CMU-CS-82-138

Delayed Binding in PQCC Generated Compilers

Wm. A. Wulf and K. V. Nori

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213, USA

4 October 1982

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Table of Contents

List of Figures

List of Tables

Abstract

We discuss issues concerning code generation in the context of the Production Quality Compiler Compiler (PQCC) project. Several important activities in code generation are cyclically interdependent. The task of structuring an efficient production quality code generator is therefore a challenging one. This paper reports on some uses of the principle of *delayed binding* in target program synthesis by PQCC generated compilers. Through the use of this principle, we have designed a sequence of phases that efficiently implement an optimizing code generator. Specifically, this paper is concerned with issues in the discovery of assignment sequences, the identification of temporaries and the utilization of implicit computations in target machines. As these phases are considered in the light of a Compiler-Compiler, we describe their parameterization with respect to source languages and target machines.

1. Introduction

The Production Quality Compiler Compiler (PQCC) project [26, 17] was an experiment in building a collection of *generators*, tools that automate all aspects of compiler construction. Compilation is comprised of a large variety of activities. They are explicitly recognized in the design of PQCC generated compilers (PQCs). For each such activity, an *expert* phase was designed, parameterized by relevant aspects of source languages and target machines (referred to as *relativization* throughout this paper). The process of relativization is automated through corresponding generators. In the broad picture, PQCC differs from other related efforts:

1. Target program synthesis is not an *ad hoc* extension of a formalized view of source program analysis. This was the case in all the early syntax-directed compiler-compilers [7, 13, 14] and, more recently, is true of systems based on formalisms for the definition of semantics of programming languages [9, 22, 16].

2. Retargetability and portability of compilers is often achieved by separating the code generation algorithm from the target machine data that drive it [10, 8, 15, 4]. The task of this code generation algorithm is usually to effect instruction selection (and register allocation in some cases). In PQCC, the correspondance between semantic primitives in the internal program representation and target machine operations is derived from a description of the target machine [4]. Others build this correspondence by enforcing conventions in the choice of the internal representation and target machine specification.

3. There is a choice to be made between the uniform use of general solution processes for solving special cases of problems and the construction of separate but specialized (and usually cheaper) procedures for tackling individual problems. In PQCC, the latter approach is used in designing the phases of PQCs, the expert system approach. In contrast, the ECS project [2] uniformly uses global data flow analysis techniques in implementing various program optimizations and systematically applies them at each stage of the compilation process.

. A substantial portion of the PQCC research effort was in the discovery and design of the phase structure of the generated PQC [20]. The BLISS-11 compiler [25] provided an initial approximation to the desired structure of a PQC. As can be appreciated, there was a strong interplay between the delineation of the boundaries of the phases and the design of specialized generators. The design of the generators themselves took up the rest of the research effort [18, 23, 27, 4, 21]. Figure 1-1 gives the gross structure of PQCC. As is conventional, the PQC can be broadly divided into a Front-End (FE) and a Back-End (BE). The FE analyses the source program and converts it into an internal representation (IR) [3, 11] depicting the abstract syntax tree. Effecting the translation and synthesizing an efficient object program is the task of the BE.

There are over 50 phases in our BE, some of which are the subject matter of this paper. Each of them accepts as input the IR generated by its predecessor and outputs another IR. The processing

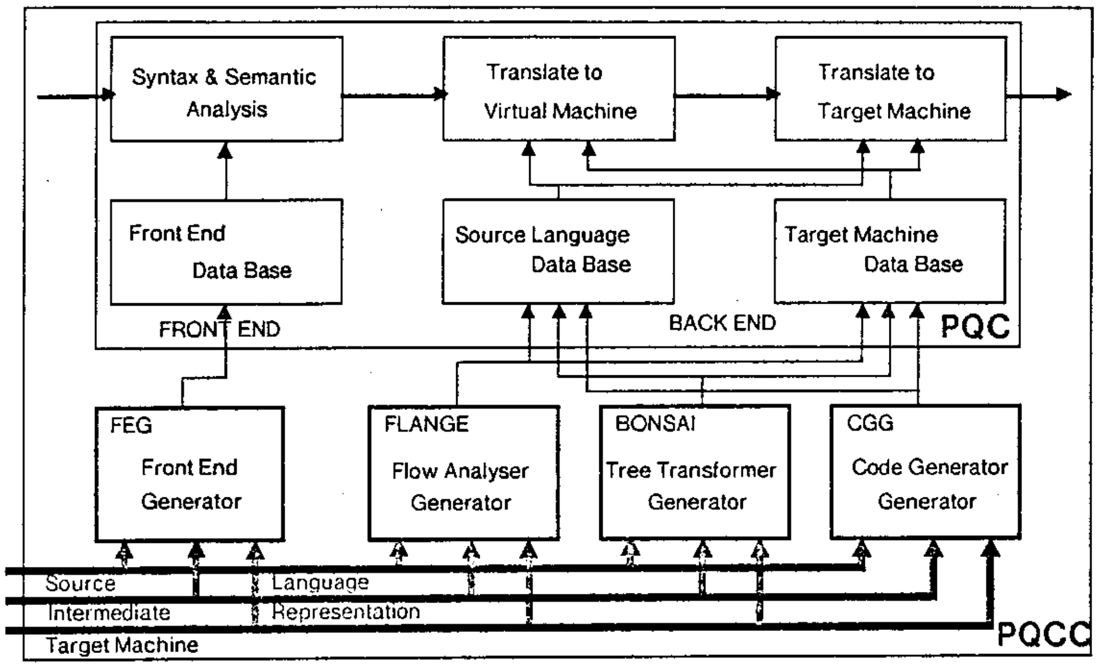

Figure 1-1: Overview of PQCC

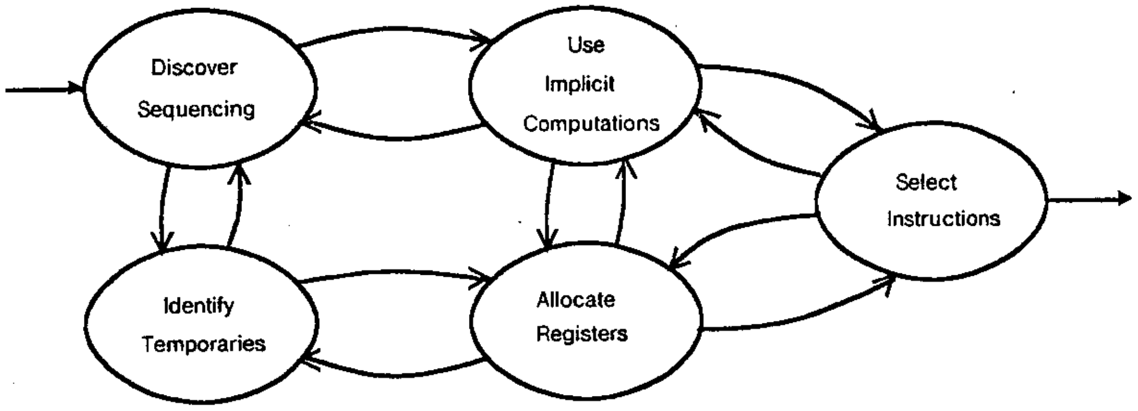

Figure 1-2: Interdependencies in Back-End Activities

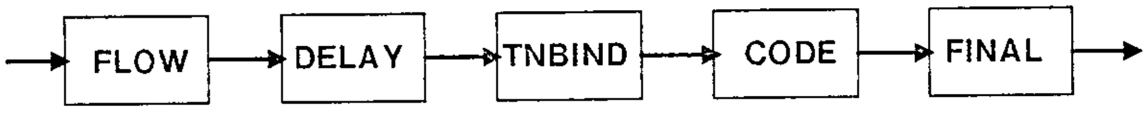

Figure 1-3: The PQCC Back-End

performed by the phase is reflected in one of several ways:

- it is a transformed version of the input, reflecting an optimization[1];
- it is an annotated version of the input, reflecting the results of the analyses performed by the phase;
- it is a translated version of the input, resulting in expansion of the input to reflect the introduction of implementation detail through *change* of operator and leaf nodes to represent the translation from one language to another.

Building interfaces between the BE phases that handle enrichment and changes in the IR can be systematized [19].

To get a conceptual hold on the problem tackled by the BE, it can be viewed as a sequence of two translation steps.

- The first stage of the translation process casts the source program in the mould of a Virtual Machine (VM) designed by the compiler writer. Herein are implemented certain global resource allocation policies concerning the representation of procedure linkage mechanisms, parameter passing conventions, storage layout design for static and dynamic user defined data structures and interface to runtime systems. In the present version of the system, the relativization of these phases with repect to target machines is done by hand. The automated generation of these phases is a topic of continuing research.
- Given the VM program in IR, the second stage completes the translation process by synthesizing an equivalent target machine program. This stage is concerned with local issues in programs, such as the basic blocks and the flow graph that describe the body of a procedure. The bulk of computing done in a program is in its basic blocks. We expend much effort in effecting a careful and efficient translation of these parts of source programs.

In this paper, we are largely concerned with the latter stage of the translation process, depicted in Figure 1-2. The activities performed in this part of the BE are:

1. **Identify Temporaries**: Anonymous operands in source program expressions have to be identified so that they may be allocated storage locations or registers in the target machine.
2. **Sequence Operations**: Expression evaluation has to be sequenced such that few temporaries are needed, registers are efficiently used, and the instruction set is well utilized.
3. **Use Implicit Computations**: Effective address generation mechansims and instructions

[1]in the spirit of source-to-source transformations.

that represent expressions involving several operators are utilized well in the translated program.

4. **Allocate Registers**: Utilize registers effectively for storing temporaries and performing operand accesses.

