Symbolic Debugging Through Incremental Compilation in an Integrated Environment*

Peter Fritzson

Linköping University, Linkoping, Sweden

It is demonstrated that fine-grained incremental compilation is a relevant technique when implementing powerful debuggers an incremental programming environments. A debugger and an incremental compiler for PASCAL has been implemented in the DICE system (Distributed Incremental Compiling environment). Incremental compilation is at the statement level which makes it useful for the debugger which also operates at the statement level. The quality of code produced by the incremental compiler approaches that required for production use. The algorithms involved an incremental compilation are not very complicated, but they require information that is easily available only in an integrated system, like DICE, where editor, compiler, linker, debugger and program data-base are well integrated into a single system. The extra information that has to be kept around, like the cross-reference database, can be used for multiple purposes, which makes total system economics favorable.

INTRODUCTION

Two possible definitions of incremental compilation are given in [3]. First, the incremental compiler is to expend an effort during the translation process that is roughly proportional to the size of the change to the source program made by the programmer. A second possible task for an incremental compiler is to allow the programmer to execute his program up to a certain point, stop and edit his source file, and then resume execution where he stopped. In the following we assume that an incremental compiler, or better: an incremental system, has both of these properties. We will see that capabilities of a symbolic debugger can be greatly extended through usage of such an incremental compiler.

*This research was supported by the Swedish Board of Technical Development.

Address correspondence to Peter Fritzson, Computer Science Department, Linköping University, S-581 83 Linköping, Sweden.

The Journal of Systems and Software 3, 285–294 (1983)

PREVIOUS WORK

Examples of incremental programming environments for block-structured languages with nested statement structure are INTERLISP [17], Cornell Program Synthesizer [16], GANDALF [12], PATHCAL [18], ECL [4], LISPEDIT [1], etc. The smallest compilation unit in systems like INTERLISP, GANDALF, ECL, and LISPEDIT is the procedure body.

INTERLISP, ECL, the Synthesizer, PATHCAL, and LISPEDIT all permit continued execution after program modifications, since these systems contain interpreters. GANDALF is a compiling environment that permits continued execution after local modifications; it does not currently (June 1982) keep track of the machine-code positions of statements after recompilation [5].

A paper by Rishel, [14] gives a thorough treatment of statement-level incremental compilation for line-oriented languages like BASIC or FORTRAN. However, all techniques proposed in that paper requires insertion of extra instructions between statements. That is not needed by the techniques proposed here, which have been implemented in the DICE system.

Kahrs [11] describes an interactive programming system for a block-structured language along ideas discussed in [13]. The system contains an integrated interpreter/compiler, a so-called Tree Factored Interpreter, which can switch between interpretation and code generation at arbitrary points in the abstract syntax tree. The code is highly fragmented, however—one procedure body for each tree node—and the code is not separable from the tree since it contains indirect procedure calls through pointers kept as attributes to tree nodes.

OVERVIEW

This paper presents two topics: implementation of a debugger through use of an incremental compiler, and

0164-1212/83/$3.00

techniques for fine-grained incremental compilation. Both the debugger and the compiler are components of the highly integrated programming environment DICE (Distributed Incremental Compiling Environment) which aims at providing programmer support in the case where the programming environment resides in a host computer and the program is running on a target computer that is connected to the host.

Commands to the debugger command level include all legitimate PASCAL statements. The debugger is machine independent—it calls the incremental compiler which generates code for evaluation of commands, or modifies the machine code of the target program for insertion of breakpoints, etc. Essentially all machine dependences are isolated inside the code generator of the incremental compiler.

Debug code at conditional breakpoints, etc. is executed as efficiently as the compiled code of the normal program since it also has been compiled by the incremental compiler. Essentially no extra code is inserted into the target program in order to be able to debug it.

Incremental compilation is at the statement level which makes it particularly useful to the debugger, since it operates at the statement level. Statement level recompilation usually is an order of magnitude faster than procedure-wise recompilation, which may be important for big programs or on loaded time-sharing computers. Also, this will permit acceptable interactivity even after changes to certain global declarations. The extra information needed in the program database to support incremental compilation is almost identical to that needed for debugging. The system allows continued execution after most program changes.

The algorithms for incremental compilation are centered around a tree representation of the source program, and are applicable to a wide class of languages. It is illustrated by the DICE incremental PASCAL compiler.

Program editing is performed through a full-screen structure editor. The editor marks new or changed nodes in the tree of the current procedure. Recompilation of changed statement nodes is performed during a preorder traversal of the tree. New and old machine code is merged; and branch instructions in the old code are updated. Branch updating can be incremental on the statement level for languages with well-formed control structures on machines with PC-relative goto instructions.

