CMU-CS-80-144

PQCC: A Machine-Relative Compiler Technology

Wm. A. Wulf

Carnegie-Mellon University
Pittsburgh, Pa. 15213
25 September 1980

Abstract

"PQCC" is a research project that is attempting to automate the construction of "production quality" compilers -- ones that are competitive in every respect with the best hand-generated compilers of today. Input to the PQCC technology consists of descriptions of both the language to be compiled and the target computer for which code is to be produced. Output from the PQCC is a PQC, a production-quality compiler. This paper focuses on the techniques used in PQCC that are likely to lead to other forms of "machine-relative" software -- software that is parameterized by the characteristics of a computer system.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Air Force Avionics Laboratory. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1 Introduction: The Message

The *subject* of this paper is PQCC. The *message* of the paper, however, is something else. Lest it be lost in the details of the subject, the message has two parts:

- a notion of "machine-relative-ness" is becoming an effective tool for reducing the cost and improving the quality of non-trivial software systems, and
- several innovations arising in connection with research in "artificial intelligence", AI, and largely ignored by systems and applications programmers until recently, will have a major impact on these latter areas.

On Machine-relative software: The term simply means software that is parameterized by the characteristices of a computer system. This definition is admittedly cryptic, so a few examples may help to more usefully characterize what I mean.

In the simplest case, a program might use conditional compilation facilities to include or exclude configuration dependent modules. Most modern operating systems, for example, have some form of SYSGEN phase. The purpose of SYSGEN is to include only that code needed for the memory and device configuration of a particular system. Thus such an operating system is machine-relative in a fairly trivial sense. A slightly more interesting example arises when the behavior of the system changes in an essential way because of the parameterization. Some operating systems, for example, use radically different algorithms when their SYSGEN parameters specify certain configurations such as the presence or absence of memory management, swapping devices, etc.

Although parameterization *a la* SYSGEN is important, it is not as interesting as a parameterization that spans completely different computing systems. Substantial success has been obtained, for example, in parameterizing mathematical software to the vagaries of the floating point hardware, and software conversion routines, of different computing systems. The parameters to these routines select the algorithms as well as the constants to achieve both accuracy and performance on a given system.

In some cases the parameterization needed to make a system machine-relative may not be conceptually deep (although it may still be difficult in practice); this is essentially the case for machine-relative mathematical software. In other cases the parameterization is substantially more difficult. Consider, for example, a the notion of a "machine independent" diagnostic program -- a program to determine whether the CPU of its host computer system is functioning correctly, and, if not, to determine the likely cause of the failure. Diagnostics embody detailed knowledge of both the processor's instruction set and its implementation; indeed, there doesn't seem to be much to one *except* this knowledge. Thus to construct a machine-relative diagnostic would imply finding a way to encode this considerable, and very detailed, body of information in a useful way.

The rationale for considering the notion of machine-relative software is based on the same arguments as those for portability, standardization of various kinds, methodology, and so on -- namely, *improving quality and reducing cost*.

By writing a system once and hosting it on many machines one can amortize its cost over a much larger number of units. This allows for a greater initial investment for, say, validation, while still reducing per-unit cost. Moreover, all maintenance is similarly applicable to the whole family of target systems. As errors are repaired and improvements are made, all customers benefit. Finally, when new machines become available, software for them can be made almost immediately available.

Obviously, the notion of machine-relative software is closely related to that of portability. They are not the same, however, and neither subsumes the other. A compiler that produces code for a number of different target machines is machine-relative even if it itself only runs on one computer and hence is not portable. Conversely, many portable systems have no need to be parameterized by the computer on which they are running -- in fact, making them machine-relative might be counterproductive. Thus, though similar, portability and machine-relative-ness are complementary techniques.

On using AI techniques: As an outsider, I find many fascinating aspects of research on artificial intelligence; three are relevant to the present discussion. First, some of the programming techniques developed along the path to other research goals have often been as interesting as the research itself. Second, these techniques have become woven into the fiber of computer science, and are often not even perceived as having originated in AI. Third, in many cases the transfer of these ideas to the rest of computer science -- and even more so to application areas -- has been surprisingly slow.

Increasingly, however, the "systems programming" and "applications" areas are tackling problems whose complexity precludes direct algorithmic solution. Both program organizations and heuristics that can cope with this complexity are needed. Machine-relative software, and PQCC in particular, is one example where these techniques are vital. AI systems are typically concerned with complex, ill-structured problems; thus, the techniques developed in connection with these systems are precisely ones for coping with the kinds of problems that, I believe, systems and applications programmers will increasingly face.

2 Introduction: The Subject

The goal of the Production Quality Compiler-Compiler project is to construct a truly automatic compiler-writing system. Input to PQCC consists of (a) a description of the source language to be compiled, and (b) a description of the target computer for which code is to be generated. The

compilers, the PQCs so constructed will be competitive in every respect with the very best hand-generated compilers of today. Automatic generation of parsers has been thoroughly researched; therefore, of particular note is the fact that the quality of the object code produced by these compilers will meet or exceed that of their current, hand-generated counterparts. This is the aspect of PQCC that I will focus on in this paper.

Compilers are relatively complex systems. Compilers that produce really good code are substantially more complex than their less ambitious cousins. Moreover, they need particularly detailed information about the machine for which they are to generate code. Thus, each PQC produced by PQCC is an example of machine-relative software. In fact, the PQCs are rather ambitious examples; the parameterization includes the knowledge needed by an optimizing compiler about its target computer.

One could hand-generate the parameters to a PQC; this might, in fact, be a quite cost-effective method of constructing compilers. However, the PQCC project takes one further step and is generating the parameters automatically from a description of the target computer[1].