5. **Instruction Selection**: Several target machine instructions may realize the same source language function. Choice of proper instructions so as to efficiently perform the task described by the source program is crucial for generating quality code.

As shown in the figure, these activities are heavily *interdependent*. Optimal translation of some of these aspects is known to be computationally expensive. In PQCC, we have designed a *linear* sequence of phases that performs the above activities efficiently and produces good quality code. We adopt *delayed binding* as the general philosophy of breaking cycles of interdependence and designing a sequence that approximates the original [25]. Application of the principle of delayed binding has been often advocated in the program development process. Choice of data representation and the introduction of variables in programs are typically delayed till an appreciation of the operations involving them is obtained. Then, with hindsight, efficient programs can be constructed. As this part of the BE is concerned with the issue of target program synthesis, the utility of this principle should not come as a surprise. This philosophy is reflected in the nature of these BE phases:

- Some activities are performed twice. The first time, a set of feasible solutions are generated. At this stage of compilation, perhaps not enough is known for choosing the best solution. However, the set of feasible solutions may provide adequate information for performing a part of another activity. The second time around, the best possible solution is chosen with respect to the prevailing circumstances at that stage of compilation.

- Making conservative assumptions about a succeeding activity may resolve the cyclic dependancy and yet allow near-optimal code to be generated.

- Appreciation of a good approximation of the effect of a succeeding cyclically dependent phase may allow optimistic assumptions to be made to resolve it. The bindings performed by this phase may then be made with the confidence that pessimization that may result in the succeeding phase is marginal and locally contained.

With this background, a broad view of this part of BE is given in figure 1-3 as a sequence of groups of phases, each of which, in turn, being a phase sequence in itself. A brief description of each group follows:

- FLOW: Detect basic blocks and construct flow graphs that comprise procedure bodies in the source program. Global data flow analysis is performed, common subexpressions are detected, strength reduction and code motion are performed, and assertions are propagated to strengthen the effectiveness of future phases.

- DELAY: The main subject of this paper, these phases delay the identification of temporaries, discover an efficient assignment sequence to effect expression evaluation, and sequence operand evaluation so as to minimize register requirements. To perform these tasks, we need to discover feasible uses of the effective address generation mechanisms of the target machine and make good use of composite operations in its opcodes. An important strategy here is to algebraically restructure source expressions with a view to simplify, e.g., folding of constant expressions, or with a view to utilize features of the target machine, e.g., multiple arithmetic units, general registers, implicit address alignments during operand access, etc..

- TNBIND: Identify temporaries, analyse their lifetimes, and efficiently pack them into registers and storage locations of the target machines.

- CODE: Choose the effective address generation mechanisms to be used for performing individual operand accesses, order the basic blocks to avoid explicit jumps in the representation of the flow graph, select instructions to effectively utilize the instruction set, and handle register spilling and short term register allocation.

- FINAL: Perform peephole optimization.

Expectedly, the phase groups from DELAY onwards are increasingly target machine dependent. In constructing a PQC, the programming of the algorithms of the various phases in BE is a one time effort. The parameters that particularize these algorithms with respect to a source language and a target machine are obtained from respective data bases output by the PQCC generators. The data bases are produced afresh for every generated PQC.

Now for an overview of the rest of the paper. Section 2 gives the details of construction of ALGEBRA, a phase that performs algebraic IR transformations. This phase is automatically constructed by BONSAI, a PQCC generator that is also described. Section 3 is concerned with the discovery of efficient assignment sequences equivalent to source expressions. To perform this task, we consider (a) the desirability of computing results in user defined variables, (b) the target machine dependent possibility of combining some unary operations with binary operations, and (c) the feasible uses of effective address generation mechanisms. The basic reason for undertaking these activities is to avoid making demands for temporaries. This is achieved by overloading user variables, overloading unavoidable temporaries, and associating with each relevant source expression a set of feasible effective address generation mechanisms that either compute the desired value or perform the necessary operand access (thereby obviating the temporary need for a target machine resource to store either the value or an address). Disscussion of efficient sequencing of operand evaluation brings us to the end of this section. Finally, in Section 4, we review the presented material and point out possible directions for further work.

2. Simplifying Algebraic Transformations

In this section we discuss ALGEBRA, an early phase of DELAY, and its generator BONSAI [27]. ALGEBRA transforms source language expressions represented in IR. The transformations use the commutativity, associativity and distributivity properties of source language operators. Some of the transformations are specialized to common sources of optimization that arise in subscript expressions and addressing array elements. ALGEBRA is not a generalized algebraic simplification system; rather, it is a simple-minded simplifier that works torwards efficient expression evaluation on target machines. Techniques predominant in this phase are:

- canonical representation of expressions so as to detect and evaluate constant subexpressions;
- restructuring the representation using algebraic properties of operators so as to reduce demands on temporary storage locations;
- expanding subscript expressions which can give rise to constant subexpressions;
- restructuring expressions (that explicitly reflect operand alignment) so as to use effective address generation mechanisms in target machines.

Of the above transformations, three are indirectly target machine dependent and need appropriate relativization:

- the compile time evaluation of constant expressions requires knowledge of the representation of constants in the target machine and good algorithms that respect the properties of the representation when performing the evaluation;
- the properties of IR presumed by later phases in locating operand accesses and their translation to effective address generation mechanisms of the target machines;
- the set of algebraic transformations to be applied to the IR so as to reduce demands on use of target machine resources depends on the characteristics of the target machine, such as the availability of multiple arithmetic units, etc.

The above issues are comparatively simple. The first problem does require a suitable parameterization of the target machine. The second is simply resolved by using uniform conventions in the coding of the PQC phases. The last is tackled by generating different ALGEBRA phases for different target machines, thereby providing one of the motivations for designing and implementing BONSAI.

As can be seen, the scope of the projected transformations is limited and an efficient realization is feasible.

In the BE, source language relativization is largely obtained through the richness of the IR. As ALGEBRA is designed with respect to the IR, rather than a specific programming language, much of the phase can be common to many generated PQCs; this is so because there are many operators common to the programming languages handled by PQCC. Even so, the generation of ALGEBRA for the DELAY part of a specific PQC is eased by BONSAI, a tree transformer generator.

We will first look at BONSAI and then at ALGEBRA.

2.1. BONSAI: A Tree Transformer Generator

BONSAI is a simple language for describing certain kinds of tree transformations, expressed as ⟨*pattern, action*⟩ pairs. The action in each such pair is applied to a tree node if it satisfies the pattern. The described transformations are those internal to phases of PQCs. Given the narrow application domain, and a need for efficient implementation, BONSAI has several low-level aspects, the most restrictive of which is the tight coupling between the language constructs of BONSAI and the implementation language for PQCs, currently BLISS [24].

A BONSAI program is a specification of a phase of a PQC and it consists of the following sequence of definitions.

- phase name.
- sequence of routines in the implementation language which may be invoked during pattern matching or tree transformations defined below.
- sequence of productions defined in the ⟨*pattern, action*⟩ sublanguage.
- specification of the driver that performs the overall control of the pattern matching and tree transformation process.

Of these, only the last two are important to the discussion. The driver is specified by a string over the alphabet {L,R,N}, respectively indicating traversals of left or right subtrees and a visit to the node. For example, preorder, inorder and postorder traversals are given by the strings NLR, LNR and LRN respectively. Multiple occurrences of L or R in the string denote multiple traversals of the corresponding subtree, and multiple occurrences of N denote multiple visits to the tree node; the interpretation of the control string is from left to right. From this simple specification, an appropriate driver routine that controls the phase is generated. The main part of a BONSAI program is expressed in the ⟨*pattern, action*⟩ sublanguage elaborated below.

A pattern is either built-in or finitely constructed by the user. Some built-in patterns are **any**, matching any tree node, **anyseq**, matching any sequence of tree nodes, **k**, matching any constant leaf node, and **nk**, matching any non-constant node. Constructed patterns are n-tuples (providing a prefix representation of the root node of the pattern); elements of the n-tuple may be labelled $*\$i*$, where i>0, and may be n-tuples themselves, all labels occurring in a pattern being unique. The first element of the n-tuple must be either one of the built-in patterns or a specific operator. Patterns may optionally have predicates associated with them so as to express not only structure but also properties of labelled nodes in the pattern. Arbitrarily complex predicates can be specified by passing labelled nodes as parameters to routines in the implementation language of PQCs. The syntax of BONSAI was designed to easily relate to the tree structure of IR. For the sake of simplicity of illustration in this paper, we will use the normal infix notation to describe both patterns and the transformed expressions. For example, the pattern

$1:($2:**k** + $3:**any**)

describes an expression, tagged as $1 for later reference, which is a sum of a constant, the addend, tagged as $2, and any other operand, the augend, tagged as $3. Another example of a constructed pattern is

$1:($2:**k** + $3:($4:**k** + $5:**nk**)) | **not** *cse*($3)

a sum of a constant and an expression, which in turn is a sum of a constant and a non-constant; additionally, the latter expression is not a common subexpression. The predicate following the bar (|) is assumed to be a boolean expression in the implementation language of PQC, and hence it is presumed that *cse* is a user supplied boolean function of a single (tree) parameter, available to the BONSAI program through a preceeding declaration.

Transforming actions are ways of restructuring elements that match the labelled portions of the pattern. Built in transforming actions allow for node replacement, construction of nodes, and construction of constant and non-constant leaf nodes. Invocation of PQC implementation language routines in actual parameter positions of these built-in transforming actions is allowed. An example of a statement in the *<pattern action>* language is

$2:**k** + $3:($4:**k** + $5:**nk**) | **not** *cse*($3) => $5 + *eval*($2 + $4)

where the constructed pattern is taken from the above example; the transforming action invokes a user defined PQC implementation language routine called *eval* which is assumed to effect folding of constant expressions, and is accessible through the declarations in the BONSAI program.