Incremental compilation is consistent, i.e., no degeneration of code quality occurs after many edit-recompilation cycles. Dynamic linking is made easy by generating position-independent code and having an extra level of indirection for procedure calls.

The main components of the DICE system are shown in Figure 1.

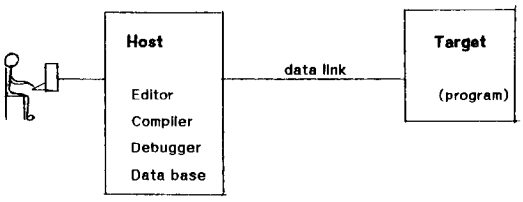


Figure 1. The main components of DICE, Distributed Incremental Compiling Environment.

PROGRAM DATABASE

Most information in the data-base is packaged in procedure-units. Each unit contains the source tree program representation, machine code, local symbol table, and dependency lists for declarations that are referenced in other procedures. The symbol table is permanent and allows updating.

IMPLEMENTATION STATUS

Currently (July 1983) DICE is implemented in 20000 lines of INTERLISP on a DEC20 computer. The incremental compiler accepts essentially full PASCAL including input–output (currently not including sets and procedural parameters) and produce PDP11 code of reasonable quality. A very rudimentary program database exits. A full-screen structure editor is implemented, but it still lacks adequate text editing facilities. A screen-oriented debugger is partly implemented and currently allows all PASCAL statement types as commands. It also allows setting of breakpoints, single-stepping and flow-tracing for control of execution. The debugger operates on the program in the target computer via DecNet.

ECONOMY OF AN INTEGRATED SYSTEM

An integrated programming environment contains tools such as editor, compiler, linker, debugger, data base manager, etc. These tools cooperate. For example, the editor informs the incremental compiler which statements need to be recompiled. Also, the parser can be used to parse program text or debugging commands, and it also serves as a general command level user interface. This sharing of resources is true also for data structures, as exemplified by the source-machine-code mapping and the cross-reference database.

The mapping between source code and machine code at statement level is used both for incremental compilation and for debugging purposes like single stepping and insertion of breakpoints.

The cross-reference and static analysis database (similar to the Masterscope subsystem in INTERLISP) serves at least three purposes:

It aids the programmer in understanding his program and in finding interesting parts of it.

It makes incremental recompilation possible even after edits to global entities, since the cross-reference database contains information about which procedures reference such entities.

It is used by the debugger to implement traces and watches on variables, since the system contains information about which procedures reference or modify a certain variable. The incremental compiler can be called to insert extra code around relevant statements in such procedures.

An incremental compiler provides a single interface to the low-level machine. Besides the traditional translation task, it can be used instead of a command level interpreter, or it can be utilized by a machine-independent source level debugger in implementing breakpoints, single-stepping, etc.

A DEBUGGER IMPLEMENTED THROUGH INCREMENTAL COMPILATION

The DICE debugger is oriented towards programs that may execute on small machines under real-time constraints. It is essential that a minimal amount of extra code should be executed in the target program during debugging, compared to ordinary execution. This may of course also be valuable on an ordinary time-shared machine. It is also important that one should be able to invoke the debugger if a run-time error occurs during an ordinary execution.

Another requirement is that all legal PASCAL statements should be accepted as legal commands by the debugger; continued execution after source program modifications should also be permitted.

Our approach to meet these requirements is to use an incremental compiler to generate code for complex commands and to make necessary modifications in the object code for conditional breakpoints, etc.

Since fast response to user commands is important, and single-stepping should be provided, the compiler has been designed to be incremental on the statement-level.

We will consider some of the more important functions that are normally found in good debuggers, and show how they can be implemented in an incremental compiling context.

EVALUATION OF EXPRESSIONS AND FUNCTION CALLS

The ability to evaluate expressions or call functions is just a special case of the general ability of the DICE debugger to accept any PASCAL statement as a command. A given statement-command is compiled and sent to a free memory area in the target computer where it is executed. Output from write-statements executed on the target is sent to the host through a special logical DecNet channel on the same physical link, and eventually reaches the users terminal.

Write-statements are useful for printout of expression values; assignment statements are used to change variable values and procedure calls are useful to be able to perform small tests on single procedures or on subcomponents of the subject program. Small loops are sometimes useful in order to initialize a data structure.

DECLARATIONS AS COMMANDS

The user may even declare new types or new variables that can be useful during the process of locating a bug through a series of small experiments. The following sequence of three commands is perfectly valid:

```
type FOOREC = record A:integer;
    ARR:array [1..10] of integer end;;
var X:FOOREC;;
for I: = 1 to 10 do X.ARR[I] : = 3;
```

Such temporary variables or types exist only during the debugging session and are not inserted into the subject program unless it is specifically requested by the user.