It is difficult to describe all of the PQCC technology in a single paper; I shall not even try. Rather, I shall describe something of the overall structure of a PQC and sketch some of the key algorithms and how they are parameterized to make them machine-relative. This in turn will allow us to sketch the process by which parameter derivation is automated.

Before proceeding with the meat of the paper, I should note that a more complete overview of PQCC may be found in [lev80]. Also, PQCC is not the only current research effort with the goal of constructing machine-relative compilers. The papers [aho80, johnson80, graham80, lev80] in the August 80 issue of the IEEE Computer magazine survey much of the current state of the field and provide references to other contemporary work.

3 The PQC: A Knowledge-Based Structure

To the user, a compiler is usually a single "black box". As in Figure 1a, input consists of source text and output is (relocatable) object code. Typical texts on compiling, e.g., [ahoull77], reveal a more detailed view; as shown in Figure 1b, a student is told that a compiler is composed of 3 phases that successively transform the source text through various intermediate forms into the final object program. (Often an optional fourth, "optimizing" phase is also described. This is shown as a dotted

[1] This statement is a bit misleading. A number of (parameterized) decisions *must* be made by a human because they are in fact forced by external factors. Examples of such decisions include system-wide conventions on the use of specific registers, library and operating system calling conventions, and so on.

box in the figure.) The structure of a PQC consists of a much larger number of phases -- around 50, in fact.

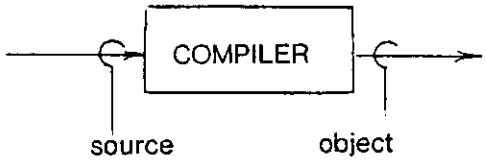


Figure 1(a): User's View of a Compiler

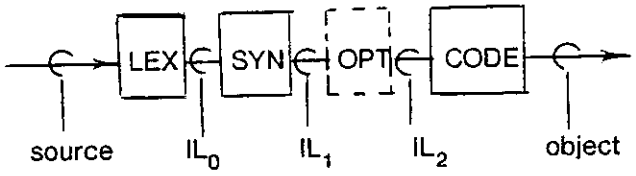


Figure 1(b): Textbook View of a Compiler

An obvious question is "why does the PQC have so many phases", and, possibly, "isn't that terribly inefficient". These questions go right to the heart of several important issues in the PQCC technology. To answer them, it is easiest to go back and ask the same questions about the typical textbook view of a compiler.

In principle, it is possible to use a single, very general, very powerful technique to effect the translation performed by any compiler -- including all the transformations done by the fanciest optimizing ones. Any of the "production systems" equivalent to general recursion is, for example, a conceptually plausible method. If such a formalism were used directly, there would be no reason to decompose the model of a compiler into components. There are at least two reasons, however, why this isn't done: (a) human understandability, and (b) efficiency, or, to use a more accurate term, cost-effectiveness. Much has been written in the programming-methodology literature on the first of these, so I will avoid these arguments and concentrate on the second point.

All compiler writers are familiar with the fact that there is no real necessity for separating lexical and syntactic analysis -- and, in fact, the division between them in a particular compiler is chosen somewhat arbitrarily. The more general methods used for context-free syntactic analysis could be used to do all the work of the lexical phase. Because these methods are slower than the finite-state methods that can be used for lexical analysis, however, the speed of the total system is improved by their separation.

This is a simple example of a more general phenomenon, and one that has been especially noticeable in AI research. Much of the early research in AI focused on general, "powerful" methods

of problem solving. In realistic, complex task domains, however, these methods are vastly too slow to be practical. A common characteristic of these task domains is that they require both many kinds of knowledge, and deep knowledge of the domain to be applied. General methods do not contain this knowledge in any direct sense. Loosely, general methods always work from "first principles"; they know "nothing in particular and everything in general". Thus, when faced with a specific task, general methods must "rediscover" things that would be obvious to a human expert. In these complex tasks it has been found that a collection of complementary methods, or "experts", is much better than more general, powerful methods. Systems organized around this scheme are usually referred to as "expert systems" or "knowledge-based systems".

An "expert", is simply an algorithm or a heuristic with a narrow domain of application -- but which embodies a great deal of specific knowledge in that domain, and hence is very effective on problems within it. In contrast with a "general", or "powerful" method, an expert simply cannot solve most problems. On the other hand, those that it can solve, it solves well and cheaply. Lexical analysis based on finite-state machines is an expert method: there are a large number of parsing problems it cannot solve. On the other hand, it solves a subclass of the problems of interest more cost-effectively than a more general method could.

It is unlikely that the typical compiler writer or textbook author thinks of the 3 (or 4) phases of the canonical compiler diagram as a collection of expert methods. When writing a compiler for a specific source language and target machine such a conceptualization is unnecessary. Nonetheless, even in that simple case, the phases are in fact precisely such a collection. In the case of PQCC, however, the knowledge-based, "expert", conceptualization was essential. Only by carefully analyzing existing compilers and explicitly decomposing them into their constituent component methods was it possible to find efficient and parameterizable pieces that could be made machine-relative.

So much for "why" the decomposition into phases. The issue of efficiency for the 3 phase compilers is also clear -- they *are* more efficient *precisely* because the decomposition allows more cost-effective algorithms to be used. The answer for the PQC structure *should* be the same; it *should* be more efficient than the 3-phase structure. Unfortunately, the data is not yet available to support that claim. Indeed, the present research version of the PQCs makes a separate traversal of the program representation for each phase; although its speed is quite respectable, it is not necessary to make these separate traversals and so it is hard to predict final performance.

4 Some "Experts" in the PQC

In this section I shall consider some simple examples of code generation and use them to illustrate, if not motivate, the decomposition of the PQCs and some of the expert methods they use. These in

turn will allow us in subsequent sections to look in more detail at some of these methods and how they are made machine-relative.