The BONSAI processor converts a BONSAI program into a PQC implementation language program. The basic nature of this translation is to generate a routine for every production, and a controlling driver program. On visiting a node, the controlling driver program invokes each production generated routine, in the order of their declaration.

2.2. ALGEBRA: A Simple Algebraic Simplifier

In PQCC, exploitation of target machine features was the main motivation for choosing amongst the large body of feasible compile-time optimizations. As a result, the kinds of algebraic simplifications attempted in the generated PQCs are limited to those that aid in, or reduce, the work involved in generating good quality code; semantics preserving source-to-source optimizations are presumed to have already been effected. The nature of tree transformations that are attempted in the phase ALGEBRA contribute to good use of target machine resources.

There are two groups of transformations in the present version of this phase, designed to be a part of the BE of an ADA subset compiler to produce code for the VAX-11 system [5]. Each of these groups is discussed below, both with respect to the transformations performed and the ways in which they contribute to good code generation. The transformations are presented in the order they are applied by ALGEBRA. To simplify the description of the transformations, the following conventions are used:

- *ak* denotes any constant operand;
- *nak* stipulates that the operand is not a constant;
- *leftleaf* stipulates that the left operand is a leaf of the expression tree and is either a literal, or a simple variable;
- *lc* denotes a load-time constant, e.g., a relocatable address;
- *cse* stipulates that the operand is a common subexpression;
- *ln* denotes a literal or a name;
- *isaddr* checks that the expression is to evaluate to an address;
- *eval* denotes an operation that performs compile-time folding of constant expressions and occurs in transforming actions of productions.

The above conventions are used to describe the nature of operands in patterns either *in situ* or as predicates in conditions associated with patterns.

2.2.1. Basic Transformation to Canonical Representations

In the present implementation, ALGEBRA is limited to transforming arithmetic expressions, the largest class of expressions commonly found in programs. Generalising to other classes of expressions should prove to be a straightforward extension of the ideas used in handling arithmetic expressions.

There are many possible canonical representations of expressions. The representation that we have chosen is guided by the desire to simply perform folding of constant expressions. For our purposes, it is enough if simple operands of infix operators, such as constants or simple variables, are their right operands. For commutative operators, this transformation is trivial. For others, simple algebraic equivalences suffice to provide the basis for the transformations.

The first group of transformations in this catagory deal with the simplest cases, viz., the left operand of an infix operator is either a constant or a simple variable.

- ($2:**any** + $3:**any**) | *leftleaf*($2) => ($3 + $2)
  ($2:*ak* + $3:*nak*) => ($3 + $2)
  Commutativity of addition is used to transform simple expressions to canonical form by applying the above two rules. Due to this transformation, simple operands are referenced last; so the temporary storage demand that they may create will have the shortest possible lifetime. Hence, even this simple transformation can lead to good code.

- ($2:*k* - $3:*nk*) => (- $3 + $2)
  ($2:*nk* - $3:*k*) => ($2 + *eval*(- $3))
  These two simple rules convert subtraction to addition for the purposes of canonical representaion. The latter rule causes the negation of the constant operand to be performed at compile time so that the possibility of using indexing to represent addition in target machine is created.

- ($2:*k* * $3:*nk*) => ($3 * $2)
  Commutativity of multiplication is used for obtaining canonical representation.

The next group of transformations concern address arithmetic expressions. Complex address computations, representing component accesses of statically allocated objects, usually involve arithmetic with load-time constants that represent relocatable addresses. In some cases, the relocatable address is known at compile time; adding constant offsets to such addresses can then be performed at compile time. Else the relocatable address is known only at load time; in such cases, adding constant offsets has to be done by the loader too. To take care of the above two cases, either at compile time or by providing requisite information to the loader, the load-constant and the constant offset must be paired together with an appropriate operator. The following two rules provide the required transformations.

$2:*lc* + $3:($4:*nk* + $5:k) | not *cse*($3) => $4 + ($2 + $5)
$2:*lc* + $3:($4:*nk* − $5:k) | not *cse*($3) => $4 + ($2 − $5)

The above sort of expressions typically result from references to fields of component records of global arrays, e.g., *a[e].f* where *a* is a global array whose base is a relocatable address, and hence a load-time constant, *e* is any valid non-constant array-element-access computation, and *f* is a constant offset required to access a field.

For multiple general register machines, such as the VAX-11, expressions of the form $((a + b) + c) + d$ are preferable to $(a + b) + (c + d)$ for efficient left to right evaluation. As can be seen, the lifetimes of the intermediate values to be preserved during the computation of the expressions are short. This leads to resuability of temporary store and good utilization of target machine registers. Also, should *b* and/or *d* be constants, canonical representation transformations are in order. ALGEBRA has a class of transformations of the form

$2:($4:*nk* + $5:k) + $3:($6:*nk* + $7:k) | not *cse*($2) and not *cse*($3)
=> ($4 + $6) + *eval*($5 + $7)

Table 2-1 summarizes all the transformations in the above class. The standard Decision Table format is used to describe the relevant cases. In the implementation of this class of transformations, all possible combinations of operators and conditions satisfied by the operands are explicitly enumerated.[2] Similar transformations of the form

$2:($4:*nk* + $5:k) + $3:*ln* | not *cse*($2) => $4 + *eval*($3 + $5)

are generically described by the rule

$2:($4:*nk* O_2 $5:k) O_1 $3:*ln* | not *cse*($2) => $4 O_1 *eval*($3 $O'_2$$5)
where O_1 and O_2 are as in Table 2-1.
and $O'_2 = f(O_1, O_2)$

This completes the description of transformations to canonical representations.

2.2.2. Simplifying Subscript Expressions

The second group of transformations in ALGEBRA is specifically concerned with subscript expressions, a common source of simple optimizations. Consider an array *a* declared as

var *a*: **array** [*i..m;k..n*] **of** *T*

The reference *a[x,y]* will usually be represented by the expression

[2] In retrospect, improved pattern construction facilities in BONSAI can lead to simpler implementations of such groups of transformations.

	Rule 1	Rule 2	Rule 3	Rule 4	else rule
$k(T_5)$	T	T	T	F	
$k(T_7)$	T	F	–	–	
$cse(T_2)$	F	F	F	–	
$cse(T_3)$	F	F	T	F	
$O_3' = O_3$			X		X
$O_3' = f(O_1,O_3)$		X		X	
$O_3' = f(O_2,f(O_1,O_3))$	X				
$(T_4\ O_2\ T_5)\ O_1\ (T_6\ O_3'T_7)$					X
$(T_4\ O_1\ T_6)\ O_2\ eval(T_5\ O_3'T_7)$	X				
$(T_4\ O_1\ (T_6\ O_3'T_7))\ O_2\ T_5$			X		
$((T_4\ O_1\ T_6)\ O_3'T_7)\ O_2\ T_5$		X			
$((T_4\ O_2\ T_5)\ O_1\ T_6)\ O_3'T_7$				X	

where $T_1 = T_2\ O_1\ T_3$
$T_2 = T_4\ O_2\ T_5$
$T_3 = T_6\ O_3\ T_7$
and $O_1, O_2, O_3\ \varepsilon\ \{ + , - \}$

O_m	O_n	$f(O_m,O_n)$
+	+	+
+	–	–
–	+	–
–	–	+

Table 2-1: Transforms for $(T_4\ O_2\ T_5)\ O_1\ (T_6\ O_3\ T_7)$

$a + ((x + i)*j + y + k)*l$
where a is the base address of the array
x, y are arbitrary (row and column) expressions
i, k are the lower bounds of (row and column) index ranges
j is the number of rows, i.e., $m - i + 1$
l is the size of an element of type T in terms of target machine storage units.

By distributing multiplication over addition in the above expression, constant subexpressions $i*j$ and $k*l$ can be made candidates for folding. The relevant transformations, to be repeatedly applied to such expressions from the innermost nested subexpression outwards, are generically expressed as

$2:($4:any O_1 $5:**k**) * $3:**k** | not *cse*($2) => ($4 * $3) O_1 *eval*($5 * $3)
where $O_1 \varepsilon \{+, -\}$

A common problem in dealing with array elements is address alignment. Target machines do have some built in, preferred, data types for which efficient access mechanisms and manipulation are provided in hardware. For instance, the VAX-11 has 4 byte words. Effective address computations involving indexing causes the index values to be automatically scaled by 4. Recognizing these special cases can lead to fruitful uses of implicit address computations in target machine instructions. The transformation described below sets up the convention used by succeeding phases in searching for the presence of these special cases.

$1:($2:**any** + ($3:**any** * $4:**any**)) | *isaddrctx*($1) => ($4 * $5) + $2

In all the above transformations, it is assumed that the invocation of *eval* deals with the special case of a constant expression folding to zero.

3. Delaying the Identification of Temporaries

Anonymity of intermediate states of computation and simplifying conventions are two common principles in the design of higher level languages:

- Anonymity frees the user from identifying (naming) parts, and is usually achieved by recursive specification of language features, e.g., operands of operators in infix expressions are implicit in the notation and can be arbitrarily complex.

- A use of simplifying conventions is to obviate the need to explicity represent the nested structure of computation. For example, the use of operator precedence obviates the need for paranthesizing all operands, and hence leads to lexical simplicity as well as the avoidance of over specification of sequencing of operators.

Our interest in these principles is from the standpoint of basic blocks. For the higher level languages of interest to PQCC, the application of these principles gives rise to a rich applicative sublanguage of expressions and a composite imperative language of assignments and sequencing. On the surface, these two aspects seem orthogonal and are often treated as separate conceptual worlds, the world of values and the world of side effects.