THE DEBUG COMMAND LANGUAGE

Our experience from the INTERLISP system indicates that it is very useful if the source language is a subset of the debug language. This is also true in the DICE system.

Is the source language powerful enough as a debug language? In the case of LISP we know from experience that it is. For PASCAL, one might have to extend the debug language in order to overcome the lexical, scope rules, and to make the flexibility of the write statement more generally available, i.e., generic argument types and variable number of arguments. Further experience will indicate what is necessary. Essentially all the debug commands now available in INTERLISP or LISPEDIT could readily be expressed in PASCAL syntax after some adaptation.

Also, the most common commands should be assigned to function keys or control characters in order to make the user interface more convenient.

TEXTUAL WATCHES ON VARIABLES

A variable in a PASCAL program can be modified either by an assignment statement or as a VAR parameter in a function or procedure call. The system can use the cross-reference data-base to find these positions in the

source tree, and insert extra code to trace variable values or to generate a break when such variables change value. This kind of watch on variables is called *textual watch* since it is coupled to an occurrence of a variable name in the program text.

The selective code insertions used for textual watches slow down execution much less than the traditional method common in many PASCAL debuggers, which is to test values of relevant variables at runtime after each executed statement.

Ideally one would like special hardware facilities for watches. An approximation to this may be found for computers with a paging virtual memory. The method is to write-protect the page that contains the interesting variable and to use page trapping on write access. Problems may arise if the variable is on the stack (the subroutine call and return instructions on the VAX may not work if the stack is write-protected), or if an inner loop references the write-protected page.

ALIASED WATCHES ON VARIABLES

The traditional method of checking the contents of a certain memory location after the execution of each statement will detect the "first" statement (first in the sense of execution order) in the program which changes the value of any of the aliases of a certain variable. We call such watches *aliased watches* on variables.

The DICE debugger normally uses textual watches on variables. The question is: can aliased watches be implemented in our scheme? The answer seems to be yes, at least to a large extent.

The information kept in the cross-reference database can be used to identify most of the code positions where aliases might occur, and conditional breakpoints can be inserted at these positions. This will give the same effect as the traditional method and still will not slow down execution as much as the traditional method, which is important when debugging programs that execute under real time constraints. One drawback of this method may be the great number of possible aliases and corresponding code insertions in certain situations, e.g., consider pointers to records of a variant record type that is often used.

POSITIONING

The DICE debugger uses the full-screen structure editor in order to provide a simple means of pointing to the program-fragment where a certain operation is to be performed, e.g., an insertion of a breakpoint. The user interface is simplified, since only a single set of positioning commands has to be learned. The idea to integrate the editor and execution-monitor has previously been applied successfully both in the Cornell Program Synthesizer and in LISPEDIT. We have often experienced the lack of such an integration between the editor and the debug package in the INTERLISP system, although it is possible to evaluate any LISP expression inside the editor.

SINGLE-STEPPING

An important requirement is that it should be possible to provide single-stepping for compiled programs which have no extra code inserted between each statement for debugging purposes.

We use the alternative method of calling the incremental compiler on each step during step-wise execution in order to insert a new breakpoint before the next statement and eliminate the breakpoint that was previously inserted before the current statement. On entry to a loop-construct, a temporary breakpoint is also inserted immediately after the loop in order to catch the exit from the loop.

VARIED STEPPING

It is often useful to be able to request atomic completion of the execution of the current statement, e.g., it may be tedious to step through a small loop hundreds of times. This step command is called *long resume* in the Synthesizer and *run* in LISPEDIT. Its implementation in the DICE debugger is straightforward: insert a temporary breakpoint at the beginning of the next statement on the same level, or at a higher level if it was the last statement on this level.

One complication arises, however, if the execution of the current statement causes a recursive call to the current procedure. An unwanted break, which is caused by the inserted temporary breakpoint, will happen during the execution of the new invocation of the current procedure.

This problem is solved if a conditional breakpoint is inserted instead of the unconditional breakpoint. The condition is a test that the value of the framepointer corresponds to the correct invocation of the current procedure.

A general step-command is called *come* in LISPEDIT. It means that if the user points to a certain program position and issues the command *come*, then the system will execute until it hits the indicated position. This command is easily implemented in the DICE debugger: simply insert a temporary breakpoint at the indicated position and remove all other temporary breakpoints.

CONDITIONAL BREAKPOINTS

A conditional breakpoint is written as an ordinary if-statement in PASCAL. It is compiled and inserted into the object code by the incremental compiler. Since the extra code associated with the conditional breakpoint is compiled with the regular PASCAL compiler, it will have the same quality as ordinary compiled code and it will usually exact a small amount of extra execution time.

