is given a representation of this extra explanatory structure along with the source description. My underlying assumption about the design process is that the designer develops an (at least informal) explanation of correctness and purpose of the program's structure in the process of designing the program. Once the basic design is intact, the designer then uses this explanation to help in optimizing the code. My program, therefore, will require that this explanatory structure be submitted along with the program's structural information. The definition of this extra structure will be an important research result in its own right.

An added difficulty is that *complete* explanations may be difficult to produce and maintain. Therefore, a related research question will be how the program improver can deal with partial explanations. I expect to explore the complete-information case before the partial.

1.3 Significance of the Problem

The problem of program improvement is of interest to both the artificial intelligence and software engineering communities. From an artificial intelligence perspective, optimization is a key step in the design process; design, in turn, is an important kind of human problem solving. If a significant part of the process

could be understood as an application of one basic principle to a well-defined input representation, it would be an important contribution. Such a result would lay groundwork for studies of more sophisticated optimization and automated design in general. I believe that a connection can be drawn between the redistribution principle and the domain-independent notion of function sharing in design. This is the general notion that a single component of a design can serve several (possibly unrelated) purposes; it can be seen in domains as diverse as electrical engineering, mechanical engineering, and architecture.

From a software engineering perspective, the work promises to contribute to our understanding of the representation and manipulation of the purpose, correctness, and behavior of programs and other complex devices. This research can be viewed as an attempt at representing and exploiting knowledge of (part of) the design history of a program, particularly the explanatory structure pertaining to correctness and purpose. It could also help to unify the existing theory of program optimization, in that I believe that many existing forms of program improvement can be understood as limited applications of the redistribution principle.

Though my primary goal is not the production of a practical tool for software engineering, this work could form the basis for one. The desirable goal of widespread use of data and procedural abstractions[5] is inhibited (in part) by the inefficiency of the resulting implementations. It appears that a large portion of the inefficiency introduced by abstraction can be understood as useless and redundant code, precisely the type of inefficiency at which the redistribution principle is directed. Thus, if this research is successful, it could make the use of abstraction much more practical.

The requirement of providing an explanation of purpose with the program source would seem to complicate the use of such a tool; however, recent programming methodologies (for example, see Gries (1981)) advocate the concurrent development of code and correctness proofs, roughly the kind of extra information needed. Current software development methodologies rally about the slogan, "First make it right, then make it fast." This research could contribute significantly to automating the second half of that phrase.

In summary, I expect that this research will make the following specific contributions.

- A better and more formal understanding of the teleology of program designs (and the designs of complex information processing devices in general)
- A computer representation of teleological explanations of program designs.
- Fully automatic techniques for detecting and exploiting opportunities for redistribution of intermediate results in a program.

[5]Also: the widespread reuse of libraries of such abstractions

- Basic theory which could lead to a tool making more practical the widespread use and reuse of general data and procedural abstractions.

2 Examples

In this section, I will try to sketch the behavior of the proposed system on a set of examples. I will not attempt to hint at possible solution methods for the problems; that will be done in Section 4. My purpose is to make a clean separation between the task and its potential solutions. The problems considered are of varying difficulty.

Some readers may take issue with the examples on the grounds that "no one would write code THAT inefficient!" While it is possible to carry anything to extremes, I believe this objection misses the basic point that the objectives of efficiency and clarity/correctness conflict. Most programmers today are forced to think as much in terms of efficiency as in terms of correctness; in fact, it is difficult for us to think in terms of a programming style whose primary emphasis is not efficiency. This is not a new idea; it is a fundamental tenet of the program transformation school, as well as other methodologies.

I believe that there are objective reasons for desiring some (possibly less efficient) formalizations over others. These reasons can be stated as desiderata for the structure of a program.

- *Modularity:* Good abstractions should have a high degree of modularity, partitioning the implementation task along abstraction boundaries. This allows dividing up large programs among many programmers and localizes the effects of changes to the implementation.

- *Elegance:* Programs should be expressed simply and elegantly. This greatly increases their comprehensibility and makes automatic manipulation of the source much more likely.

- *Reuse:* General, powerful abtractions are easier to reuse.

- *Intellectual Economy:* To be usable over a long period of time by many programmers, a library of software abstractions must not have too many highly-specific tools. In short, the user's manual should not be two feet thick; otherwise, no one will ever bother to master it.

A goal of this research is to make it possible to divorce, to a greater degree, the task of programming from issues of efficiency. Thus, the examples should be judged against the above desiderata and not primarily for efficiency.