Let us consider one of the simplest, yet interesting statements that can appear in a source program. Almost all languages have an assignment statement and integer variables; so, suppose *i, j* and *k* are integer variables and consider the statement

<var> : = i-j;

where *<var>* is one of *i, j* or *k*. The "best", indeed any correct code to produce for this statement is obviously a function of the target computer. But, in addition, it also depends upon:

- whether the expression "*i-j*" is a common subexpression, and, if so, whether it is a creation or use of that subexpression, and
- the variable appearing on the left-hand side of the assignment, and
- the allocation of storage for variables, and, in particular, whether any of the variables involved has been allocated to especially "interesting" locations (e.g., registers, a short displacement from the stack-frame pointer, etc.).

Let's illustrate the dependencies by considering a few examples of code generation for some real computers -- the PDP-11 and the PDP-10[2]. Since the impact of common-subexpressions is often discussed, let's leave them out for now. The effect of the left-hand side of the assignment can be clearly seen in the code to be generated for a PDP-11:

source	object
k : = i-j;	MOV i,k SUB j,k
i : = i-j;	SUB j,i
j : = i-j;	SUB i,j NEG j

Notice that in the third case the right-hand side is being treated as the expression "*-(j-i)*"[3]; I shall return to this point later. Because the operands of PDP-11 instructions can be in either memory or registers, the examples above do not illustrate the effect this allocation decision has on code generation. Suppose, therefore, that we were compiling for the PDP-10 and that all variables had been bound to primary memory. We would get:

---

[2] I apologize to those readers that are not familiar with the PDP-11 and PDP-10. Knowledge of these machines is not really necessary. The most important thing to notice about the examples is simply that they are *different* from one another.

[3] For the esoterically minded, I appreciate that there is a problem with this transformation arising in connection with the "most negative number" on two's complement machines. I chose to ignore the problem to simplify the discussion.

source	object

k : = i-j;	MOVE r,i
	SUB r,j
	MOVEM r,k

i : = i-j;	MOVN r,j ADDM r,i
j : = i-j;	MOVE r,i SUBM r,j

If, on the other hand, all three variables were allocated to PDP-10 registers, better code would be:

source	object
k : = i-j;	MOVE k,i SUB k,j
i : = i-j;	SUB i,j
j : = i-j;	SUBM i,j

And, of course, one can get all sorts of combinations and variations on these when only some of the variables have been allocated to registers; I won't try to reproduce them all here. It should be noted, however, that they are not all simple variations on the examples above. If, for example, *k* has been allocated to memory but the other two are in registers, then we would like to produce:

source	object
k : = i-j;	MOVEM j,k SUBM i,k

Each of the cases above would be "obvious" to a knowledgeable PDP-11 or PDP-10 assembly language programmer. Simple compilers, and even many optimizing ones, do not apply this "obvious" knowledge, however. Why?.

To answer this question, it is instructive to consider the kinds of knowledge that that programmer uses in making these selections. They include, at least,

- *language semantics*: although these examples are trivial, one must at least be sure that the variables are not *aliases* for one another -- otherwise the destruction of *k* in the cases for "*k : = i-j*" might also destroy either of the other two variables and invalidate the code sequences shown above,

- *algebra*: again, these examples are trivial, but they do use the identity

  $$i-j = -(j-i),$$

- *global resource allocation*: a computer's registers are a limited resource: usually not all of the programmer's variables can be allocated to them simultaneously, for example.

- *global resource allocation*: a computer's registers are a limited resource; usually not all of the programmer's variables can be allocated to them simultaneously, for example. Deciding which variables are to be allocated to the register involves some complex, global tradeoffs,

- *target computer instruction set*: knowledge of the instruction set is obviously needed, of course, but notice that it is much deeper than merely knowing *some way* to implement each language construct. In particular there is a knowledge of the best choices under various allocations, and "best" implies some knowledge of relative costs.

There are in fact quite a few other kinds of knowledge that an assembly language programmer will bring to bear -- but they are not revealed by these trivial examples. Discussing too many at this point is likely to be confusing in any case, so I shall omit them.

Now, let's return to the question of why most compilers don't generate the "obvious" code sequences illustrated above. If one starts with the textbook view that "code generation" is a single component, any attempt to incorporate all these diverse sources of knowledge is bound to lead to either an inefficient, "general" method, or to an unmanageably complex collection of special cases. Since general methods are too slow, (too) many optimizing compilers tend toward the second alternative. Faced with the existing evidence, researchers have shrunk away from the notion of a compiler that is both optimizing and machine-relative. Starting with a different premise, however, namely that each knowledge source is a distinct expert, leads rather directly to a clean and efficient structure -- *and*, from our perspective more importantly, one that is more easily made machine-relative.

The bulk of the remainder of this paper is devoted to describing a number of the expert methods used in the PQCs -- and how, given this information, we can make them machine-relative. Before starting, however, I need to make one more point that would have been difficult to motivate earlier. Notice that the application of the various kinds of knowledge *interact* with each other. To take a simple example -- the best choice of code depends upon the allocation of variables to memory and registers. At the same time, however, the best allocation of memory will depend on the code to be generated -- in general, for example, one would like the most frequently accessed variables to also be the cheapest to access. To know the actual number of references, however, I need to know what code will be generated.

The potential for interaction among the various expert methods used in a system requires careful consideration for how they are to be organized -- that is, the overall control and data structures to be used. The simplest such organization is a linear sequence of "phases" as is used in the canonical 3-phase compiler, and this is also, in fact, the organization of the PQCs -- but the reader should be aware that other possibilities exist and may be appropriate for other kinds of expert systems[4].

---

[4]The interested reader might, for example, examine [erman]

The next several sections discuss three of the phases in a PQC: CODE, TNBIND, and UCOMP. Although it may be unusual, we shall discuss them in the the reverse of the order in which they appear in the compilers.