Machine languages differ in degree with respect to the above concerns: the applicative sublanguage here is rudimentary and is fixed in structure, as is the imperative part. However, these two aspects are not separate orthogonal parts of such languages; every statement (instruction) in a machine language has a bit of both. The applicative sublanguage consists of a fixed class of subexpressions that arise in effective address computations for identification of operands, and a fixed class of simple, typed expressions, both of whose evaluation is internal to the machine. The imperative part is just simple assignments. All operands of the applicative and imperative parts of a statement are explicitly identified. We refer to the applicative sublanguage of machine languages as *implicit computations* in machine instructions. An example will illustrate the above delineations. Consider the PDP-11 [6] instruction

```
BIC X(R1), R0.
```

Expressed in the form of imperative assignments in BLISS, the equivalent statements would be[3]

R0
= (NOT(.(X + .R1))) AND .R0 ; PC = .PC + 4

where *NOT* and *AND* boolean operations on 16 bit words. Hidden in this instruction is the applicative expression *.(X+.R1)*, and the expression of the form *(NOT(A)) AND B*. They are the implicit computations performed in the execution of this instruction. The former implicit computation is typical of contexts requiring operand access, and the latter is typical of contexts where unary operations are combined with appropriate binary operations in contexts requiring values or denoting assignments. The imperative aspects of the above instruction are the two assignments. The first assignment is explicit, and the second assignment (to the program counter) is implicit in the target machine.

Due to the simple and fixed nature of the applicative and imperative aspects of machine languages, translation from higher level languages to machine languages is therefore seen to necessitate the loss of anonymity of operands in source language expressions. Compiler generated *temporary* names are used to denote these anonymous operands; the loss of anonymity is complete with the introduction of explicit assignments with these temporary names as their destinations. Optimized translation requires the minimization of the number of temporary names generated, a process that interacts with the choice in sequencing the newly introduced assignments.

In this section, we describe the techniques we have employed to postpone the identification of

---

[3]The unary *contents of* operator is represented by the period (.) in BLISS, and denotes an access to the value of its operand.

anonymous entities till it becomes clear that their identification cannot be avoided. Issues in sequencing influence the design of the techniques described below. However, the final determination of an optimal sequence is held off till the end of the application of these techniques.

3.1. Overloading and Target Path Determination

A common programming practice in languages with restricted scoping facilitites is to use the same variable for different purposes in temporaly disjoint parts of a procedure. For instance, a single loop counter variable may be used to control several non-nested loops in languages like FORTRAN and Pascal. The same effect could have been achieved by using separate control variables for each of the loops. However, noting that the lifetimes of non-nested loops are disjoint permits the use of a single variable to control all of them. The variable has different semantics at different points in the procedure, viz., it has overloaded semantics. This complication of the semantics of the variable is offset by the storage optimization that results.

We explore the possibility of applying the above principle to minimize the number of temporary names required to identify anonymous operands of operators in basic blocks. Specifically, we look for cases where it is feasible to overload the semantics of

- user declared variables with some anonymous operands
- some anonymous operands with other anonymous operands.

To perform the above analysis, however, requires a preliminary identification of anonymous operands. In traditional IRs, such as quadruples, a unique temporary name is associated with the result of every applicative expression. This view is simplistic in that it ignores the structure of implicit computations in target machines and that good code generation entails their effective use. As we are interested in generating production quality code, our discussion on overloading the semantics of some temporary names has to be mixed with the issue of discovery of feasible implicit computations. We refer to the analysis required to effect a good overloading policy as *target path determination*.

With respect to the two cases of overloading itemized above, the following brief observations are in order:

- for user declared variables that are assigned a computed value in a basic block, there is a hole in the lifetime of that variable between its last use in the basic block and the point it is assigned its new value -- during this period, it may prove beneficial to overload this target of the assignment with anonymous operands in the computation of the value to be assigned to the target. We refer to this aspect of target path determination as *computation in destination*.

- for anonymous operands that are not common subexpressions, their only use in the computation of a new (perhaps anonymous) value is their last use. Associating a temporary name with such an operand amounts to assigning the value of the operand to the temporary name, and leads to the situation typified by the following example: the expression $a*b + c*d$ could lead to the sequence

$$T_1{:}= a*b;\ T_2{:}= b*c;\ T_3{:}= T_1 + T_2$$

where T_1, T_2 and T_3 are temporary names, and either the semantics of T_1 could be overloaded with the semantics of T_3 or the semantics of T_2 could be overloaded with that of T_3. This choice is target machine dependent, as will be explained in a section below. As it turns out, the decision is based on the nature of unary operations available in the target machine, and the associated costs when these operations are combined with the assignment operation in target machine instructions. Making good use of such combinations is a part the discovery of use for implicit computations in target machine instructions. We refer to this class of optimizations as *unary-complement optimizations* [25]. Effecting this choice completes target path determination in basic blocks.

A detailed discussion of the issues in computing results in the destinations of assignments and in unary-complement optimizations follows.

3.2. Computation in Destinations

For the purposes of introduction, the above discussion of destinations of assignments only mentioned user defined variables. Actually, aside from user declared variables, an important class of destinations is the set of implicit variables required by the semantics of higher level languages. Examples of such implicit variables are locations that contain value parameters, bounds of **for** loops, computed addresses of **with** variables in Pascal, computed expression values passed by reference in FORTRAN, etc.. In each of the above cases, the semantics of the source language necessitates an assignment to the implicit variable. Also, in each case, there is no prior use of the implicit variable, and hence its semantics can be overloaded with that of some temporary names right up to the point where it is assigned the value it should denote. In such cases, as well as in the case of assignments to user declared variables in the basic block under consideration, we take the conservative view that these locations are used, if benefical, only to store anonymous operands of the expressions they finally denote. Also, we ignore the possibility of using user variables to store common subexpressions, as illustrated by the following example:

$$a{:}= b*c;\ b{:}= b*c + a.$$

In this case, as $b*c$ is a common subexpression, neither a nor b are considered as candidates for overloading with the anonymous operand $b*c$. We believe that the conservative view we have taken only affects somewhat unnatural or pathological cases and contributes to the simplicity of analysis required by the task.

The phase in DELAY that determines the legality and desirability of performing computation in destinations in respective passes is called CDEST. As can be expected, legality is a target machine independent issue. Not so obvious is the fact that computation in destinations may not be always desirable. This value judgement is target machine dependent. Hence the need to relativize CDEST with respect to target machines. The effect of this phase is to annotate every expression tree node of the IR. The annotation states whether it is desirable to store the value of the computed expression in a program specified destination, and if this is indeed the case, then it points to appropriate destination.

It is legal to compute a value in a destination if and only if the current value in it is not needed at a later point in the evaluation of the expression under consideration. To determine legality of computation in destinations, the first pass of CDEST locates basic blocks in the IR, and then locates the assignment nodes in the basic blocks. Each of the assignment nodes identifies as its left operand (subtree) a possible candidate destination. A LRN traversal of the right operand is performed to determine the legality of using the candidate destination for storing the value of the computed subexpression represented by each node visited during the traversal. Information about the candidate destination is passed from the root of the expression tree in the IR to its leaves. The legality issue for each node, and the question of which operand of the operator at the current node should be computed in the destination, are both determined bottom-up.

The factors that determine legality are several and concern both the candidate destination and the node being processed:

- do the candidate destination and the expression represented by the node overlap?
- Is the expression represented by the node a common subexpression?
- Will the evaluation of the expression in the candidate destroy its original contents?

Combinations of these factors represent the legality status of a node with respect to the candidate destination; they are encoded as *states* and associated with the nodes as a result of the processing. The state of the current node is determined in terms of its descendants, whose states would have already been determined due to the nature of the LRN traversal. These states are:

NIND	not involved and not destroyed
NID	not involved and destroyed
IND	involved and not destroyed
ID	involved and destroyed
CNI	common subexpression not involving the candidate destination
CI	common subexpression involving the candidate destination.

A candidate is not destroyed if no new value is stored in it, e.g., leaves in the tree that represent user variables are just addresses in the target machine store and the unary *contents of* operation has to be performed before any action takes place -- so such leaves by themselves do not destroy the candidate destination.

To illustrate the process of setting the state of the current node in terms of the states of its descendants, we run through a small example. Consider the VM expression *(.a + .b)*(.c − .d)*, assuming the destination candidate to be *a*.

- the node leading to the leaf *a* would have the state IND;
- the nodes leading to the leaves *b, c, d* would have the state NIND;
- the node for the subexpression *.a* would have the state ID;
- the nodes for the subexpressions *.b, .c, .d* would have the state NID;
- the node for the subexpression *.a + .b* would have the state ID;
- the node for the subexpression *.c − .d* would have the state NID;
- and the node for the entire expression would have the state ID.

Knowing the state associated with the descendant nodes can further help in deciding which of the descendants should be computed in the candidate destination.

Two tables are constructed, Table 3-1(a) for binary operators and Table 3-1(b) for unary operators, for determining the state for the current node in terms of the state of its descendants and for determining which descendant should be computed in the candidate destination. The possibilities for the latter are:

R	the right operand should be computed in the destination
L	the left operand should be computed in the destination
B	the current node can be computed in the destination, but it is not legal to compute its descendants in the destination, i.e., the candidate is *blocked* as a destination for the descendants.
*	for the purposes of legality, it does not matter which descendant is computed in the destination.

To use Table 3-1(a), we select the row according to the state of the left operand and the column according to the state of the right operand. The entry in the table gives the state to be associated with the currently visited node and also determines the descendant that should be computed in the candidate destination. Table 3-1(b) has only one column; the row is selected according to the state of the only descendant of the current node.