CONTINUED EXECUTION AFTER PROGRAM MODIFICATIONS

One of the important goals in the DICE system is to allow continued execution after program modifications. This goal is based on our own extensive experience with the INTERLISP system, which has shown us that it is valuable to be able to correct small bugs during the course of an execution. According to [1] users of the Lispedit system tend to fix bugs in the same way.

Continuation after editing of procedures that currently have no activation records on the stack poses no special problems. The same is true if global declarations are added.

If a procedure is changed that has several invocations on the stack, then it might not be wise to continue execution at the current point. Instead execution should be restarted at the first invocation of the current procedure, or higher. This will be easy if there exist states in the call chain that are relatively independent of previous side-effects, e.g., we know that relevant data-structures are initialized at a certain level. Otherwise a lot of manual undoing of side-effects will be necessary before restart.

Edits of declarations of global data structures can cause severe problems in the general case. The system will take care of the code, but existing data structures must be updated according to the changed declarations. In the general case reinitialization of data-structures is left to the programmer, but some common simple cases may be identified where the system could be helpful. For example, when changing the maximum number of elements of an array, the system could copy the relevant part of the old data into the new array. When a new field is added to a record type, the system could trace all pointers to data items of that type, reallocate the data item, copy data from the old data item, and fill the new field with some null value like NIL or zero.

If an active procedure is edited, parts of its code may be moved. Since this might affect procedure return addresses, the stack must be searched for such addresses and updated if needed (Both the GANDALF debugger and the INTERLISP code swapper [7] update return addresses on the execution stack.) The code of other procedures is not affected since procedure calls go one level indirect. If formal parameter declarations for a procedure are edited, all calls to that procedure must be recompiled (and probably edited). If such a procedure is active, one might imagine manipulating argument areas on the stack so that subroutine returns still would cause no problem, but one might ask if this is meaningful. It is probably better to pop a few activation records from the stack.

At last we consider modification to the procedure that contains the current stopping-point. The incremental compiler will keep track of the machine-code position of the current statement, even if extensive modifications are done to surrounding statements. This makes it possible to continue from the current statement or from any other statement in the procedure. The user must take care of side-effects himself. Changes to declarations of local variables will be handled by the incremental compiler (see the part about incremental variable allocation). The compiler will also restructure the contents of the current activation record.

GOALS FOR FINE-GRAINED STATEMENT-WISE INCREMENTAL COMPILATION

Fast recompilation. A responsive interactive programming environment is obtained.

Consistent compilation. Code quality should not degenerate after repeated program changes followed by incremental recompilations.

Preservation of execution state. Execution should be able to continue after most program changes.

Good quality of the machine code produced. The code should be good enough for production use in order to demonstrate the feasibility of incremental compilation concepts in practice.

Separability. The compiled program should be separated from the source code so that it can execute outside of the program development system.

Concerning compilation speed: preliminary measurements on the recompilation time for procedures after making a small modification indicate a speed improvement of a factor of ten to 20 in most cases, compared to recompiling the procedure from scratch.

The greater part of the code generator of the DICE PASCAL compiler originates from the Portable C Compiler [10]; it has simply been translated by hand from C to INTERLISP. This causes the quality of code produced by the DICE PASCAL compiler to be almost equivalent the code produced by the Portable C Compiler, which has been used to compile many UNIX systems (although the C language permits the program-

mer to do more source-level optimizations than does PASCAL).

The current DICE incremental PASCAL compiler is fully consistent in code generation (see Figure 2). The code quality does not degenerate after arbitrarily many edit-recompilation cycles. It may not be fully consistent in memory allocation, however, since harmless holes in stack-frames may arise after deletion of variable declarations.

AN OVERVIEW OF FINE-GRAINED INCREMENTAL COMPILATION

This compiler is centered around an abstract syntax tree representation of the source program. During an editing session, the editor marks those nodes of the current procedure that represent changed, inserted, or deleted statements or declarations.

When the user has finished editing, or wants to continue execution after he has made program modifications, the compiler does a preorder traversal of the source code trees of all edited procedures, and recompiles all statement nodes that have been changed or inserted.

The compiler achieves flexibility by dividing the translation into three passes: (1) name binding, (2) type analysis and variable allocation, and (3) generation of machine code.

Changes to Declarations

Changes to declarations occur when the user modifies an existing variable or type declaration, or inserts a new one. This may force changes in memory allocation of variables, i.e., their size and position. It also invalidates the compiled code of all statements that contain affected variables. When recompiling a procedure, the incremental compiler will search the tree, and recompile the smallest enclosing statement nodes that references such variables.

After a change to a global declaration, the compiler will use dependency information maintained in the program data-base, which tells which procedures depend on the current declaration.