5 CODE: The Code-generation Expert

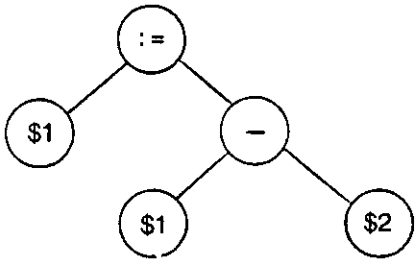
CODE is one of the last phases of a PQC. In contrast with the complexity that may seem necessary on the basis of the earlier examples, it is also a conceptually trivial phase. In the following I will only try to provide the gist of the scheme used; details may be found in [cattell78].

All of the critical optimization decisions in a PQC are made before code is generated. Even the key aspects of the relatively low level decisions illustrated in the previous section have been made: registers have been allocated, the decision to use a user-variable for the evaluation of an expression, the algebraic transformations such as *(i-j) => -(j-i)*, and so on have all been made. These decisions are all represented in the intermediate representation of the program by either explicit transformations (such as the algebraic identity) or by decorating the tree with relevant information. The code generator has only to match this information against a "canned" collection of instruction sequences. Thus, the major problem in the formulation of the code generator was to find a technique that was both comprehensive and efficient with respect to the enormous case analysis involved in the matching process.

The basic strategy used in the PQC code generator is based on a database consisting of *pattern-action* pairs. The patterns, like the intermediate representation of the program, are trees; the leaves of these pattern trees specify properties of the program tree that they match. The actions are (usually) simply code to be emitted. Writing the pattern trees in a LISP-like prefix form, typical pattern-action pairs for the PDP-10 might be:

```
pattern: (:= $1:memory $2:register)
action:  MOVEM $2, $1

pattern: (:= $1:register $2:memory)
action:  MOVE $1, $2

pattern: (- $1:register $2:register)
action:  SUB $1, $2

pattern: (:= $1:register (- $1:register $2:memory))
action:  SUB $1, $2

pattern: (:= $1:memory (- $1:memory $2:register))
action:  SUBM $2, $1
```

The patterns above may require a bit of explanation. Each pattern describes a tree. The first item is an operator at the root node of the tree. The remaining items describe the descendants of that root; the dollar signs followed by a digit are merely names for the descendants and the words like "register" describe properties of them. In the last two examples, the second descendant of each is paranthesized; this indicates a sutree of the same form as described above. Thus, for example, the fourth pattern describes the tree



where the value represented by the node called $1 must be in a register and that represented by $2 must be in memory. Notice, too, that the repeated use of $1 implies that these must actually name the same location.

CODE traverses the program tree. At each node it finds *all* pattern trees from the database that "match"; it then selects the lowest cost, maximally beneficial code sequence. In general, of course, the pattern trees will contain more than one node; this has two implications for CODE. First, it must "mark" all nodes matched by the selected pattern and must not subsequently generate code for nodes so marked. Second, and very importantly, CODE generates code "backwards".

This second statement may be a bit surprising, however, the PQC code generators actually generate the last instruction of the program first. To see why this is so, let's first note that proceeding "forward" through the program representation is equivalent to a bottom-up, left-to-right tree traversal; proceeding "backwards" is equivalent to a top-down, right-to-left traversal. Now, consider the tree for something like one of our example assignments,

i : = *i*-*j* .

If we were to use the "forward", bottom-up traversal, the first interesting node that we would encounter would be the subtraction, and we would generate code for that node from a pattern like those given above. We would not see the larger context, in which *i* appears as the left-hand side of the assignment, until too late. Instead of generating code to compute the result directly in the variable, some sort of temporary would be needed and an additional instruction would be generated to store the temporary into *i*.

On the other hand, by traversing the tree "backwards", that is, top-down, we see these larger contexts first and can always apply the maximal pattern[5].

---

[5]Cattell, [cattell78], named this strategy, the "maximal munch" method of code generation.

Although the previous explanation has been brief, hopefully it conveys the gist of how the code generator works. There are, of course, myriad technical details. Rather than delve into these, however, let's reflect on what has actually been done. I set out to build an expert code generator, but that is not quite how it turned out. Instead, all the expertise about the target computer wound up in the database of pattern-action pairs. The actual algorithm of the "code generator" is an expert at something else entirely -- namely pattern matching. Because of previous work, *cf.* [forgy79], and because the particular kind of patterns is well specified, *very* efficient algorithms exist -- and, the result is a code generator that is machine-relative, more comprehensive and, hopefully, faster than typical hand-fabricated ones that do comparable analysis.

6 TNBIND: The Allocation Expert

In this section I shall describe a simplified version of the phases that implement register allocation within a PQC. For historical reasons this set of phases is called TNBIND, a contraction of Temporary Name Binding. As with the previous section on CODE, the objective here is to supply only enough detail to convey the gist of the scheme; to that end, I will consider only the allocation of user-declared variables and neglect compiler-generated temporaries. Details are available in [lev80thesis].

As noted earlier, good allocation of registers and memory depends in part on the code to be generated. It should be clear, for example, that the frequency with which a variable is used affects the desirability of allocating a register for it; *usually*, though not always, it's a good idea to keep the most frequently accessed variables in easily accessible locations like registers. In addition, however, the *way in which a variable is used* is an important factor in determining the best allocation for it.

The distinction between "registers" and "memory" is vastly too crude to be of much use for allocation of resources on most real computers. Even the "nicest" common machines have restrictions such as:

- only certain registers can be used for indexing, multiplication, division, etc.
- certain locations are data-type sensitive -- as, for example, the "floating point" registers on many common machines,
- some memory locations can be addressed more cheaply, as for example, with short displacements in the instruction word,
- some memory locations can be accessed *only* by indexing, by loading a base register, or some other awkward means.