2.1 Fibonacci Numbers

As a more difficult example, consider the well-known Fibonacci numbers, defined in Section 1. The most natural program for finding the nth one is taken directly from the definition:

```
(defun F (n)
   (cond ((= 0 n) 1)
         ((= 1 n) 1)
         (:ELSE (+ (F (- n 1)) (F (- n 2))))))
```

By expanding the recursion one level, one finds that the first call, `(F (- n 1))`, calculates `(F (- n 2))`, among other things. Thus, if only the code could export this intermediate value, it would not need to call F again. We can define a new subroutine, FF which is like F except that it returns the pair $[F(n), F(n-1)]$ on input n. But now the same critique applies to the subroutine FF since it contains the same recursive structure as the original F; however, the same fix applies and uses FF itself:

```
(defun F (n) (values FF(n)))

(defun FF (n)
   (declare (values Fn Fn-1))
   (cond ((= 0 n) 1)
         ((= 1 n) 1)
         (:ELSE
            (multiple-value-bind (Fn-1 Fn-2)
                (FF (- n 1))
              (values (+ Fn-1 Fn-2) Fn-1)))))
```

This restructuring changes the original, exponential program into a linear one. A related method for redistribution of intermediate results, called memoizing, would cache the F values in a table as they were computed the first time and then look up the values on subsequent calls.

```
(defvar *BIGGEST-N* 20000)
(defvar *MEMO* (make-array *BIGGEST-N* :initial-element NIL))

(defun F-m (n)
   (cond ((= 0 n) 1)
         ((= 1 n) 1)
         ((aref *MEMO* n))
         (:ELSE
            (setf (aref *MEMO* n)
                  (+ (F-m (- n 1)) (F-m (- n 2)))))))
```

There are subtle space/time tradeoffs[6] dictating which method would be better in a given situation which I do not wish to explore here. Also, one can do much better than the worst-case linear number of recursive calls exhibited by these two approaches. However, that is not achievable solely through redistribution of intermediate results.

2.2 A Mergesort Optimization

The mergesort algorithm operates by splitting the input sequence in half, recursively sorting the halves, and then merging the sorted subsequences to form the result. During the recursive splitting phase, we can maintain indices into the original sequence rather than actually consing up the intermediate sequences, waiting until the base case (length 1) to create new lists. This gives us the following program.

```
;; assumes seq is a list of numbers
(defun mergesort (seq)
  (msort seq 0 (length seq)))

(defun msort (seq base lngth)
  (if (= 1 lngth)
      (list (nth base seq))
      (let ((lngth-by-2 (div lngth 2)))
        (merge (msort seq base lngth-by-2)
               (msort seq
                      (+ base lngth-by-2)
                      (- lngth lngth-by-2))))))

(defun nth (n list)
  (cond ((null list) ERROR) ;;non-local exit
        ((= 0 n) (car list))
        (:ELSE
         (nth (- n 1) (cdr list)))))

;;; details unimportant: assume it's linear time
(defun merge (seq1 seq2) ...)
```

Normally, mergesort requires only $O(n \lg n)$ time, because `less-than-predicate` is only called that many times. However, the above implementation takes $O(n^2)$

[6] *e.g.*, "The first method is faster for a single call to `F(n)` for any `n`, but the second method is faster for multiple calls to the same `n`. The second method requires $O(n)$ space, however."

time, even though it only does $O(n \lg n)$ comparisons. This is because **nth** requires linear time to access the *nth* sequence element, and each of the n sequence elements is accessed in this way.

The key insight for optimizing this program is to note that the sequence elements are accessed (using **nth**) in sequence order; that is, the zeroth element is accessed first, the oneth next, etc. Realizing this, we can see that there is an opportunity for redistribution: a call to **nth** with index k computes a pointer to the kth cons cell in the list. But to compute the $(k+1)$th cons cell, we need also to compute the kth. Thus, we should be able to reuse this pointer between succeeding calls to **nth**.

```
(defvar *list-pointer*) ;implementing redistribution; <---

(defun mergesort-1 (seq)
  (setq *list-pointer* seq) ;initialize            ; <---
  (msort-1 0 (length seq)))

;; note: no longer need seq as parameter
(defun msort-1 (base lngth)
  (if (= 1 lngth)
      (let ((result
              (list (car *list-pointer*))))        ; <---
        (setq *list-pointer* (cdr *list-pointer*)) ; <---
        result)
      (let ((lngth-by-2 (div lngth 2)))
        (merge (msort-1 base lngth-by-2)
               (msort-1 (+ base lngth-by-2)
                        (- lngth lngth-by-2))))))
```

This change removes the quadratic term in the program's running time, rendering an $O(n \lg n)$ program. Further redistribution optimizations are now possible; for example, the program no longer needs to maintain a "base" offset, because the base is only used to calculate other bases and has no effect on the result. This will have a smaller effect on the run-time of the program, only reducing the constant factor somewhat.

```
(defun mergesort-2 (seq)
  (setq *list-pointer* seq)
  (msort-2 (length seq)))

;; note: no longer need 'base'     ; <---
(defun msort-2 (lngth)
```

```
  (if (= 1 lngth)
      (let ((result
              (list (car *list-pointer*))))
        (setq *list-pointer* (cdr *list-pointer*))
        result)
      (let ((lngth-by-2 (div lngth 2)))
        (merge (msort-2 lngth-by-2)
               (msort-2 (- lngth lngth-by-2))))))
```