	NIND	NID	IND	ID	CNI	CI
NIND	NID, *	NID, *	ID, R	ID, R	NID, L	ID, B
NID	NID, *	NID, *	ID, R	ID, R	NID, L	ID, B
IND	ID, L	ID, L	ID, L	ID, B	ID, L	ID, B
ID	ID, L	ID, L	ID, B	ID, B	ID, L	ID, B
CNI	NID, R	NID, R	ID, R	ID, R	NID, B	ID, B
CI	ID, B	ID, B	ID, B	ID, B	ID, B	ID, B

(a) Table for binary operators

descendant state	node state
NIND	NID
NID	NID
IND	ID
ID	ID
CNI	NID, B
CI	ID, B

(b) Table for unary operators

Table 3-1: Tables for node state determination

On PDP-11	On DEC System 10	On DEC System 10	On DECsystem 10
	a,b,c in registers	*a,b,c* in store	avoid destination
MOV B,A	MOVE A,B	MOVE R,B	MOVE R,B
ADD C,A	ADD A,C	MOVEM R,A	ADD R,C
		MOVE R,C	MOVEM R,A
		ADDM R,A	

Table 3-2: Code sequences for *a: = b + c*

At the end of the first pass, the relevant expression trees are annotated by state information and the selected descendant. Now that legality of computation in destination is known, we need to analyse the desirability of actually performing the evaluation in the destination. The reason why this analysis is target machine dependent is that we know very little about the target machine resources that will be assigned to represent these temporary names, i.e., we do not know whether these temporary names will be bound to registers or locations in the target machine store. Also, we have not made any assumptions about the target machine resource that is assigned to represent the destinations. All these details are needed to compare the various code sequences that could be generated for the assignment under consideration. Table 3-2 gives an example to illustrate the point. The last column shows conditions where it is undesirable to compute in the candidate destination, even though it is legal to do so.

To undertake the desirability analysis, the second pass has a table of target machine costs of register-to-register, memory-to-register, register-to-memory, and memory-to-memory forms of each node operator. Temporary names required for operands that are not to be computed in the candidate destination are assumed to be available in registers. Reasonable assumptions, influenced by the representations effected by the translation to VM, about the target machine resources assigned to the candidate destination, and other variables that are operands of the expression under consideration, are made in order to cost the sequence that will be generated if the destination is used (CYES) or otherwise (CNO). For example, we assume that arrays and global or external variables are not available in registers, and that simple local variables are bound to registers. If the former assumption proves to be false, better code will be generated by the succeeding phase collections. And if the latter assumption proves to be false, later phase collections will ignore the recommendation being made here.

The second pass is a Nx traversal to the expression tree, where x stands for the chosen descedant in the first pass. That is, the first action is to visit the node and evaluate CYES and CNO for the node. If CYES is greater than CNO, and x does not indicate a blocking of the candidate destination at the current node, traversal continues by taking up the visit to the node specified by x. So long as CYES is greater than CNO at the visited node, it is annotated by a pointer that refers to the candidate destination.

The analysis of feasibility of overloading the semantics of the candidate destination, and the desirability of actually doing so in light of the characteristics of the target machine is now complete. No transformations have been effected due to this analysis. We have ignored the possibility of profitably using implicit computations in the target machine when conducting the above analysis.

The only effect of this simplification is that the semantics of the candidate destination may be overloaded with the semantics of temporaries more number of times than may be necessary, for good use of implicit computations can lead to the avoidance of some of these temporaries.

3.3. Problems in Identifying Implicit Computations

As noted earlier, there are two forms of implicit computations:

- those representing effective address computations in target machines. Precise identification of opportunities for effective use of such implicit computations would require knowledge of *register allocation*. However, the task of the DELAY phases is to delay the identification of temporaries, and hence cannot presume register allocation. At this stage of the compilation process therefore, only feasible uses of such implicit computations can be identified, based on assumptions about the availability and contents of target machine registers.

- those representing combinations of unary operations with other appropriately typed binary operations, mainly arithmetic and logical operations. Precise identification of such combinations in basic blocks is again an algebraic simplification problem and does not require any other target machine information aside from the combinations inherent in its instruction set. Desirability of use of such target machine features depends on associated costs.

Representing instructions as operator trees [12] -- with the root (and some of its immediate descendants) as operator nodes, and fixed depth subtrees representing operand accesses -- can lead us to the simplistic conclusion that opportunities for effective address computations can be identified by traversing the IR of basic blocks bottom-up, and that combinations of unary and other operations can be discovered during their top-down traversal. It turns out that the computations concerning the above identification processes are expensive, and hence wasteful if performed separately for each of the above problems. Consequently, an attempt is made to segment basic blocks so that only one of the identification processes is applied to each node.

A LRN traversal of the source program tree is performed. If the visited node is expected to yield an address, then feasible uses of target machine effective address computation mechanisms are identified. We refer to this process as feasible *access mode determination*. In contexts where the node is expected to either produce a value, or affect control flow, or result in an assignment, identification of combination of unary operators with appropriate binary operators is performed, i.e., *unary complement optimizations* are applied. The effect of this identification process is to avoid temporaries for interior nodes of the identified subtrees as they will be subsumed within the implicit computations internal to the target machine hardware. Given the results of the feasibility analyses of the CDEST phase and access mode determination, unary complement optimization leads to the identification of *unavoidable* temporaries, a precondition that allows for the completion of target path determination.

The phase that effects unary complement optimizations is called UCOMP, and AMD is the phase that performs feasible access mode determination analysis. From the above description, it should be clear that neither of these phases is applied to entire basic blocks; rather, a controlling traversal program determines which of the phases should be applied to the visited node.

3.4. Targetting and Unary Complement Optimizations

On the surface, unary complement optimizations may conjure the image of a traversal of the IR with a view to locate certain unary operations in it, followed by a check of the neighbourhood of these operations for appropriate binary operations that will provide uses for implicit computations in the target machine. Such ready made cases are rare. Our task is create opportunities for using such implicit computations by algebraic transformations, provided that the transformation yields economy in terms of quality of generated code.

The following examples will serve to illustrate the flavour of the task on hand.

- We would like to compute $a - b$ instead of the expression $-(b - a)$ as the latter expression has an extra negation operation in it. This example does not necessarily lead to use of implicit computations in the instructions of the target machine, but achieves a simplification of the intended computation.
- The expression $(-a)*b - c$ is more desirable than $-(a*b + c)$ on most target machines because addition and subtraction usually have the same costs, and because negation of an accessed operand usually entails no cost.
- For some target machines, such as the PDP-11, the expression (**not** *a*) **and** *b* is preferable to **not**(*a* **or not** *b*) because of the nature of its instructions.

In each case, the expression is either simplified with respect to the unary complement operators in it, or the expression is massaged so that they are suitably postioned for taking advantage of the target machine characterisitcs.

For the purposes of discussion, we will limit ourselves to integer arithmetic. Extension of the analysis to other arithmetic types as well as logical connectives is straightforward. With respect to the nature of instruction sets of most target machines, the only unary operation of interest is complementation, viz., negation of integers in our case.

The crux of the matter is in the target machine cost of evaluation of an expression or that of its complement. This analysis is carried out for every arithmetic operator in the source program tree. As the analysis is performed LRN, we need to know a bit about the operands of the arithmetic operator being analysed, e.g., will they be in registers or in the store of the target machine, are they common subexpressions or user defined variables, do they match the intended destination of the value of the expression being computed. These observations will obviously affect the choice of the target path and the target machine instruction sequence that best implements the semantics of the analysed operator. Also of consequence are assumptions about the signs of the operands, e.g., the left operand is as expected but the complement of the right operand is available, etc.; a part of the analysis is devoted towards discovery of the best combination of such assumptions that results in optimal code. The outcomes of the analysis are a decision with regard to the target path in the evaluation of the considered expression, and transformations to the expression tree that incorporate the *correct* signs for operands that yields optimal target machine code.

The task of this phase is performed in two passes. The first pass determines the target path and the assumptions about operand signs that lead to optimal code. The second pass uses the latter information to drive the tree transformation process.

Tables 3-3(a) and 3-3(b) reflect the logic of the analyses performed in the first pass on visiting a node with the integer + operator. Similar tables can be used to describe the analysis performed for other operators. The significance of the entries in each column of the tables is:

1. The first column describes the temporariness of the left and right operands of the operator being considered. An operand is considered *destroyable* if it is expected to be in a register and its value is not needed after the current operation. User defined variables are not destroyable, neither are constants and common subexpressions. If there exists a candidate destination for this expression as identified by the phase CDEST, and the operand matches it, then this operand can be considered as destroyable. However, such an operand would not be normally located in a register, and hence will entail the penalty of load and store costs in addition to the costs identified in the last column described below. The destroyability information is useful for deciding the target path and in the computation of target machine dependent costs.

2. The second column describes the assumptions about the operands. If the operand is assumed to be available as expected, i.e., it is positive, this condition is signified by the entry "p". On the other hand, if the complement of the operand is assumed to be available, then the entry is "n".