Because of the restrictions such as those above, we clearly need to know how variables will be used in order to make good allocations. Floating point variables, for example, should probably be allocated to floating point registers if possible. Not all allocations are this simple, of course. An integer variable,

for example, may be involved in both arithmetic and indexing; on an architecture where these involve disjoint registers a tradeoff must be made. Making this tradeoff intelligently requires knowing something about the code that will be generated -- the number of accesses to the variable in each of the contexts and the cost of moving the variable between them.

Bothersome as they may be, the sorts of assymmetry present in the list above generally reflect difficult technological tradeoffs in machine design. It is the compiler-writer's job to make the best possible use of the given architecture[6]. So, with that attitude, and with the knowledge that code generation will follow allocation -- even though allocation depends upon it -- need a formulation of allocation that will make extremely good, if not perfect, use of the machine's resources.

The first step in this formulation is simply to divide the possible locations into a number of *storage classes* -- each class consists of the set of locations that have the same properties, and members of different classes have different properties. Thus, for some machines the even registers and odd registers will be in different classes; on others there may be a distinction between integer and floating registers, between index registers and accumulators, between stack memory and other memeory, and so on.

At least conceptually, TNBIND uses the same database and algorithm as CODE to generate *all possible* code sequences for the program -- or, more precisely, all code sequences that would result from various allocations of variables to the different storage classes (but not to specific members of those classes). Thus, conceptually, TNBIND generates one code sequence that assumes all variables have been bound to *some* register, another that assumes all variables have been bound to *some* memory location, and one that assumes each of the combinations in between. The idea is to then pick the cheapest *feasible* code for the complete program. Note that the cheapest code sequence found by TNBIND may not be feasible because it requires more elements of a storage class, e.g. the registers, than are present in the hardware.

In practice, of course, we cannot generate all possible code sequences for a program -- but we don't need to in order to get the same net effect. Instead, we associate a table of costs with each variable. These per-variable tables quantify the incremental contribution to overall program size based on its allocation; in effect, they say

> ... if this variable is allocated to an even register the incremental cost to the overall program will be C_e, while, if it is allocated to an odd register the incremental cost will be C_o, while, if ....

---

[6]Which is not to say that it isn't possible to do a better job of instruction set design; quite the contrary is true. The author advocates that all computer architects should be *required* to implement two (2) optimizing compilers -- one just as practice before they design a machine, and a second one for the machine that they design.

To implement this conceptual model of TNBIND, as is the case in all of the PQC, we first break the problem into pieces such that we can devise an expert for each. I shall describe three sub-phases here[7]: temporary name assignment (TNA), lifetime determination (LIFE), and packing (PACK).

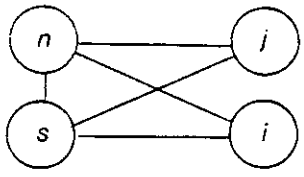
The first subphase, TNA, accumulates the per-variable cost information. It uses almost the same database and algorithm as does CODE. Because it does not actually generate code, however, there are two differences in the database. First, those patterns that differ only in the allocation properties of their leaves are merged. Second, the code-emission actions are replaced by a table of incremental costs for each of the variables involved in the pattern. These tables contain one cost for each of the allocation possibilities and are derived from the (merged) code sequences used in CODE. Thus, they accurately reflect the actual number and types of references to each variable.

TNA traverses the program tree, just as CODE does, matching the patterns in its database against the tree and finding the best pattern at each step. Once found, the cost tables associated with the pattern are added to those being maintained with each of the variables in the matched portion of the program tree.

The second subphase determines the "lifetime" of each of the variables -- that is, the set of points in the program during which the value of the variable is valid. By noting when a variable is "alive" we can construct a "conflict graph" -- a graph whose nodes represent variables and whose arcs indicate that the connected nodes must be alive simultaneously. In a language in which loop control variables are implicitly declared, a program fragment such as

```
var n,s,j: integer;
var A: array(1..n) of integer;
...
n := j-1;
s := 0;
for i := 1..n do s := s + A(i);
j := s-4;
...
```

gives rise, in part, to a conflict graph such as



Notice, in particular, that *j* is used prior to the loop and set after it. Thus, its value need not be valid during the loop. This fact is captured in the graph by the absence of a link between *i*, which is alive only during the loop, and *j*. These two variables do not conflict.

---

[7] In the actual PQC, allocation is accomplished by something more like 10 subphases.

Although this graph is incomplete because the complete program is not given above, it can nonetheless be seen that variables *j* and *i* can be allocated to the same location because their lifetimes do not overlap. Similarly, it can be seen that *n,s* and *i* must be allocated to different locations because their lifetimes do overlap. Lifetime analysis is a simple flow analysis problem for which a great deal of literature is available, therefore I won't dwell on it here. Notice, however, that it is a completely language and machine-independent process.

The third subphase of TNBIND that I shall describe is PACK; its function, given the cost and lifetime information from the previous phases, is to find a good, if not optimal, feasible allocation. It can be cast as a "graph coloring" problem. If one thinks of each distinct location as a distinct color, then

- A *feasible* coloring of the graph is one in which no two connected nodes are the same color, and
- An optimal coloring is a feasible one in which the sum of the costs associated with each particular node coloring is minimized.

Again, a rich literature exists on graph coloring algorithms. Although general solutions to coloring problems are very costly, the literature provides a number of alternative algorithms for the kind of special situation here -- and in practice they are both fast and effective. Note, in particular, the only dependency in these algorithms on the target computer is the number of "colors" (locations of various kinds) available, and the mapping from colors to costs.