2.3 A Binary Tree Program

Consider a binary tree abstraction, where the leaf nodes contain integers. We define an operation, `inc-leaves`, which returns a new tree isomorphic to the input such that corresponding leaf values are incremented by one. Now suppose that we also wish to define a program that adds two to each leaf. In keeping with the desiderata given earlier, we would like to do this in terms of inc-leaves.

```
(defun inc-leaves-twice (tree)
   (inc-leaves (inc-leaves tree)))

(defun inc-leaves (tree)
  (cond ((is-leaf? tree)
         (make-leaf (+ 1 (data-field tree))))
        (:ELSE
          (make-internal-node
            (inc-leaves (left-child tree))
            (inc-leaves (right-child tree))))))
```

Basically, `inc-leaves` operates by copying the structure of the input tree, and creating leaves with incremented data fields. It is easy to see that, in the context of `inc-leaves-twice`, the second call to `inc-leaves` need not copy the intermediate tree; it only needs to alter its leaves' data fields.

```
(defun inc-leaves-twice-1 (tree)
  (let ((result (inc-leaves tree)))
    (inc-leaves-1 tree)
    result))

(defun inc-leaves-1 (tree)
   (cond ((is-leaf? tree) (increment-data-field tree) tree)
         (:ELSE
            (inc-leaves-1 (left-child tree))
```

```
            (inc-leaves-1 (right-child tree))
            tree)))
```

Note that in this case, the redistribution is between the input and output of an operation (the **make-leaf** and **make-internal-node** operations); this redistribution eliminates the "useless" copy operations.

Redistribution can be implemented in many different ways. Standard ways are storage in a global variable, extra parameter passing, and storage in a hash table. These methods were illustrated above. A less obvious way is to actually move the second use backward in time to the local context of the first use. (Obviously, this is only feasible in certain cases.)

```
;;; This is not a legal Lisp program.
;;; It is intended as a schematic of
;;; the dynamic behavior of the program.

...

*ENTRY-POINT: inc-leaves*
    (cond ((is-leaf? tree)
           (make-leaf (+ 1 (data-field tree))))
          (:ELSE
            (make-internal-node
              (inc-leaves (left-child tree))
              (inc-leaves (right-child tree)))))
                    |
                    | (produces 'new-tree')
                    |
                    V
*ENTRY-POINT: inc-leaves-1*
    (cond ((is-leaf? new-tree)
           (increment-data-field new-tree)
           new-tree)
          (:ELSE
            (inc-leaves-1 (left-child new-tree))
            (inc-leaves-1 (right-child new-tree))
            new-tree))
...
```

Here the redistribution opportunity consists of the fact that each tree node is created in the first recursion (*ENTRY-POINT: inc-leaves*) and then used in the second recursion (*ENTRY-POINT: inc-leaves-1*). That is, the left-right

recursion below the `inc-leaves-1` entry-point is there for the purpose of finding the leaves of `result`. But the leaves are created in the code above, so the optimizer could move the "actions" of the second tree-walk up to where the tree nodes are created.

```
(defun inc-leaves-twice-2 (tree)
  (cond ((is-leaf? tree)
         (let ((newleaf (make-leaf (+ 1 (data-field tree)))))
           (increment-data-field newleaf)
           newleaf))
        (:ELSE
          (make-internal-node
            (inc-leaves-twice-2 (left-child tree))
            (inc-leaves-twice-2 (right-child tree))))))
```

This can be viewed as a sophisticated instance of "fusion." Traditional compilers fuse loops; this effectively fuses recursive control structures as well. The recursion above was not a simple linear loop. Fusion required showing that the leaf nodes reached by the second tree walk were precisely all and only the leaf nodes created in the first one, and that none was visited more than once. This would have been easier had the two tree walks been over the same tree; this case was tricky because the two were over different, but isomorphic, copies.

Some further optimization is possible, like adding 2 instead of adding 1 twice, but that requires thinking of and exploiting an algebraic identity (namely $(x + 1) + 1 = x + 2$), which is a different type of optimization than redistribution of intermediate results.

2.4 A Set Example

Suppose we have the following implementations of the set abstraction. (Many operations are omitted as unnecessary to this example.)

```
;; we choose to represent a set as a list of its elements
;; (set-to-list and list-to-set change views of an object)
;; subject to the invariant that the list has no duplicates

(defun set-add (element set)
  (if (member element (set-to-list set))
      set
      (list-to-set (cons element (set-to-list set)))))

(defun set-cardinality (set)
```

```
  (length (set-to-list set)))

...
```

Consider the following user-supplied program (detail omitted).

```
;; assume that no set-cardinality operations appear
(defun my-program (...)
  (declare (values integer))
  ...(set-add ...) ...
  ...(set-member? ...) ...)
```

The redistribution opportunity here is a bit subtle. The optimizer should realize that the only reason for the representation invariant "no duplicates in list" of the set representation is so that **set-cardinality** will be correct[7]. (The invariant is required so that the number of distinct elements in the list will match the list's length.)