3. The third column gives the target path. Given an expression $E_1 + E_2$ and that its sub expressions E_1 and E_2 are evaluated in temporary locations T_1 and T_2 respectively, we then say that the target path is LEFT if the generated code is equivalent to

$$T_1 := T_1 + T_2$$

Destroyability	Signs	Target Path	Operations
TT	pp	–	add
TT	pn	LEFT	sub
TT	np	RIGHT	sub
TT	nn	–	preneg; sub
TF	pp	LEFT	add
TF	pn	LEFT	sub
TF	np	LEFT	sub; postneg
TF	nn	LEFT	add; postneg
FT	pp	RIGHT	add
FT	pn	RIGHT	sub; postneg
FT	np	RIGHT	sub
FT	nn	RIGHT	add; postneg
FF	pp	–	load; add
FF	pn	LEFT	load; sub
FF	np	RIGHT	load; sub
FF	nn	–	negload; sub

(a) Evaluating the + Operator

Destroyability	Signs	Target Path	Operations
TT	pp	–	add; postneg
TT	pn	RIGHT	sub
TT	np	LEFT	sub
TT	nn	–	add
TF	pp	LEFT	add; postneg
TF	pn	LEFT	sub; postneg
TF	np	LEFT	sub
TF	nn	LEFT	add
FT	pp	RIGHT	add; postneg
FT	pn	RIGHT	sub
FT	np	RIGHT	sub; postneg
FT	nn	RIGHT	add
FF	pp	–	negload; sub
FF	pn	RIGHT	load; sub
FF	np	LEFT	load; sub
FF	nn	–	load; add

(b) Evaluating complement of +

Table 3-3: Target Path and Operation Sequence for +

and that the target path is RIGHT if the generated code is equivalent to

$$T_2 := T_1 + T_2.$$

Sometimes, it does not matter which operand lies on the target path. This condition is signified by the entry " – ".

4. The final column lists the optimal sequence of operations to be carried out to implement the semantics of the source operator, given the destroyability status of its operands and assumptions about their signs. Though the entries of this column seem independent of the target machine, the important fact required by the analysis here is the cost associated with the operation sequences. That is not the only target machine dependance in the table. Should the machine have interesting instructions such as reverse subtract and reverse divide, the operation sequences entered in this column would be changed accordingly. That is, whenever the target path favours the computation of the minuend (or the divisor) in a destroyable location, and the subtrahend (dividend) is in a non-destroyable location, an occasion to use such an instruction arises. Aside from this tuning of the column with respect to details of some target machine, relativization is fairly simple here and only amounts to filling in the cost of the different operation sequences. It should be noted that the operations in these sequences are not target machine operations. Instead, they maybe thought of as generic operations available on most target machines. Conventional mnemonics have been used to represent these operations. The sequences are to be interpreted relative to the specified target path in the third column described above. So, if the right operand is destroyable and the target path is to the right, and the right operand is assumed to be in a register, then the sequence "sub; postneg" is understood to mean that the left operand is to be subtracted from the register (right operand) and the result complemented.

On visiting a node, this pass of UCOMP performs the node-operator-specific analysis using appropriate tables as described above. The result of this analysis is the minimum costs of computing both the positive value and the complement of the expected result of the computation described by the visited node. Then it sets the destroyability status of the visited node. As either UCOMP or AMD are applied to a node, the last operation performed in this pass of UCOMP is to associate all simple access modes with the current node; this operation sets the assumptions required in the feasible access modes identification of this nodes' ancestor.

Recalling that the visit to the current node is during a bottom-up LRN traversal, and assuming the visted node to be a integer +, the above tables are used as follows:

- Each table has four quadrants, determined by the destroyability of the node's left and right operand. The first step is to choose the quadrant by examining this information in the left and right subtrees of the node.

- Each quadrant has four possible combinations with regard to the assumptions regarding the signs of the node's operands. Inspecting the minimum costs of computing the positive value and the negative value of each of the operands -- recorded in the respective subtrees below -- and looking up the costs associated with the operation sequence associated with each combination in the table, provides sufficient information for deciding which combination of assumptions leads to the minimum cost of implementing the node. As there are two tables for each operator, one for producing the positive value of the desired result, and the other for computing the complement, minimum-cost sequences for both can be identified and the costs recorded as attributes of the visited node. It is understood that the additional costs of fetch and save operations from the store, due to presence of CDEST created information, is incorporated in the above exhaustive analysis.

- Also recorded is the target path to be used in each of the identified cases above.

Some observations are in order before we consider the next pass. The completion of this pass does not bind the rest of the compiler to the generation of the operation sequences identified by the entries of the above tables. The transformations that reflect this analysis, and which are performed by the next phase of UCOMP, are such that a simple code generation scheme would produce a sequence as identified by this pass. It is quite conceivable that better code may be generated as nothing is yet known at this stage about the actual allocation of registers and the final choice of effective address computation mechanism to be used.

Starting with the condition that the root of the expression tree being analysed must yield a result of the correct sign, an analysis of the signs of each of the operands in the expression tree is undertaken. This analysis is obviously dependent on the minimum cost positive and negative alternatives recorded at each node in the previous pass. This done, transformations to the nodes in the tree are effected reflect the chosen sign of the desired result at each node.

3.5. Determination of Feasible Access Modes

The computation of the set of feasible access modes (AMs) applicable to a node in an expression tree is undertaken whenever the node represents an address value, a fetch operation, or a leaf. This activity is integrated with the pass 1 activity of UCOMP; hence the order of visits to nodes is LRN. As in UCOMP, the set of feasible AMs that are applicable at the visted node are computed using the feasible AMs of its descendants. This is done by exhaustive composition of the operator at the visited node with each possible pair of feasible AMs of the descendants. For a target machine with a limited number of access modes, this exhaustive analysis is not expensive. Rich architectures in this regard, like the VAX 11 systems, have a large number of access modes for effective address generation, partly because of many operand types, and partly due to operator / operand access separation and systematic encoding of AMs in the instruction formats. For such architectures, an exhaustive composition is expensive and it contributes significantly to the cost of the phase.

In order to describe the activity of this phase, it is necessary to motivate and introduce our terminology and classification of relevant features of target machines.

storage bases	the storage resources of the target machine, broadly classified as *registers*, *statically allocated store*, and *dynamically allocated store*.
storage classes	a description of the storage bases in terms of each *type* intrinsic to the instruction set of the target machine.
AMs	the set of effective address computation mechanisms in the instruction set of the target machine. This set is parameterized by the type associated with the computed address, and by the context in which it occurs, viz., whether its l-value is required, or its r-value.
operand classes	a set of sets of AMs. Each operand class denotes the AMs that may be associated with specified operand positions in the instruction formats of the target machine, information used by the code generator to filter the usable AMs from the feasible ones identified by AMD.

In a LRN traversal of the IR tree of a basic block, we expect to find address or operand literals at the leaves, and either l-values or r-values at interior operand nodes. On visiting such an interior node, its feasible AMs are determined in terms of the feasible AMs of its descendant(s) and the operator at the node. To capture this notion of AM compositon by the operator, and to initiate this process of composition, we identify AMs that denote literals, basic l-values, and basic r-values:

- *literal AMs*: Modes that are used to obtain literals of the various operand types in target machines. Special cases of literals like 0, 1, 2, 4, etc., often implicit in AMs due to address alignment constraints in target machines, are included here as separate literal AMs.

- *base AMs*: The set of modes that denote basic l-values -- *base* addresses in storage classes. Register names and indexed modes are typical to this set. Also included are VM influenced addressing modes. These modes are not intrinsically different from the usual addressing modes in the target machine. The VM usually reserves some registers of the target machine to address source language related concepts like local variables, actual parameters and temporaries. Uniquely named base AMs are used to identify these special cases.

- *molecular AMs*: The basic r-value AMs. Direct access of *values* of elements in storage classes is made possible through these modes. Typical cases are register mode, indirect register mode, and indexed mode. These modes provide the default for operand value accesses in the feasible AMs determination process described in this section. The default is used whenever the feasible set of AMs for an operand node is empty, a case that arises when the operator at the visited node cannot combine the feasible AMs of its descendant(s) to yield an available AM in the target machine. The use of the default presumes that a move-to-register instruction will be generated, unless the register allocation phase assigns a register for the operand.

- *interesting AMs*: A set of AMs that result from composition of feasible AMs with the operator at the visited node.

From the above classification, it is easy to observe that several target machine AMs are found in more than one class. This is done primarily to ease the problem of relating source language semantics to multiple views of a target machine feature. For specific use by the AMD phase, every described AM has a list of transformations associated with it. A transformation is a pair,

<operator,transformable modes>.

The transformable modes specify conditions to be satisfied by the operands of the operator. For a unary operator, there is just one set of transformable modes. For a binary operator, a set of transformable modes is associated with its left operand and a single transformable mode is associated with its right operand. The interpretation of a transformation is as follows: if the node visted by AMD has the binary operator in the transformation, and if a feasible AM of its left operand is a member of the associated transformable set described in the transformation, and if the transformable mode associated with the right operand in the transformation is a member of the feasible access modes of right operand of the visited node, then the described access mode is a feasible AM of the visited node.

An example of the analysis required to discover all feasible access modes for a node will set the stage for discussing the details of the phase. Consider the simple statement

x: = a[i]

where *x* and *i* are local variables and *a* is a global variable. In BLISS this statement would be expressed as

x = .(a + .i)

where the period represent the "contents of" operator. Assume the target machine to be PDP-11. If the variables *x* and *i* have been allocated to registers, then instruction

MOV A(I), X

implements the statement. Identifying the use of the indexed mode as a feasible AM is the task of AMD. To do so requires assumptions with regard to the disposition of the variables *x*, *a* and *i*. Hence the need for a LRN traversal to determine feasible AMs applicable to a visited node. To systematically perform the analysis, we start with visits to the leaves in the tree representation denoting the variables *x*, *a* and *i*. We do not as yet know whether the interest is in the operand locations or their values. So we associate with each leaf node the base AMs associated with the storage classes to which each of the variables respectively belong. The "contents of" operation on *i* allows the transformation of the feasible AMs associated with *i* to appropriate molecular modes. Here, register mode would imply that the local variables *i* is allocated a register, and access to storage location would imply indexed access via the frame pointer register of the VM (an member of the base AMs). On visiting the node with the + operator, the absolute address AM of *a* and the register mode of *i* combine to give the indexed AM as a feasible mode for the + node. However, the feasible indexed AM of *i* does not combine with other feasible AMs of *a* in any way. The "contents of" operator inherits these AMs as feasible for its operand and passes as feasible modes the register mode and indexed mode to its ancestor, the assignment operator. A variety of code sequences that implement this statement can now be generated, depending on the decisions of the register allocation phase.