Changes to Statements

This kind of program modification is much more local in scope than changes in declarations, and it usually only invalidates the compiled code associated with the changed statements. Sometimes, however, such changes will affect memory allocation of temporaries and constants.

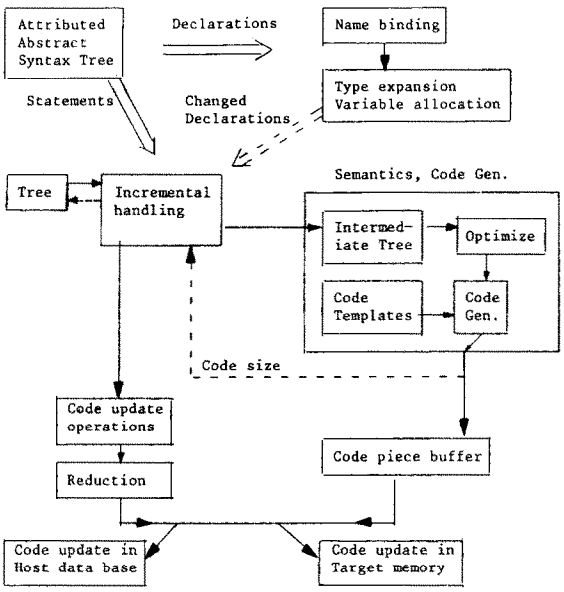


Figure 2. A block diagram of the DICE incremental compiler.

INCREMENTAL COMPILATION AT THE STATEMENT LEVEL

This problem can be decomposed into *two subproblems:* to find an appropriate mapping between the source code and the machine code, and to find an efficient method to update branch and goto instructions in the machine code of a procedure body.

The mapping between the source level and the machine code level should have the following properties:

Given a source statement, it must be easy to identify the associated machine code. This is for purposes of incremental compilation, and for finding statement positions for breakpoints during debugging.

Given a machine code address, it must be easy to identify the corresponding source code statement. This is for debugging purposes only. The user should be able to stop the target program when it is running, and the system should identify at which statement in the source code the program was stopped, and generate a breakpoint at that position.

The mapping should be easily updated after program edits. Incremental updating properties are of course desirable in an incremental system.

AN IMPLEMENTATION OF THE MAPPING BETWEEN SOURCE CODE POSITIONS AND MACHINE CODE POSITIONS

Binary machine code is the only low-level code that is kept in the database or in the target computer. The *codesize* attribute of each statement node in the tree

implements the mapping between source and machine code. The address of a statement can easily be found by adding the procedure start address with the codesizes of preceding statements.

During the edit-session, the editor marks changed and deleted statement nodes by modifying an attribute called *editmark*. A new inserted node has the *editmark* value *changed* and *codesize* value zero; an unchanged previously compiled node has editmark value *unchanged;* an old statement that has been edited is also marked as *changed* and finally, a node which is to be deleted is marked *deleted.* Deleted nodes with nonzero codesize cannot be removed immediately by the editor because then the old machine code would remain. Such nodes are eventually physically removed by the incremental compiler.

The previously mentioned attributes *codesize* and *editmark* are synthesized for compound statements like BEGIN-END, IF-THEN-ELSE, WHILE, REPEAT, FOR, WITH, and CASE statements in PASCAL.

The editmark attribute is synthesized by the editor, and the codesize attribute is synthesized by the incremental compiler according to the following rules (we assume that S is a compound statement that has some sons).

Attribute *editmark:*

If all sons of S are *deleted* then S is marked as *deleted*

If all sons of S are *unchanged* then S is marked as *unchanged* otherwise S is marked as *changed.*

Attribute *codesize:*

The codesize of S is the sum of the codesizes of all its sons.

(One might regard the backward branch of a WHILE-statement as a virtual son in order to preserve the truth of the above rule for *codesize*).

Initially, all new nodes have *editmark* value *changed* and *codesize* value zero.

A SMALL EXAMPLE SUBTREE

A subtree corresponding to the statement

while $i <= 15$ do $i := i + 2$;

is shown decorated with the attributes *(codesize, editmark),* together with the PDP11 machine code in Figure 3. The constant *2* has been changed to *1* by an edit, which also has changed the *editmark* attribute.