Inasmuch as **my-program** contains no calls to **set-cardinality**, and its return value (of type **integer**) contains no references to sets, the maintenance of the invariant is not needed. Furthermore, the test in the implementation of **set-add** has the sole purpose of enforcing the invariant. Therefore, its input may be redistributed directly to its output, avoiding the membership test. (This has the effect of converting **set-add** from linear time to constant time.)

```
(defun set-add-1 (element set)
  (list-to-set (cons element (set-to-list set))))

;; assume that no set-cardinality operations appear
(defun my-program (...)
  (declare (values integer))
  ...(set-add-1 ...) ...
  ...(set-member? ...) ...)
```

As a final twist on this example, note that this optimization can actually have the effect of slowing down **my-program**; **set-member?** may now have to search a very long list (compared to what it searched before) in order to determine that an element is not in the set. Whether this happens is dependent on the actual frequencies of additions, removals, and membership tests. Discovering this reasoning is beyond the scope of the present proposal.

In this case, it is clear that the "no duplicates" invariant of the set representation has an additional purpose, one outside of correctness issues: it also has

[7]Of course, if there were other operations defined, their correctness might depend on the no-duplicates invariant also. I assume not for this example.

the purpose of keeping **set-member?** (and presumably other, unmentioned operations like **set-intersection** and **set-union**) *efficient*. This is but one reason why I believe it is crucial that an optimizer understand the purpose structure in an implementation in order to be able to optimize effectively. I intend that my system be able to understand this type of purpose, as well as correctness, and use it to guide optimization. That is, explanatory structure may contain reasons (purposes) based on efficiency considerations, and my system should handle these. On the other hand, my system is not expected to reason itself about the efficiency of code.

3 Literature Review

This section places this proposal in relation to previous approaches to the problem of improving the performance of programs.

Traditional Compiler Techniques. I view this proposal, in part, as an attempt at generalizing many traditional compiler techniques to apply to arbitrarily high-level abstractions. Techniques like common subexpression elimination, dead code removal, loop fusion, code motion, and data and control flow analysis (Aho, Sethi, & Ullman, 1986; Hecht, 1977) are limited by the level of their source languages. That is, they can only exploit the semantics of the data types that are primitive to the language, because they cannot capture and fully exploit the semantics of user-defined types. For example, a FORTRAN optimizer can only exploit the algebraic laws of integers, arrays, and other low-level types.

For this reason, I believe traditional approaches to optimization are fundamentally limited. To be significantly more effective they must solve two problems: first recover the explanatory structure of the high-level abstractions used by the programmer (or the equivalent); then exploit that knowledge to optimize the program. Consider the technique of common subexpression elimination. Optimizers can only perform this operation when they know (1) that the two expressions actually denote the same value, and (2) that there are no side-effects in the calculation of the expression that could cause a difference in meaning of the two expressions. Typically both of these can only be known about expressions in terms of language primitives (like arithmetic). A common subexpression involving a user-defined subroutine will in general not be shared by the optimizer because it can be arbitrarily difficult to determine whether (1) and (2) are satisfied.

I include in the category of traditional compiler techniques the operations of type inference and automatic datastructure choice and aggregation performed by the SETL compiler (Freudenberger, Schwarz, & Sharir, 1983). SETL is a much higher-level language than most, hence the optimizations its compiler performs have greater impact on the performance of programs. Nevertheless, these optimizations are still limited to the semantics of language primitives. The optimizer demonstrates a great deal of ingenuity in determining when certain optimizations regarding sets and mappings may be performed, but the language cannot capture any extra semantic information about higher-level, user-defined abstractions. While it might infer that a particular copy operation on a set is unnecessary, it will not be able to infer the analogous fact about a copy operation on a user-defined type. Since no language will ever predefine anywhere near all of the useful programming abstractions, it is doubtful whether the traditional approach to optimization will ever achieve the flexibility and power of a human expert.

Program Transformation School. The program transformation school (Partsch

& Steinbruggen, 1983; Cheatham, 1984; Darlington, 1981) takes the view that optimization should take place as a process of *program transformations*, usually at the source code level. Each transformation must provably preserve program correctness. Consequently, each has a set of applicability conditions which must be verified. An as yet unattained goal of the research is that these conditions be checked automatically so that program correctness is guaranteed no matter how the human influences the process.