1. If *x* and *i* are allocated to registers, then the result could be

   MOV A(I), X.

2. If *x* is in a register and *i* is located in the store then the result could be

   MOV I(R_{FP}), R_I
   MOV A(R_I), X

   where R_{FP} denotes the frame pointer register.

3. The reverse of the above situation could lead to the sequence

   MOV A(I), X(R_{FP})

4. Alternatively, if both *x* and *i* were located in the store, we can expect the sequence below to be generated.

   MOV I(R_{FP}), R_I
   MOV A(R_I), X(R_{FP})

To move on from a simple example to details of the phase, we need to know how to handle common subexpressions, and nodes with operators that do not figure in the transformations associated AMs. The fact that AMD has been applied to such nodes implies that these trees occur in a context expected to yield an address value. Even so, the computation denoted by the visited node is a computed value rather than an address. Our strategy is to make a restricted traversal of the identified subtree in the LRN fashion, and apply the pass 1 activity of UCOMP to the revisited nodes. During this traversal, we make sure to set the descriptor of the result of each node to indicate that an address is in the making. Also, recall that UCOMP has a final operation that sets the default feasible AMs to be the molecular AMs. The restriction on the traversal is primarily with regard to making sure that the subtrees represent address contexts and in avoiding the application of UCOMP to subtrees that represent operand fetches.

Relativization of AMD with respect to the target machine necessitates the construction of a map from IR operators to lists of AM transformations. The construction of this map is straightforward and is derived directly from the description of AMs. On visiting a node, the AMD phase performs the following steps:

1. If the node is a common subexpression, then a restricted application of UCOMP described above is taken up.

2. If the node is a leaf, then either appropriate literal AMs or base AMs are deemed to be the feasible AMs of the node.

3. If there are no AM transformations associated with the operator of the visited node, then too a restricted application of UCOMP to the subtree is performed.

4. Finally, if there are AM transformations associated with the operator of the visited node, then the transformations are applied as described in the example to obtain the set of feasible AMs for the node. If this set turns out to be empty, the molecular AMs applicable to the storage class of the result type of the node are deemed to be its feasible AMs.

As a result of the processing in the AMD phase, a set of feasible AMs is associated with every node in the tree. To achieve this economically, these sets are represented by bit vectors. Even so, in order to compute the feasible AMs for a node could prove expensive either because of the transformation computations of the last step described above, or because of repeated traversals with application of UCOMP to the visited nodes. The basic problem seems to be intrinsic to the issue of discovery of implicit computations. When we are close to the machine, address computations seem to be different and special with respect to computations that yield intended results. If we back off from this point and view the computations from the source language standpoint, then again the distinctions between addresses (l-values) and denoted values (r-values) are clear. However, in between these two very separated stages, the problem can be segmented in many ways, thereby leading to different identifications of the inherent implicit computations. Our solution represents an empirical trade-off between a good identification and the cost to obtain it.

From the preceeding two sections and the above description of AMD, we see a lot of interaction between UCOMP and AMD. A quick review of the overall control of these phases may help place all these details in proper perspective.

1. An LRN traversal of the tree is commenced. Depending on the nature and context of the visited node, either UCOMP or AMD is invoked to process the visited node.

   a. UCOMP determines the destroyability of the result, determines the target path, and assesses the cost of computing the expected and complement values of the result. This information is recorded in the node. As a value is the expected outcome of such a node, it associates as default feasible AMs of this node the molecular AMs of the target machines restricted to the type of the result.

   b. AMD either computes through transformations the feasible AMs to be associated with the node, or invokes a restricted traversal of the subtree at the visited node, applying UCOMP to the revisited nodes.

2. Now that the processes of identification are complete, the input tree is transformed to reflect the least cost computation of the expected value at the root.

3.6. Determining Evaluation Order

The completion of the UCOMP and AMD phases do not completely determine the target path. There are some cases where it does not matter to UCOMP; either choice of the target path leads to optimal code due to the symmetric nature of these cases. Also, target path and evaluation order are not synonymous; target path determines the utility of overloading whereas evaluation order is concerned with minimization of demand for temporaries. A good evaluation order can lead to reduction in the number of registers needed for evaluation on general register machines, it can reduce the number of intermediate load and store operations on single accumulator machines, and it can reduce stack height on stack machines. Obtaining an optimal evaluation order is known to be NP complete. We term our resultant evaluation order to be good because, in our experience, the deviation from the optimum is small.

The algorithm we implement is a variation of a well known algorithm. The inputs to our algorithm are the target path information generated in UCOMP and a flag to indicate whether it is legal to reverse the accepted left-to-right evaluation order of operands. Reversal is assumed to be illegal in the presence of side-effects in the subexpressions of an expression, or if there are common subexpression creations that could affect the value of the expression.

An LRN traversal of the tree is performed to estimate the number of registers required for the evaluation. Using this estimate along with the flag for reversal, the final choice of target path is made for the cases left undecided by UCOMP. Also, the evaluation order is determined. The register requirement estimation process is started off by setting the number of registers required for leaves of the tree that represent user variables to 1. Also, references to common subexpressions require a register to store the copy. As can be seen, the flow of this information is bottom-up. Register requirements for nodes at a higher level depend on those of the subtrees of the node and on the size of the result generated by the operator at the node.

Table 3-4 characterizes the logic of this phase for binary operators. The table is divided into two parts to separately take care of the cases with regard to the reversibility of evaluation of operands. The first three columns of the table provide the inputs to the decision making process. The first column gives the target path information generated by UCOMP. The second column portrays the relationship between N_L and N_R, the number of registers required for the evaluation of the left and right operands respectively. The third column indicates whether any of the operands are such that they require no registers for their evaluation, e.g. constant operands. The next three columns give the desired order of evaluation (by specifying the first operand to be evaluated), the final target path, and an adjustment to determine N, the number of registers required to evaluate the visited node. The adjustment is an increment to $max(N_L,N_R)$. The last column flags those rows of the tables that lead to non-optimal sequences either because the order of evaluation is irreversible, or because the later evaluation of the operand on the target path is preferred. The latter bias indicates the desirability of the target path being maintained in a register. We observe the following patterns in the table:

- Target path decisions are made only in those cases that are left undecided by UCOMP. In these cases, the target path chosen is either the one that requires registers when the other operand does not, or, if both operands need registers for their evaluation, the one that is evaluated last.

- Choice of order of evaluation only exists when the order is reversible in the original expression. When registers are needed by both operands, the order of evaluation is chosen to be the opposite of the target path. This choice is dictated by the desire to keep the target path in a register for immediate use by the operation at the node being visited.

- The symbol β appearing in the column giving the adjustment to N is computed by the formula

  $$\beta = max(0, r(OP) - max(N_L,N_R))$$

  where r(OP) is the number of registers to evaluate OP, the operator at the visited node. For most target machines and operators, r(OP) is one. Hence β should be zero in most cases.

A last possibility in avoiding generation of temporaries is in the short circuit evaluation of expressions that direct control flow. For such expressions, we are not interested in preserving this value in store. For instance, the statement

if $X > 0$ **and** $y < 0$ **then** *s1* **else** *s2*

could be implemented as

```
if x ≤ 0 then goto else;
if y ≥ 0 then goto else;
s1; goto fi;
else: s2;
fi:
```

The algorithms to effect such transformations are fairly standard and can be found in textbooks on compilation [1, 12]. The phase LABEL implements these transformations in our PQC.

At this stage of the compilation process, the intermediate tree represented can be discarded, and

Target Path	$N_L : N_R$	No Reg	Order	Target Path	N increment	Remarks
–	$>$	TRUE	LEFT	LEFT	β	
–	$=$	TRUE	LEFT	LEFT	$\beta + 1$	
–	$<$	TRUE	RIGHT	RIGHT	β	
LEFT	$>$	TRUE	LEFT	LEFT	β	
LEFT	$=$	TRUE	RIGHT	LEFT	$\beta + 1$	
LEFT	$<$	TRUE	RIGHT	LEFT	β	
RIGHT	$>$	TRUE	LEFT	RIGHT	β	
RIGHT	$=$	TRUE	LEFT	RIGHT	$\beta + 1$	
RIGHT	$<$	TRUE	RIGHT	RIGHT	β	
–	$>$	FALSE	LEFT	RIGHT	β	
–	$=$	FALSE	RIGHT	LEFT	$\beta + 1$	
–	$<$	FALSE	RIGHT	LEFT	β	
LEFT	$>$	FALSE	RIGHT	LEFT	$\beta + 1$	*
LEFT	$=$	FALSE	RIGHT	LEFT	$\beta + 1$	
LEFT	$<$	FALSE	RIGHT	LEFT	β	
RIGHT	$>$	FALSE	LEFT	RIGHT	β	
RIGHT	$=$	FALSE	LEFT	RIGHT	$\beta + 1$	
RIGHT	$<$	FALSE	LEFT	RIGHT	$\beta + 1$	*

(a) Evaluation Order is Reversible

Target Path	$N_L : N_R$	No Reg	Order	Target Path	N increment	Remarks
–	$>$	TRUE	LEFT	LEFT	β	
–	$=$	TRUE	LEFT	LEFT	$\beta + 1$	
–	$<$	TRUE	LEFT	RIGHT	$\beta + 1$	**
LEFT	$>$	TRUE	LEFT	LEFT	β	
LEFT	$=$	TRUE	LEFT	LEFT	$\beta + 1$	
LEFT	$<$	TRUE	LEFT	LEFT	$\beta + 1$	**
RIGHT	$>$	TRUE	LEFT	RIGHT	β	
RIGHT	$=$	TRUE	LEFT	RIGHT	$\beta + 1$	
RIGHT	$<$	TRUE	LEFT	RIGHT	$\beta + 1$	**
–	$>$	FALSE	LEFT	RIGHT	β	
–	$=$	FALSE	LEFT	RIGHT	$\beta + 1$	
–	$<$	FALSE	LEFT	RIGHT	$\beta + 1$	**
LEFT	$>$	FALSE	LEFT	LEFT	β	
LEFT	$=$	FALSE	LEFT	LEFT	$\beta + 1$	
LEFT	$<$	FALSE	LEFT	LEFT	$\beta + 1$	**
RIGHT	$>$	FALSE	LEFT	RIGHT	β	
RIGHT	$=$	FALSE	LEFT	RIGHT	$\beta + 1$	
RIGHT	$<$	FALSE	LEFT	RIGHT	$\beta + 1$	**

(b) Evaluation Order is Irreversible

Notes:

* Non optimal evaluation order chosen to make target path evaluated second

** Non optimal evaluation order chosen to guarantee left-to-right evaluation

Table 3-4: Target Path Completion and Evaluation Order Determination

ordered lists of simple trees can be produced for traversal by the register allocation and code generation phases.