MERGING OLD AND NEW MACHINE CODE

The strategy for merging code is actually quite simple. To begin with, reserve a memory area where the updated code will be stored. A variable OLDPOS holds the current machine code position in the old code, and

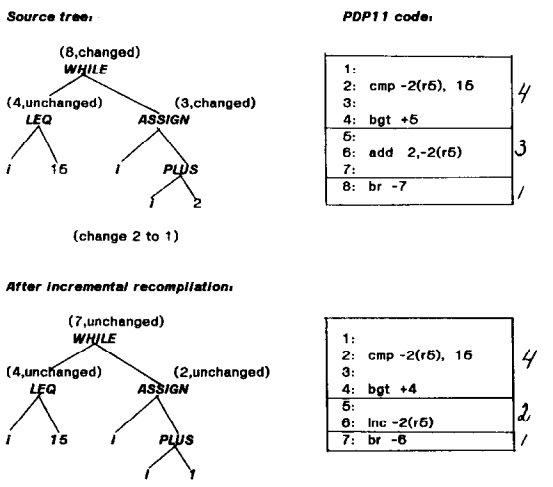


Figure 3. An example of an incremental recompilation of a while-loop in the form of an abstract syntax tree.

a variable CURPOS holds the current position in the merged code. Both location counters are relative to the beginning of the current procedure. Do a preorder traversal of the tree and during that traversal perform the following:

If a node is *unchanged,* copy its code to the new area and advance OLDPOS and CURPOS by an equal amount *codesize.*

If a node is marked as *deleted,* skip its code, i.e., advance OLDPOS but do not change CURPOS.

If a node is *changed,* first compile the node and store generated code into the new memory area. The CURPOS counter is automatically incremented by the code generator during code production. Thereafter, advance OLDPOS by the old value of the attribute *codesize,* and update *codesize* to be the size of the new code piece that was generated.

It is important that CURPOS always refers to the current position in the updated machine code, in order that the *back-patching* of label references in new branch instructions should be done correctly. A more detailed description of the alogrithm may be found in [6].

In the current implementation, new machine code is temporarily stored in a code buffer, and the sequence of insert/delete operations are temporarily stored and not executed immediately. This approach permits the the removal of some of these update operators, which is described in Figure 4.

Before the sequence of insert/delete operations are performed, further reduction of the number of update operations is possible if the copy part of insert operations are extracted. This enables the system to detect the important special case where the total net expansion

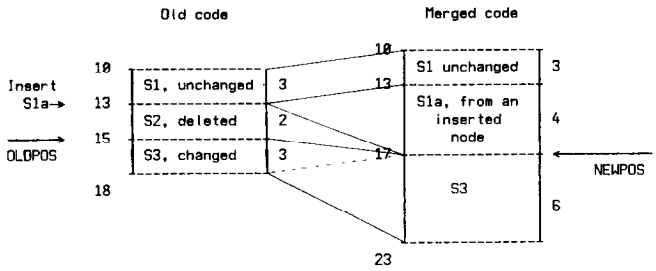


Figure 4. An example of the merging of old and new machine code, together with deletion of some old code.

is zero. Figure 5 shows the saving of expand/contract operations.

If all contractions are performed before the expansions, at most $2N$ word moves are needed for a code piece of size N words.

In the DICE system, machine code is updated simultaneously in the target computer address space and in a code area of the program database. Simple replacement and data movement commands are sent to routines in the target computer over the network link.

INCREMENTAL UPDATING OF GOTO INSTRUCTIONS

Control structures in high-level languages are usually compiled to a number of *goto* or *branch* instructions in the machine-code. If the code inside a construct, i.e., a while-loop, expands or contracts during incremental recompilation, the addresses of certain goto-instructions must be updated. This may be hard or easy to do incrementally depending on the source language and the target machine instruction set. We will consider two methods which are applicable to different classes of languages and machines.

INCREMENTAL BRANCH UPDATING ON MACHINES WITH RELATIVE GOTO INSTRUCTIONS

This method relies on the fact that structured-programming constructs like WHILE-loops transfers control in a local well-structured manner. The only exception from this rule in the language PASCAL is the GOTO statement—this is yet another argument against goto programming. This updating method also requires the existence of PC-relative branch instructions in the target computer instruction set, which makes it possible to move code associated with compound statements without invalidating the code.

Figure 5. Reductions in the number of insert/delete operations and conversion to cheaper operations like copy.

Before reduction:		*After reduction:*	
Update Operations:		Code Move Ops:	Transfer Ops:
Insert 4 words at 13 Delete 2 words at 13 Delete 3 words at 15 Insert 6 words at 18	=>	Expand 5 at 13,	Copy 4 to new 13 Copy 6 to new 17

Each control structure is compiled in a predefined way; it is known which parts of the code contain branch instructions which may need updating. In the current compiler, a hand-written procedure for each type of control-structure takes care of the branch updating. These procedures should not be too hard to generate automatically, e.g., from a denotational description of the control flow aspects of PASCAL. Sethi, [15] has done something similar to that in a small nonincremental compiler for a subset of C.