Fully automatic approaches to choosing the sequence of transformations are not generally capable of producing efficient code, both because the search space is too large and because it is too difficult to determine the relative efficiency of the results. Consequently, most transformational approaches are semi-automatic in that a human must guide the selection process (also, the human must sometimes assist in verifying applicability conditions). This line of research cannot be termed a success as yet because the process of (a human) guiding the transformations is difficult and tedious; each transformation is relatively low-level, so many are required to carry out any particular optimization. Optimizing a large program from its clear, but inefficient specification requires too many small-grain steps to be feasible. (There is, however, ongoing research into structuring the transformation process; see Fickas (1985) for one approach and Meertens (1986) for many papers on this subject.)

Inasmuch as the program transformation research does not address the issue of automating the search, it cannot shed much light on that aspect of the current proposal, Of course, some of the transformation classes defined there are useful here, such as folding, unfolding, and specialization.

A particular branch of this field (Wile, 1981; Scherlis, 1981) investigates a basic issue of interest to my proposal: specializing data type implementations to their contexts. While these approaches discuss some of the transformations possible, they again do not discuss the issue of automating the search. In particular, there is no discussion of explanatory structure or the use of correctness information in guiding the search.

Finite Differencing. Finite differencing (Paige & Koenig, 1982) is a method for improving programs by replacing repeated all-at-once computations with more efficient incremental versions. The implementation in RAPTS (Paige, 1983) is semi-automatic in that a user must decide which instances of differencing to apply and whether an instance is desirable. The system is given a sizable base of specific "differentiation" rules, each of which applies to some pattern of operations expressed in SETL. For example, one such rule says that the expression, $\#S$, (size of the set S) can be maintained incrementally by (1) initially calculating the size of S, (2) for every addition to S adding 1 to it, and (3) for every deletion subtracting 1.

Finite differencing is an elegant idea, originating in the work of the 16th century mathematician H. Briggs on evaluating polynomials. Expressing this idea in terms of a large rule base of highly specific instances is rather *in*elegant, however. The main problem with this "expert system" style of approach is that to exploit the idea of differencing to its fullest, the system would require new rules for any new abstraction (language contruct or user-defined)[8]. This is not simply the "standard expert system complaint," however. Typical expert systems solve relatively "fixed" problems, where the expertise can remain fixed over time. By contrast, the designer creates new abstractions to meet specific problem needs. Therefore, in order for the rule-based implementation of finite differencing to be considered a complete theory of optimization, it must also account for how the rules are (automatically) derived from the definitions of the abstractions[9].

With regard to redistribution, I believe that much of the finite differencing *behavior* can be seen as an application of the redistribution principle. A principled approach to redistribution would, therefore, supply a partial answer to the problem mentioned above. Finite differencing could then be seen as an emergent behavior rooted in deeper principles. Of course, I am not opposed to rules, *per se*. Transformation rules compiled automatically from experience (those which could otherwise be derived automatically in a principled but slow way) are very useful for speeding up the system.

Memoizing. Mostow & Cohen (1985) have investigated automating the well-known technique of *memoizing.* In its most general form, this is the idea of a subroutine maintaining a cache of its output values for those inputs for which it has already computed an answer. If the subroutine is ever called more than once on the same input, the answer is looked up in the cache the second and succeeding times, rather than being recomputed. Mostow and Cohen explored the addition of caches to Interlisp functions, with an eye to building a fully automatic tool. Unfortunately, it appears that this problem is too difficult, because side effects and large datastructures make the technique difficult to justify in many cases. I believe the chief problem in this approach is that the memoizer has neither knowledge of nor control over the design process, because, like a compiler, it takes in only the Interlisp source code. It must always assume the worst possible cases of usage for any given subroutine, cases which could quite possibly be ruled out if extra information relating to purpose and correctness were known. I expect that memoizing will be a valuable technique for implementing redistribution of intermediate results.

[8]Other problems include the difficulties in deciding which rules to apply and whether a given rule will improve the efficiency of the program.

[9]This, of course, sounds like a perfectly reasonable subject for future research on finite differencing!

Tupling. Pettorossi (1984) has defined *tupling* to be the combination of two initially separate functions into a single, vector-valued function in order that their implementations may share partial results. Though he proposes it as a (manual) program transformation, it is clearly a useful technique for our purposes. The Fibonacci example in Section 2 is a special case of a tupling transformation, where the two functions to be tupled are the same function applied to different arguments.

Automatic Programming Approaches to Efficiency. Even though automatic programming systems really address a different problem from the redistribution task, they are nevertheless worth reviewing. They are in some sense complementary to the redistribution of intermediate results in that they operate by choosing implementations in a (more or less) top-down fashion, always staying within abstraction boundaries. Redistribution, on the other hand, is primarily concerned with breaking these boundaries.

Kant's LIBRA system (Kant, 1983) is designed to be a search control expert which guides the refinement process of the PSI (Green, 1976) synthesis system. PSI is based on stepwise refinement; that is, starting from a top level description, the implementer successively refines the functional blocks of the current design stage using implementation rules from a knowledge base. This grammatical rule knowledge defines a search space, with possible designs as the leaves of the search tree. Kant's work is a direct attempt at solving the search control problem in that space.