As with our description of the code generator, rather than delve into the next level of technical detail on TNBIND, let's reflect on what was actually done. I set out to construct an expert on allocation -- in fact, an expert on allocating variables for a particular machine. As with CODE, however, all of the expertise about the specific machine is encoded in the patterns and cost tables used by TNA. The three algorithms turn out to involve pattern matching (TNA), flow analysis (LIFE) and graph coloring (PACK). None of them has anything to do with allocation per se, and certainly none of them is machine specific. Finally, as was also the case with CODE, I have very particular cases of matching, flow analysis and coloring. Fast and effective algorithms exist for them, and there is every reason to expect that they will perform as well or better than the ad hoc techniques that have been used in the past.

7 UCOMP: A Simple Algebraist

In the last section I noted that good register allocation depends on knowing something about the code to be generated. In this section I will describe one of the phases that precedes TNBIND and provides information on the nature of the best code to generate. The phase is called UCOMP, which is a contraction of "unary-complement optimization", one of the primary functions of the phase.

UCOMP's responsibility is to apply various algebraic transformations to the program; the objective of these transformations is to simplify the computation of arithmetic and logical expressions. For example, UCOMP will convert the expression

(-y) > (x(a-b))*

into

*(b-a)*x > y*.

Our code generator, and indeed any code generator that is to be relatively fast, is dumb. It would probably generate 6 or 7 PDP-11 instruction for the first form of the expression, for example. But even the dumbest of code generators will generate good code for the transformed expression; for example,

source	object

*(b-a)*x > y*	MOV b,r
	SUB a,r
	MUL x,r
	CMP r,y

Algebraic simplification is a theoretically unsolvable problem. Even to do a good, not perfect, job is extremely hard. A PQC, however, does not require general simplification. It requires only that we simplify the *computation* of the expression's value -- that is, reduce the number of instructions needed to evaluate it. As a result, the complexity, such as it is, arises from the need to account for the the vagaries of the target instruction set rather than from the inherent difficulty of algebraic simplification. The kinds of properties of the instruction set that affect us include:

- Operands of certain instructions may be required to be in special locations. Most general-register machines, for example, require one operand to be in a register.
- The instruction set may not be complete. Some computers, for example, provide an incomplete set of conditional branch instructions.
- Certain "obvious" instructions may not be present. The PDP-11, for example, does not contain an *and* instruction; its BIC instruction is actually an *and-not*.
- Certain operations, particularly unary ones, may be "free". Many machines, for example, include *load-complement* and *load-negative* operations. In such cases, the complementary *store* instructions may or may not be present.

In the remainder of this section I will show how these properties of the instruction set affect the kinds of transformations performed by UCOMP. In particular, I will use the class of *unary complement* and *commutativity* transformations on algebraic expressions to illustrate the process; these ideas extend naturally to boolean expressions and to related transformations.

Before beginning I need to make two observations that affect the approach.

- First, the *destroyability* of the operands of an operator affect the choice of transformations. User variables and the values of common subexpressions, for example,

are usually not destroyable, while the results of other subexpressions normally are. The locations in which a destroyable result resides can often be used profitably for the computation of its parent expression; non-destroyable locations cannot.

- Second, in many cases the compiler is free to compute the negative of the value of an expression rather than its correct value. In the example at the beginning of this section, for example, the value *(b-a)* was computed rather than *(a-b)*. Doing so allowed the compiler to avoid computing *(-y)*, and to modify the relational operator instead.

Now, let's turn to the question of how UCOMP works. It consists of two passes over the program representation. The first of these selects a limited set of transformations that *could* be applied at each node; in fact, it selects just two possibilities, one that would compute the expression's correct value and one that would compute the value with an inverted sign. Each of these transformations represents the least expensive way to compute the value with the indicated sign. The second pass chooses which transform to apply, and does it.

The reason for two passes is obvious, once you see it. Consider the example at the beginning of the section again. At the time that UCOMP's first pass examines the subexpression *(a-b)* it can easily detect, in a manner to be described below, that *(a-b)* is the cheapest form that will create the value with the correct sign; similarly, it can detect that *(b-a)* is the cheapest form for computing the value with the inverted sign. In all probability, however, the cost of evaluating these two expressions will be the same; there is no basis for choosing between them. It is only in the (much) larger context of the relational operator that its possible to see that computing the inverted signs of both *(a-b)* and *(-y)* will produce globally better code.

The task of the first pass can be characterized as simply associating nine pieces of information with each node of the program tree:

Kp	the cost of producing the value of the expression represented by the node with its correct sign,
Tp	an encoding of the transformation, if any, to be applied to the node in order to produce the value with its correct sign,
Slp	a boolean that specifies whether Tp assumes the correct or inverted sign of its left operand (or only operand in the case of unary operators),
Srp	a boolean that specifies whether Tp assumes the correct or inverted sign of its right operand (ignored if the operator is unary),
Kn, Tn, Sln, Srn	similar quantities for producing the value of the expression with inverted sign, and
D	a boolean that indicates whether the value represented by the node is destroyable.

This information is gathered on a bottom-up traversal of the tree with the aid of an auxiliary table. The auxiliary table contains an encoding of the transformations that are possible, together with the incremental cost of applying them. In effect, these tables are an encoding of normal algebraic axioms that have been augmented with cost information that is derived from the code productions used by CODE. The encoding is such that it is directly in the form that becomes attached to the nodes, and is guaranteed to include at least one way to produce both the correct and inverted sign for each operator that can appear in the program tree.

In general, of course, there will be several applicable transforms; notably different transformations will make different assumptions about the sign (correct or inverted) of its operands and about their destroyability. Since the first phase of UCOMP traverses the tree bottom-up, these properties of the operands of a node are already known when the best choice(s) for that node must be made. Hence, the phase computes the total cost, that is the sum of the costs for the operands and the incremental cost from the auxiliary table, for each applicable transformation and chooses the cheapest. It then records these choices in the node and continues the traversal.