An example, the WHILE-statement:

Syntax: WHILE ⟨pred⟩ DO ⟨statement⟩
Record: TYPE whilestm = RECORD pred : expr;
body : statement END;

Predefined machine code structure of a WHILE loop:

L1: ⟨code for conditional branches to L2⟩
⟨code for statement body⟩
L2: ⟨branch to Li⟩

The branches to L1 and L2 need be updated only if the code for the statement body changes size. A special procedure scans through the relevant code portion and identifies those branch instructions that must be updated. More details on the updating algorithm, are given in [6].

If the computer has both short branch instructions with a limited range and long branches with unlimited range, then short branches sometimes may have to be converted to long ones. This means that sometimes an extra traversal of the tree is required since conversion to long branches causes insertion of binary code.

GOTO statements must be handled specially; all such instructions in the procedure body must be checked and updated if needed.

BRANCH UPDATING FOR A WIDER RANGE OF MACHINES

On machines that lack PC-relative branch instructions our previous incremental algorithm for updating branch addresses will not work because branch instruc-

tions in the code for statements like IF statements and WHILE statements will be invalid if the code is moved. A single insertion at the beginning of a procedure body can thus invalidate all branch instructions in its machine code.

Thus, if a code expansion or contraction occurs, all branch instructions in the procedure body must be scanned and updated. (A bit-vector marking all branches can be maintained for each procedure body.) This means a slightly increased overhead which is proportional to the size of the procedure body, but the scanning process is still very fast compared to recompilation of the procedure from scratch. This method is equivalent to the traditional loading process, although it is done incrementally for each procedure body.

CONFLICTS BETWEEN INCREMENTALITY AND CODE OPTIMIZATION

The current DICE compiler uses the same code generation strategy as the portable C-compiler: i.e., no intermediate results in temporary registers are kept between statements, and optimizations are performed only inside statements. This goes along well with symbolic debugging and incremental compilation, but of course gives lower object code quality.

There is a conflict between optimization and incremental compilation: global optimizations introduce dependencies between different parts of the program, which will destroy the possibility for incrementality. We are currently starting to investigate if there exist a reasonable compromise between optimization and incremental compilation for medium-sized code pieces (5–10 small statements).

The problem of combining some degree of optimization with incremental compilation is similar to the problem of providing source language debugging on optimized code. [8] gives a thorough treatment of part of the debugging-optimization problem.

Is the code quality produced by this compiler acceptable for normal usage? As mentioned before, it is comparable to the quality of the code which is generated by the Portable C Compiler. The enormous amount of software that has been produced using that compiler seems to indicate that there are relatively few cases where global optimization might be essential.

If global optimization is introduced in this compiler, it should probably be as optional extra passes over a procedure body as a unit. This will, however, destroy the transparent debugging behavior that is possible with statement-wise compilation. One strategy to preserve the debugging facilities would be to replace the optimized procedure body by an unoptimized version as soon as the user tries to plant a breakpoint inside it, or single-step through it. This will work for all breaks except those generated by keyboard interrupt or runtime error.

THE IMPACT OF INCREMENTAL COMPILATION ON VARIABLE ALLOCATION

In this section we will consider variable allocation in the context of incremental compilation. The goal is to choose storage allocation schemes that minimize the amount of recompilation without sacrificing efficiency of compiled code. The following discussion is centered around variable allocation in a procedure activation record; static allocation of global variables and constants can be regarded as a special case.

A typical activation record, or stack-frame, for the language PASCAL is shown in Figure 6. It shows both a frame pointer, FP, and a stack-pointer to top of stack, SP. Parameters, local variables, and (sometimes) temporaries are addressed with constant offsets relative to FP. It might be unnecessary to have both FP and SP, since all addressing can be done relative to SP.

Parameters are constrained to reside at the bottom of a stack-frame in their declared order, since they are pushed on the stack at procedure call.

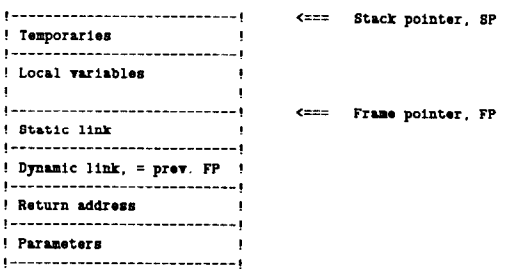
If a variable does not increase in size after editing its declarations, only statements referring to that variable need be recompiled.

If a variable declaration (not a parameter) is deleted, only statements referring to that variable need be recompiled. This will usually result in an unused hole in the stack-frame. This hole can be used to accommodate future local variables, or it can be eliminated by packing the stack-frame, which will usually affect the positions of other local variables and cause recompilation of more statements.

Incrementality for Variable Insertions

If we have both a Frame Pointer, FP, and a Stack Pointer, SP, this redundancy makes possible the insertion of new variable declarations, without forcing recompilation of existing source code.

Figure 6. An activation record for PASCAL with both stack pointer and frame pointer.