4. Review

The previous two sections have presented much detail about a part of a compiler. Lest the overall perspective be dimmed by this mass of detail, we sum up here the goals of the phases described, their salient points, and their shortcomings.

The purpose of the DELAY phases is to put off translation issues till sufficient information is gathered about the source program. Once this information is available, it is used to direct the translation activity such that relatively simple subsequent phases can produce good code. Expending more effort in later phases should lead further improvement in the quality of generated code.

There is obviously a trade-off between the number of translation issues that are put off for later consideration, and hence the amount of work to be done by the DELAY phases, and the quality of code we can expect from this delayed binding. Recall that the object of our study is more to consider the effective use of target machine features as a result of translation than to discover target machine independent optimizations. The position of the DELAY phases in our PQC assumes that some target machine dependent translation has has already been done by the the CWVM phases. These activities concern the nature of the procedure linkage mechanism, the layout of source language data structures in the virtual machine, and the interface to the run-time supports. The translation activities that remain are those that concern the representation of the control flow, and the representation of the basic blocks in the program. The bulk of the computational activity in a program is in the basic blocks. Accordingly, the DELAY phases are entirely devoted to the discovery of efficient mappings of source language basic blocks, expanded to reflect the CWVM decisions, to target. machine computations. The specific goal of the DELAY phases in our PQC is to obtain a controlled unravelling of the intermediate stages of computation that surface because of limited applicative capabilities of target machines.

The task of the DELAY phases is accomplished in two broad steps. The first is to take cognizance of the general structure of computational unit of the target machine, e.g., does the machine have multiple arithmetic units, is it a general register machine, etc.. This analysis is performed by the phase ALGEBRA. The second task is concerned with the issue of discovering a sequencing of the computation such that intermediate stages that surface make minimal demands on target machine resources. The phases CDEST, UCOMP, AMD, EVO and LABEL, all described in the previous section, analyse and restructure the source program so that its expansion to articulate the intermediate stages is controlled and the ground prepared for effective register allocation and instruction selection during code generation. We hope that the preceding two sections have not only provided the structure and logic of these phases, but also provided insight about the process of relativization with respect to target machines and our solutions to this problem.

There are many possible directions for improvement. We mention a few below:

- The discovery of implicit computations and its relationship to the problem of reducing temporaries as well as register allocation is a difficult problem. Our solution is a first cut of the problem. There seems to be a lot of scope for developing good techniques of description of such target machine features with the view that the description can be used to generate the data and algorithms for such analysis.

- Tree transformations as expressed through BONSAI have eased the programming burden in PQCC. The extension of such techniques to other applications of tree transformations in compiling could have large payoffs.

- Attribute grammars have been extensively used as formal handles on the specification of a variety of analyses undertaken during compilation. The extension of these techniques such that their processing is of use in the presence of optimizing transformations provides a rich source of problems worth serious consideration.

- Delayed binding can be applied to the tasks performed by CWVM to realise better storage allocation and management policies. That is, instead of a fixed policy embedded in the CWVM phases, the issue can be considered on a program by program basis. Interprocedural analysis as well as global flow analysis can be used to provide a better basis for making the CWVM decisions.

Acknowledgements

A number of people have contributed - some substantially - to the PQCC project. The participants are (in alphabetic order) Mario Barbacci, Ron Brender, Ben Brosgol, Steve Byrne, Rick Cattell, Reidar Conradi, Frank DeRemer, David Dill, Gary Feldman, Peter Hibbard, Paul Hilfinger, Andy Hisgen, Regis Hoffman, Steven Hobbs, Paul Knueven, Bruce Leverett, John Nestor, Joseph Newcomer, Andy Reiner, Bruce Schatz, David Stryker, Lee Szewerenko, Franco Turini, and Gideon Yuval. We are grateful to Bruce Leverett and Bhubaneshwar Mishra for their reviews of an earlier draft.

References

1. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, 1972.

2. F.E. Allen et al. "The Experimental Compiling System." *IBM J. Research and Development 24*, 6 (Nov. 1980), 695-715.

3. B. M. Brosgol, J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf. TCOL$_{Ada}$: Revised Report on An Intermediate Representation for the Preliminary Ada Language. Tech. Rept. CMU-CS-80-105, Carnegie-Mellon University Computer Science Department, Feb., 1980.

4. R.G.G. Cattell. *Formalization and Automatic Derivation of Code Generators*. Ph.D. Th., Carnegie-Mellon University, April 1978. Available as Technical Report CMU-CS-78-115

5. Digital Equipment Corporation. VAX-11 Architecture Handbook. 1979.

6. Digital Equipment Corporation. PDP-11 User's Handbook. 1971.

7. J. Feldman and D. Gries. "Translator Writing Systems." *Comm. ACM 11*, 4 (1968).

8. M. Ganapathi. *Retargetable Code Generation and Optimization Using Attribute Grammars*. Ph.D. Th., Computer Science Department, University of Wisconsin, Madison, 1980.

9. H. Ganzinger, R. Giegerich, U. Moncke and R. Wilhelm. A Truly Generative Semantics-Directed Compiler Generator. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, 1982, pp. 172-184.

10. R. Glanville and S. Graham. A New Method for Compiler Code Generation. Fifth ACM Symposium on Principles of Programming Languages, SIGPLAN-SIGACT, Jan., 1978, pp. 231-240.

11. G. Goos and W. A. Wulf (editors). Diana Reference Manual. Tech. Rept. CMU-CS-81-101, Carnegie-Mellon University Computer Science Department, March, 1981.

12. D. Gries. *Compiler Construction for Digital Computers*. John Wiley and Sons, Inc., 1971.

13. J. Horning, W. McKeeman and D. Whortman. *A Compiler Generator*. Prentice Hall, 1972.

14. S.C. Johnson and M.E. Lesk. "Language Development Tools." *Bell System Technical Journal* 57, 6 (July-August 1978), 2155-2175.

15. S.C. Johnson. A Portable Compiler: Theory and Practice. Fifth ACM Symposium on Principles of Programming Languages, SIGPLAN-SIGACT, Jan., 1978, pp. 97-104.

16.. N.D. Jones and D.A. Schmidt. Compiler Generation from Denotational Semantics. Semantics Directed Compiler Generation, 1980. also *Lecture Notes in Computer Science, Vol. 94*

17. B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, W.A. Wulf. "An Overview of the Production Quality Compiler-Compiler Project." *Computer 13*, 8 (Aug. 1980), 38-49.

18. J. R. Nestor, M. Beard. *Front End Generator User's Guide*. Carnegie-Mellon University Computer Science Department, 1981. Internal Documentation.

19. J.R. Nestor, W.A. Wulf, D.A. Lamb. IDL - Interface Description Language: Formal Description. Tech. Rept. to appear, Carnegie-Mellon University Computer Science Department, Aug., 1982. Second Edition

20. Wm.A. Wulf, M. Barbacci, B. Brosgol, R.G.G. Cattell, R. Conradi, D. Dill, S.O. Hobbs, P. Knueven, B.W. Leverett, J.M. Newcomer, A.H. Reiner, B.R. Schatz, D. Stryker, F. Turini. Specifications for the Phases of the PQCC. Carnegie-Mellon University Computer Science Department, Sept., 1980.

21. Joseph M. Newcomer. PQCC Intermediate Machine Description Language. Carnegie-Mellon University Computer Science Department, July, 1981.

22. K.-J. Raiha, M. Saarinen, E. Soisalon-Soininen and M. Tienari. The Compiler Writing System HLP. Research Report A-1978-2, Department of Computer Science, University of Helsinki, March, 1978.

23. D. Stryker. FLANGE - A Flow Analysis Generator. Users Manual. Internal Documentation

24. W.A. Wulf, D.B. Russell, and A.N. Habermann. "BLISS: a Language for Systems Programming." *Comm. ACM 14*, 12 (Dec. 1971), 780-790.

25. W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke. *The Design of an Optimizing Compiler.* American-Elsevier, 1975.

26. W.A. Wulf. PQCC: A Machine-Relative Compiler Technology. IEEE 4th International COMPSAC Conference, Chicago, Oct., 1980, pp. 24-36. Also available as CMU Technical Report CMU-CS-80-144

27. W.A. Wulf. BONSAI A Tree Transformer Generator. Carnegie-Mellon University Computer Science Department, 1981.