The system takes as input a specification in a very high level language which can be directly implemented, albeit inefficiently, by choosing default implementations. LIBRA's task is to override the defaults with appropriate choices to make the result more efficient. The system does not perform the problem solving necessary to come up with an algorithm in the first place; it operates after an operational scheme for the problem has been found.

LIBRA (1) picks a search tree node to work on, then (2) picks a refinement task within that node, then (3) picks a coding rule (implementation) which achieves the task. These choices are guided by three bodies of rules: resource management rules (1, 2), plausible implementation rules (3), and cost analysis rules (3). The cost analysis allows the system to do something akin to branch and bound search, eliminating partial implementations whose "optimistic" estimates are worse than the "achievable" estimate of a known solution.

The system seems to have been applied to a rather limited set of examples. It is not clear how general it is, nor is it clear how much new knowledge would be required for it to handle, say, graph search problems. New domains require a great deal of new knowledge in the form of PSI implementation rules, efficiency schemata and so forth.

Kant points out some interesting insights gained from this approach to using analysis to guide the implementation. First, it is easier in general to analyze an algorithm that one is designing than just to take an arbitrary piece of code and try to analyze it. This is because one knows the module structure together with schematic estimates of the costs of each implementation. This is reflected in my proposal in that I assume that the program is given some extra information about the correctness and purpose of the program elements. Kant's analysis requires only substitution and simplification, avoiding the additional difficulty of deciding how to partition the algorithm and what the relevant operations and costs are. Another key insight is that incremental analysis (as a part of the refinement process) is useful because it can give estimates of the costs of *partial* designs, so that some search branches can be eliminated early.

This approach to the use of analytic knowledge is limited, because it does not capitalize on the full power of explanation in guiding the search. That is, LIBRA eliminates search paths because they lead to inefficient implementations, but the analysis doesn't give any help in what to try next. For example, a designer might produce, as a first pass, a design of a classification program (Kant's example) that implements mappings as association-lists. In analyzing the result (or testing it on real data) it may run too slowly because the domain of some mapping is so large that the search time required to access it dominates the run-time. The natural next step is to focus attention on this implementation choice. The designer should backtrack in a dependecy-directed way, rather than a chronological one. It is not clear that Kant's approach would have this behavior.

As the author points out, her approach has other drawbacks. First, it is usually difficult to perform the (global) analysis, and second, it is impossible to get accurate measures of performance of partial designs.

McCartney's MEDUSA system (McCartney, 1987) designs efficient algorithms for computational geometry problems. In fact it has successfully solved 19 such problems (*e.g.* find the intersection of N half-planes in $O(N \lg N)$ time). McCartney's approach adds two things to the stepwise refinement paradigm of Kant's approach. First, the program has some ability, when no "skeletal algorithm" is present which fits the problem, to use domain knowledge to decompose the top level goal (a kind of *ad hoc* problem-solving). The other approach is a body of domain-specific knowledge for applying duality transformations to the problems.

MEDUSA seems to use analytic knowledge in much the same way as LIBRA. It is difficult to judge this relatively recent work; however, the fact that it works on 19 examples is impressive, at least on the face of it. On the other hand, how big is the knowledge base of skeletal algorithms, domain knowledge, and duality knowledge?

4 Some Implementation Ideas

Here, I will try to demonstrate how explanatory structure, both that given with the input and some derived by the optimizer itself, can be used to justify and help locate redistribution opportunities. I anticipate that programs' structure will be represented in some form of the Plan Calculus (Rich, 1981).

4.1 Why Give It Purpose/Correctness Structure?

The most basic reason for assuming that the optimizer is given explanations of purpose and correctness along with the program's structure is that it is an important extra source of information available to an optimizer that is part of a designer (e.g. when I optimize my code, I have access to this information). This extra structure is denied to traditional optimizers, except insofar as they can rederive it. In a study of optimization problems human experts can handle, such deprivation would make the problem artificially difficult.

Optimizations must be proven correct. In the limited, local optimizations currently performed by automatic code improvers, demonstrating correctness is limited to reasoning about a local region of the code. Optimizations depending on global reasoning are not attempted, because proving them correct would require correctness reasoning of comparable difficulty to proving the entire program correct, a formidable task. On the other hand, optimizations that appear global can sometimes be seen as affecting the structure of the correctness proof relatively little. Redistribution optimizations seem to have this property. Consider the mergesort optimization in Section 2. This would appear to be a major change: use of a global variable and side effects to replace a call to the applicative `nth` function. Considered *in vacuo*, this alteration would require a major proof effort to justify it. Given the correctness proof of the entire program, however, it is not such a major change. We are simply storing a value for later use instead of recomputing it.

Another reason for postulating that the optimizer have access to this information is ~~th~~hat it can help *suggest* redistribution opportunities as well as justify them. In the `set-add` example, the correctness of `my-program` could not have actually depended on the representation invariant, "no duplicates." This immediately calls into doubt the need for any code whose purpose is solely to maintain this invariant.