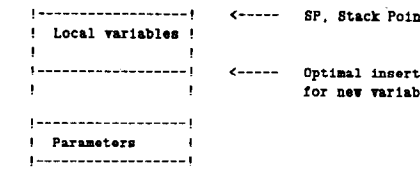


Figure 7. An activation record for PASCAL where a single register functions as both stack pointer and frame pointer.

If there exist several invocations of the current procedure on the stack, then it might be hard to update the previous activation records in a meaningful way. Instead it could be better to pop these activation records from the stack and restart execution from an earlier position (See Figure 7).

If everything is referenced relative to SP, as in the figure 7, an insertion of a new local variable will cause changed offsets for all parameters. The advantage is more efficient code for procedure entry and return.

FURTHER WORK

To this date (July 1983) a static analyzer is still missing from the system. Also, further research is needed into the organization of the program database and the user interface. It could be interesting to study debugging in an incremental system that also supports concurrency. Another topic is to study the use of special hardware like an In Circuit Emulator together with a tool such as an incremental compiler in order to aid debugging and to present the low-level data collected by such hardware in a higher-level fashion.

CONCLUSION

It has been demonstrated that fine-grained incremental compilation is a relevant technique when implementing powerful debuggers in incremental programming environments. The quality of code produced by the incremental compiler approaches that required for production use. The algorithms involved in incremental compilation are not very complicated, but they require information that is easily available only in an integrated system, where editor, compiler, linker, debugger, and program data-base are well integrated into a single system. The extra information that has to be kept around, like the cross-reference data-base, can be used for multiple purposes, which makes total system economics favorable.

ACKNOWLEDGMENTS

Jerker Wilander, who wrote the PATHCAL system, gave me the necessary inspiration at the beginning of this work. Thanks also go to Anders Sundquist who wrote the primitives that manipulate the memory of the target computer via DecNet, and who succeeded in overcoming the RSX11M operating system (on the target). Thanks, Bengt Lennartsson, Anders Haraldsson, Jim Goodwin, and Erik Sandewall for helpful comments on my work, and on this paper.

REFERENCES

1. C. Albaerga, A. Brown, G. Leeman, M. Mikelsons, and M. Wegman, *A Program Development Tool, Symposium on Principles of Programming Languages* (Jan 1981).
2. P. Brown, *Throw-away Compiling, Software—Practice and Experience* (July–Aug 1976).
3. J. Earley and P. Caizergues, *A Method for Incrementally Compiling Languages with Nested Statement Structure, Commun. ACM* 15, 12 (Dec 1972).
4. *ECL Programmers Manual,* Center for Research in Computing Technology, Harvard University, Cambridge, Mass., Dec 1974.
5. R. Ellison, *Private Communication,* The GANDALF project, Carnegie-Mellon University, June 17, 1982.
6. P. Fritzson, *Fine-grained Incremental Compilation for PASCAL-like Languages,* LiTH-MAT-R-82-15, Software Systems Research Center, Linköping University, July 1982.
7. J. Goodwin, *Private Communication,* Software Systems Research Center, Linköping University, Nov 1982.
8. J. Hennessy, *Symbolic Debugging of Optimized Code, ACM Trans. Programming Languages and Systems* 4, 3 (July 1982).
9. K. Jensen and N. Wirth, *PASCAL User Manual and Report,* Lecture Notes in Computer Science, Springer, Berlin, 1977, vol. 18.
10. S. C. Johnsson, *A Tour Through the Portable C Compiler* from *Unix Programmers Manual,* 7th ed, vol 2B, Jan 1979.
11. M. Kahrs, *Implementation of an Interactive Programming System, SIGPLAN Notices* 14, 8 (Aug 1979).
12. R. Medina-Mora and P. Feiler, *An Incremental Programming Environment, IEEE Trans. Software Engineering* SE-7, (Sept 1981).
13. J. G. Mitchell, *The Design and Construction of Flexible and Efficient Interactive Programming Systems.* Ph.D. thesis, Carnegie Mellon University. Reprinted by Garland, New York, 1979.
14. W. Rishel, *Incremental Compilers, Datamation* (Jan 1970).
15. R. Sethi, *Control Flow Aspects of Semantics Directed Compiling, Proc. SIGPLAN '82 Symposium on Compiler Construction, Boston June 23–25, 1982;* in SIGPLAN Notices, 17, 6 (June 1982).
16. T. Teitelbaum and T. Reps, *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment Commun. ACM* 24, 9 (Sept 1981).
17. W. Teitelman, *Interlisp Reference Manual.,* Xerox Palo Alto Research Center.
18. J. Wilander, *An Interactive Programming System for* PASCAL, BIT 20, 163–174 (1980).