It is worth noting that the actual information recorded is very small -- about 18 bits in the current implementation. The transformations, for example, all involve only simple things like negating an operand, commuting the operands, and negating the node itself. Thus, although the term "transformation" may suggest something complex, for this "expert" it really connotes something very simple.

Given the information gathered by the first pass, the second pass is very simple too. It is a top-down traversal. At certain points, like assignments and parameter passing, the semantics of the language usually force the compiler to generate the correct sign for the value of an expression. At these points the phase selects one of the alternatives. This choice, in turn, forces the choice of sign (correct or inverted) on its immediate descendants (recall, this is encoded in Slp, Sln, etc.). These forced choices are passed down, thus forcing similar choices on their descendants. The actual transformations are applied as the phase backs out of the traversal.

At the risk of being somewhat repetitive, let's once again consider what has actually been done in this phase. I began with what appeared to be an extremely difficult problem, algebraic simplification constrained by the anomalies of target computer instruction sets. I did not need to solve the general problem, however. Instead, I found a way of creating an "expert" on a narrow subproblem involving only commutativity and unary-complement optimizations. This expert could easily be endowed with the requisite knowledge of the target machine -- in fact, all it needed was a table of costs corresponding to the incremental cost of some simple instruction sequences. The result is easy to comprehend, is machine-relative, is substantially beyond the capabilities of most existing optimizing compilers, and is fast.

8 GEN: The Code Generator Generator

In the previous sections I focused on some phases of a PQC. The *message* of these sections was that the knowledge-based, expert system model, an idea from AI, provided a framework within which the solution to an apparently very difficult problem could be effectively and efficiently constructed. The choice of phases described in these sections was not accidental; partially they were chosen because they could be explained in isolation, but they were also chosen as preparation for the present section. They all involved using the knowledge encoded in CODE's pattern-action pairs in a fairly direct way. In this section I will turn to a portion of PQCC, rather than PQC, technology and consider how these pattern-action pairs are automatically derived from a formal machine description. This, in turn, will expose another set of ideas borrowed from AI research. As with the previous sections, I will be forced to merely sketch the relevant ideas; details may be found in [cattell78].

The program that generates CODE's pattern-action pairs is called GEN. Its basic problem is to convert a formal machine description into these pairs. Given the pairs, deriving the information for TNBIND and UCOMP, as well as some other phases of the real PQCs, is relatively simple.

The input to GEN consists of two parts: a machine description and a set of axioms. The axioms used are primarily those of ordinary arithmetic and logic. For example, they include

$$E \equiv -(-E)$$
$$E + 0 \equiv E$$
$$B \equiv \neg\neg B$$
$$B1 \wedge B2 \equiv \neg(\neg B1 \vee \neg B2)$$
$$\neg(E1 > E2) \equiv E1 \leq E2$$

where the E's denote arithmetic expressions and the B's denote logical ones. In addition, there may be (for specific machines) axioms that, for example, relate logical and arithmetic quantities; on a two's complement machine, for example,

$$-E \equiv (\neg E) + 1$$

Other axioms capture essential notions such as sequencing and branching.

The formal machine description used by GEN contains details on the storage classes (e.g., the kinds of registers and memory), effective address computations, and so on. Of particular relevance to the present discussion, however, is that it contains a detailed description of the machine's instruction set. This description is in the form of "input/output assertions" associated with each instruction. Input/output assertions are an idea borrowed from the research on formal semantics of programming languages, and are often written in the form

```
P1 { C } P2
```

where P1 and P2 are logical formulae, and C is some statement in the programming language. Such a formula is read

> If the state of the program (variables) is such that the predicate P1 is true *before* executing C, then the state of the program (variables) *after* executing C will be such that P2 is also true.

So, for example, the formula

$$x = 1 \{ x := x + 1 \} x = 2$$

asserts the (obvious) fact that if x has the value 1 before the assignment is executed, it will have the value 2 afterwards.

The instructions of most computers will operate correctly under all circumstances, so the precondition, P1, is generally not needed and I will elide it in the following. So, for example, some of the instruction of the PDP-11 can be characterized as follows:

$$\begin{array}{ll} \{\text{CLR dst}\} & dst = 0 \wedge N = 0 \wedge Z = 1 \\ \{\text{MOV src dst}\} & dst = src \wedge N = (src<0) \wedge Z = (src = 0) \\ \{\text{SUB src dst}\} & dst = dst'-src \wedge N = (dst<0) \wedge Z = (dst = 0) \\ \{\text{BIC src dst}\} & dst = (dst' \wedge \neg src) \wedge Z = (dst = 0) \end{array}$$

where N and Z are two of the 11's "condition code" bits, and the prime on dst (dst') in the definition of SUB denotes the value of dst before the instruction is executed.

The implementation of GEN consists of two parts. The first, called SELECT, proposes an "interesting" set of program trees for which it would be nice to have code-generation patterns. This set includes at least one tree for each construct in the source language, but in addition includes other trees that may have interesting implementations. SELECT knows, for example, that literals, and especially small literals like 0, 1, -1, etc., often give rise to special cases that can be handled more efficiently. The second part, called SEARCH, finds alternative implementations for each of the trees proposed by SELECT and emits pattern-action pairs for the most efficient ones.

SEARCH is the more interesting part of GEN, so let's look at it in more detail. Suppose that we are attempting to generate a code generator for the PDP-11, and further suppose that SELECT has proposed the tree

$$a := b.$$

The postcondition of this statement is simply $a = b$, and is therefore implied by the postcondition of the MOV instruction above[8]. Thus, MOV is a legal implementation of the proposed tree, and in fact in this case is probably the only such. This is the simplest kind of situation for SEARCH, and, in effect, can be done by simple pattern matching. In particular, elaborate theorem proving is not needed.