4.2 What is Explanatory Structure?

Answering this question is, of course, a major part of the thesis work; however, I will attempt here to sketch some properties explanatory structure should have.

I anticipate that my first attempt at representing purpose will involve something akin to Rich's (1987) proposed teleological links.

It should relate correctness reasoning directly to modules and program structure. That is, it should show what data properties a given module *establishes* and what properties it *maintains*. It should also indicate the purpose of intermediate results in satisfying a module's specification. For example, if the "purpose" of module A is to make a copy of a data object (maintaining all properties except identity) so that later destructive operations do not destroy the input object, this should be explicit. When A is used in a larger program, the optimizer may be able to show that there is no reason not to destroy the input object to the A instance, so that the original purpose of the module has become superfluous. This would indicate that module A may be removed (replaced with the identity function).

Explanatory structure is more than just "any correctness proof" because arbitrary correctness proofs are not constrained to follow the module structure of the program closely. They also do not explicitly distinguish between establishment and maintenance of properties. This distinction may be important to limiting the set of possible purposes a module serves.

It should be complete enough and rich enough to support incremental reasoning about the correctness of the optimizations. The basic idea is that (1) an optimization can be justified by exhibiting a correctness proof of the resulting program, and (2) the new program can be proved correct by a perturbation of the original correctness proof. Thus, if the original is not complete enough to support this reasoning, this approach fails. On the other hand, this does not imply that an entire correctness proof is required; the system can take advantage of modularity by requiring a complete correctness proof only for those modules whose code is affected by the changes. Thus, the optimization could be shown correct even though the program is not necessarily known to be correct.

This approach to justifying optimizations is quite different from the usual approaches: they reason locally, showing that a small portion of code including the change is equivalent (for correctness purposes) to the code the portion replaces. In particular, global changes to the code (those that involve coordination between points either far apart in the module structure of the program or in the temporal structure of the program's behavior) should be easier to justify using the incremental-total-correctness approach than the local-equivalence approach.

For example, one possible way of justifying some optimization is to say, "I know the original program is correct; the only difference in the new program is that object B is used instead of object A as input to module M under conditions C; I know that all properties required of A for correctness are shared by B; therefore the new program is as correct as the original." This description of the correctness proof should be explicitly represented in the explanatory structure of the new

program. This is desirable because it allows as much of the hierarchical (clean) proof to be retained as possible. The alternative is to break down abstraction boundaries, converting the proof to a lower-level one which is larger and more complex.

It should indicate an alternative for each module. A statement of the purpose of an operation brings with it (at least implicitly) a default natural alternative (commonly, this alternative is "not doing anything"). That is, the purpose of an operation only makes sense relative to doing the default thing. Explanatory structure should make this alternative explicit where possible. For example, "the purpose of doing this copy operation (as opposed to using the original) is ...;" or, "the purpose of adding x to *sum* (as opposed to not adding it) is ..." Note that the latter example is quite different from "the purpose of adding x to *sum* (as opposed to just setting *sum* to x) is ..."

4.3 Detecting Redistribution Opportunities

In addition to using the input explanatory structure to suggest opportunities for redistribution, as mentioned earlier, the optimizer can derive additional explanatory structure for this purpose. My basic idea is that the optimizer must somehow "visualize" (make explicit) interesting aspects of the program's dynamic behavior in order to detect such opportunities. Once this is done, it can compare distinct points (values, intermediate states) pairwise in the behavior representation to see if they are related (e.g. whether they are semantically equal). Purpose plays an important role here because the issue of whether a value will satisfy some other purpose must be settled based on explicit knowledge of the intended purpose.

4.4 Fibonacci Example

For the Fibonacci example, the optimizer's visualization process would expand the recursion one level; show that, since $(n-1)-1 = n-2$, one of the calls to F at the low level calculates the same value as the $F(n-2)$ call at the top level; and introduce machinery to pass that value upward through the subroutine boundary. This process is repeated, showing that the same operation may be performed on the resulting FF, and that FF may then be expressed in terms of recursive calls to itself rather than calls to F.

4.5 Binary Tree Example

Recall the `inc-leaves` program.

```
(defun inc-leaves (tree)
  (cond ((is-leaf? tree)
         (make-leaf (+ 1 (data-field tree))))
        (:ELSE
          (make-internal-node
            (inc-leaves (left-child tree))
            (inc-leaves (right-child tree))))))
```

Assuming that this is used in the inc-leaves-twice context, and that this is the second call to `inc-leaves`, the optimizer knows that it may alter the input tree, if desired, to reuse its pre-computed structure. Thus, in the base case of the recursion (when `(is-leaf? tree)` is true) it may simply alter the data-field of the input and return the input pointer. However, it is not clear from the program representation above that it may in fact return the old tree pointer in the other (`:ELSE`) case as well.