Now let's consider a slightly harder example. Suppose we are still trying to create a code generator for the PDP-11, and that SELECT proposes the tree

---

[8] Of course, the MOV instruction does more than is needed since it also sets the condition codes -- but I will ignore that here.

a := b-c.

Because the variables are all different, no postcondition on an instruction will directly imply that of the proposed tree. In such a case, SEARCH, among other things, will try a classic AI technique called "means-ends" analysis. In effect, it will note that the postcondition for the SUB instruction *almost* works -- the only problem is that SUB requires that the left-hand side of the assignment and the left operand of the subtract be in the same location. Therefore, SEARCH sets up a "subgoal" for itself, namely to get these to be the same. The subgoal is, of course, just the previous program tree,

a := b.

By applying itself recursively to this subgoal it will find the sequence

```
MOV b,a
SUB c,a.
```

Means-ends analysis supplies much of the "smarts" needed by SEARCH, but it is not enough. Suppose that SELECT had proposed

a := b and c

and that we are still targeting for the PDP-11. In this case, the closest match will be with the BIC instruction, but there are two problems. One is the same as discussed above; we must get the left operand of the *and* into the destination location. The other is that BIC complements its source operand. While means-ends analysis will solve the first problem, the second requires applying the axioms of logic. SEARCH will find that, if it applies

$B \equiv \neg\neg B$

to the right argument, and then sets up the subgoal of

t := not b

it can produce the code

```
MOV c,t
CMP t
MOV b,a
BIC t,a.
```

(For those unfamilar with the PDP-11; this sequence first moves *c* into a temporary, *t*, and then complements *t*. It then moves *b* into *a*, and finally does an and-not, BIC, from *t* into *a*.)

GEN does not, of course, stop after it finds a single instruction sequence for one of the goals proposed by SELECT. Instead, it generates all the sequences that it can within specified time limits on its search. Although I did not explicitly indicate it above, the machine descriptions also include cost information. SEARCH uses this to remember only the least costly sequences, and these are the ones emitted for use by CODE, TNBIND, etc.

From a strictly theoretical perspective, GEN is not guaranteed to find the optimal code sequence for the trees proposed by SELECT. Even if it were to run indefinitely long this would be true, but the fact

that a limit is placed on its search makes this obvious. As a completely practical matter, real instruction sets are both simple and quite complete. As a result, in our experience, GEN has always found what I believe to be optimal sequences -- typically in less than a tenth of a second per construct. Since only a few hundred such constructs are usually needed, the total time to generate the database for CODE is on the order of a few minutes.

There are things that GEN cannot do. I alluded to some of them earlier; it cannot, for example, make decisions about things like calling conventions or register conventions that are actually dictated by external factors. The current implementation, in addition, cannot cope with elaborate control constructs; it will not, for example, discover the transfer-vector implementation of case statements. Nonetheless, it does do an excellent job of the more mundane, tiresome, and error-prone aspects of generating code sequences for the bulk of constructs. Moreover, the nature of the pattern-action pairs is such that the output of GEN can be augmented manually for these few difficult cases.

9 Summary

PQCC and the resulting PQCs are concrete examples of both the notion of machine-relative software and the impact of AI techniques on a practical application area. There is, of course, much more to PQCC than either of these; it involves, for example, a lot of compiler technology and solid engineering. In addition, as should be evident from the material presented here, important contributons have been borrowed from the more theoretical side of computer science: formal semantics, and algorithm design (for graph coloring and flow analysis) are two examples of this.

PQCC is a relatively mature research project. I expect a complete prototype PQCC system to be operational by early 1981 and to have generated a number of PQCs with it by that time. As of this writing, all of the phases have been formalized and something like two-thirds of them are operational. Preliminary comparisons of the operational phases indicates that they are slightly larger and slower than the corresponding components of comparable hand-generated compilers. However, the differences are small -- and are mostly due to an artifact of the research strategy. "Eyeball" analysis suggests that real production versions will in fact be no worse than hand-constructed versions -- and will usually be better.

The lesson from all this, I hope, is that software is emerging from the state of being a black art into one in which it is a sound engineering discipline. Like all engineering, it is based on both hard headed cost-benefit tradeoffs and scientific principles. Evidence for the emerging state of software technology can be found in many places; PQCC is just one. Nonetheless, I hope it has served as an example that points out some of the important directions that I believe the discipline will take.

References

[1] Aho, A., Ullman, J.
Principles of Compiler Design.
Addison-Wesley Publishing Co., 1977.

[2] Aho, A.
Translator Writing Systems: Where Do They Now Stand.
Computer 13(8), August, 1980.

[3] Cattell, R.
Formalization and Automatic Generation of Code Generators.
PhD thesis, Carnegie-Mellon University, 1978.

[4] Erman, L., ans Lesser, V.
The Hearsay-II Speech Understanding System: A Tutorial.
In Lea, W. (editor), *Trends in Speech Recognition*, . Prentice-Hall Publishing Co., 1978.

[5] Forgy, C.
On the Efficient Implementation of Production Systems.
PhD thesis, Carnegie-Mellon University, 1979.

[6] Graham, S.
Table-Driven Code Generation.
Computer 13(8), August, 1980.

[7] Johnson, S.
Language Development Systems on the Unix System.
Computer 13(8), August, 1980.

[8] Leverett B., Cattell, R., Hobbs S., Newcomer, J., Reiner, A., Schatz, B., Wulf, W.
An Overview of the Production-Quality Compiler-Compiler Project.
Computer 13(8), August, 1980.

[9] Leverett, B.
Machine Independent Register Allocation in Optimizing Compilers.
PhD thesis, Carnegie-Mellon University, 1980.
(title is tentative, to be published Fall 1980).