If we create an expanded representation of the program's call sequence, showing the various cases and ordering among them, we can see how such a fact could be noticed. After optimizing the base case to return the portion of the old tree, it is apparent at the next level up in the recursion that both `inc-leaves` results are taken from the old tree; in fact the old tree node dominating those two results will now satisfy the purpose of this (one level up) call. Proceeding upward from the low levels of the call tree, one can show that it is okay to re-use the old tree at all levels. This results in the optimization:

```
....
(defun inc-leaves-3 (tree)
   (cond ((is-leaf? tree) (increment-data-field tree) tree)
         (:ELSE
            (inc-leaves-3 (left-child tree))
            (inc-leaves-3 (right-child tree))
            tree)))
....
```

Clearly, we cannot just create a single, finite representation for all possible call structures of the program, since the input tree may have arbitrarily many nodes. Thus, part of the research is to abstract this behavior into a finite description containing the right information. I envision a structure with (for example) the first call, all next-level calls, an ellipsis marker, a "generic" call conceptually in the middle of the hierarchy, an ellipsis, plus representations of the various base cases. Building and using this structure is, of course, a big part of the research.

4.6 Research Plan

Redistribution of intermediate results can be separated into two subproblems: (1) detecting redistribution opportunities, and (2) implementing code to perform the reuse. In addition to these "performance tasks," the research must also define the representations of purpose and correctness, define the expanded behavior representation for programs, and find an interesting set of examples.

The following is a tentative plan of attack.

- Find examples.
- Define representations of purpose and correctness. Cast examples in terms of this representation.
- Define redistribution methods, such as using global variables, adding hash tables, etc. Implement semi-automatic modules for doing this, including maintaining the correctness/purpose representation across the transformation. This may require revisiting the previous step in an iterative design process.
- Define expanded behavior representation, together with modules to support its use.
- Define and implement method for detecting redistribution opportunities. Again, this may require returning to the previous step.
- Integrate the performance modules, demonstrating that the system can significantly improve programs.

5 References

Aho, A.V., Sethi, R., & Ullman, J.D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. Copyright 1986, Bell Telephone Laboratories.

Cheatham, T.E. (1984). Reusability through program transformation. *IEEE Transactions on Software Engineering*, **SE-19**(5):589–595.

Darlington, J. (1981). An experimental program transformation and synthesis system. *Artificial Intelligence* **16**, pp 1–46. North-Holland.

Fickas, S.F. (1985). Automating the transformational development of software. *IEEE Transactions on Software Engineering*, **SE-11**(11):1268–1277.

Freudenberger, S.M., Schwartz, J.T., & Sharir, M. (1983). Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems,* **5**(1). pp 26–45.

Green, C.C. (1976). The design of the PSI program synthesis system. In *Proceedings of the Second International Conference on Software Engineering.* pp 4–18. Computer Society, IEEE Inc. Long Beach, CA.

Gries, D. (1981). *The Science of Programming.* New York: Springer-Verlag.

Hecht, M.S. (1977). *Flow Analysis of Computer Programs.* New York: Elsevier North-Holland.

Kant, E. (1983). On the efficient synthesis of efficient programs. *Artificial Intelligence*, **20**, pp 253–306. North-Holland Publishing Co.

McCartney, R.D. (1987). Synthesizing algorithms with performance constraints. In *Proceedings of the Sixth National Conference on Artificial Intelligence.* pp 149–154. Amer. Assoc. for Artificial Intelligence. Morgan-Kaufmann, Los Altos, CA.

Meertens, L.G.L.T. (ed.) (1986). *Program Specification and Transformation: Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation.* Amsterdam: North-Holland.

Mostow, J., & Cohen, D. (1985). Automating program speedup by deciding what to cache. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 165–172. Morgan Kaufmann Publishing Co., Los Altos, CA.

Paige, R. (1983). Transformational programming – applications to algorithms and systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages.* pp 73–87. ACM.

Paige, R. & Koenig, S. (1982). Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems* 4(3), July, 1982. pp 402–454.

Partsch, H., & Steinbruggen, T. (1983). Program transformation systems. *ACM Computing Surveys*, **15**(3):199–236.

Pettorossi, A. (1984). A powerful strategy for deriving efficient programs by transformation. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming.* pp 273–281.

Rich, C. (1981). A formal representation for plans in the Programmer's Apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* pp 1044–1052. Vancouver, British Columbia, Canada.

Rich, C. (1987). *Inspection Methods in Programming: Cliches and Plans* Technical Memo AIM-1005, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Scherlis, W.L. (1981). Program improvement by internal specialization. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages.* pp 41–49. ACM.

Smith, D.R. (1985). Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* **27**(1). pp 43–96. North-Holland.

Wile, D. (1981). Type transformations. *IEEE Transactions on Software Engineering*, vol SE-7 (1). pp 32–39.