Debugging Optimized Code Without Being Misled

Max Copperman

UCSC-CRL-93-21
June 11, 1993

Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064
max@cse.ucsc.edu

ABSTRACT

Optimizing compilers produce code that impedes source-level debugging. Examples are given in which optimization changes the behavior of a program even when the optimizer is correct, showing that in some circumstances it is not possible to completely debug an unoptimized version of a program. Source-level debuggers designed for unoptimized code may mislead the debugger user when invoked on optimized code.

One situation that can mislead the user is a mismatch between where the user expects a breakpoint to be located and the breakpoint's actual location. A mismatch may occur due to statement reordering or discontiguous code generated from a statement. This work describes a mapping between statements and breakpoint locations that ameliorates this problem. The mapping enables debugger behavior on optimized code that approximates debugger behavior on unoptimized code closely enough that the user need not make severe changes in debugging strategies.

Another situation that can mislead the user is when optimization has caused the value of a variable to be *noncurrent*—to differ from the value that would be predicted by a close reading of the source code. This work presents a method of determining when this has occurred, proves the method correct, and shows how a debugger can describe the relevant effects of optimization. The determination method is more general than previously published methods, handling global optimization, flow graph transformations, and not being tightly coupled to optimizations performed by a particular compiler. The information a compiler must make available to the debugger for this task is also described.

A third situation that can mislead the user is when optimization has eliminated information in the run-time (procedure activation) stack that the debugger uses (on some architectures) to provide a call stack trace. This work gives several methods of providing the expected stack trace when the run-time stack does not contain this information.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging — *debugging aids*; D.2.6 [Software Engineering]: Programming Environments; D.3.4 [Programming Languages]: Processors — *code generation, compilers, optimization*
General Terms: Algorithms, Languages
Additional Keywords and Phrases: debugging, compiler optimization, reaching definitions, noncurrent variables, call stack trace, run-time stack

Acknowledgements

I thank my family, who have given my life balance during this enterprise, and my advisor, Charlie McDowell, whose help at times acted as a lubricant and at times as a hammer.

Contents

List of Tables

List of Figures

Part I

Introduction and Background Material

1 Introduction

It is possible for a source-level debugger to provide expected or truthful behavior in the presence of a large class of sequential optimizations.

It is important because the alternatives, debugging at the assembly level, disabling optimization, or being misled by the debugger, add to the cost of software production.

I show how to provide expected behavior relative to providing a call-stack trace in the presence of a stack-frame optimization. I show how to provide expected behavior relative to breakpoint location determination when possible, and I show how to provide truthful behavior relative to breakpoint location determination that approximates expected behavior closely enough that debugger users can use standard debugging strategies. I show how to determine when expected behavior relative to data value reporting is possible, and how to provide expected behavior when possible and truthful behavior otherwise, in the presence of a large class of sequential optimizations. Previous techniques have worked in the presence of local optimization and have been tightly coupled to the particular optimizations that have been performed. My technique works in the presence of local and global optimization, and is not specific to particular optimizations. It also works in the presence of optimizations that transform the optimized program's flow graph.

Debugging is a major component of the software life cycle. The cost of the debugging component has dropped in recent years for common platforms because of the availability of source-level debuggers with good interfaces and many features. However, it is still typical of compilers that enabling the production of information designed to allow source-level debugging has the effect of automatically disabling optimization.

The problem is that the compiler-debugger interface that allows source-level debugging of unoptimized code is not rich enough to allow accurate source-level debugging of optimized code. In fact, many of the compilers that produce source-level debugging information for optimized code modify their optimizations to minimize the inaccurate reporting of information by the debugger. In general, it is impossible for a source-level debugger to provide the user with accurate information about a program being debugged if the program is optimized and the debugging information is restricted to the typical compiler-debugger interface.

There are three myths in the software industry that have misdirected attention away from this problem. One is that debugging is done in the development phase of the software life cycle and is not (or is rarely) done subsequently. The second is that optimization is not important until the production phase. The third is that if the behavior of an optimized version of a program differs from the behavior of an unoptimized version of the program run on the same input (in particular, if one exhibits a bug and the other does not), the error must be in the compiler.

Combined, these myths suggest that an unoptimized version of the software can be developed and debugged, and successfully recompiled with optimization enabled once the development phase is completed. They are myths, however.

- All large software systems, including products that have been on the market for years, have bugs. Bugs are rife in the production phase. Debugging is a large piece of the maintenance component of the software life cycle.
- Optimization is often routine during development. For some applications, optimization is necessary. These include applications that must run in limited-resource environments, such as embedded applications, real-time systems, or MS-DOS programs,

and applications that require large amounts of time or space as many scientific programs do—some of them run for hours when automatically parallelized; if run without parallelization they would run for days.

- Program bugs can cause the behavior of a correctly compiled optimized version of a program to differ from the behavior of a correctly compiled unoptimized version of the program. Optimization can mask or unmask an existing bug.

These three facts underscore the fact that it is sometimes necessary to debug optimized code. If a program is to be optimized, it is also cheaper to debug optimized code, avoiding recompilation steps, version (optimized versus unoptimized) problems, and entirely avoiding bugs that simply don't show up in the optimized version. Debuggers are also used for performance enhancement; clearly this must be done on the optimized version of the program.

It is common for optimized code to be debugged at the assembly level either because source-level debugging information is unavailable or because the information provided by a source-level debugger is not trustworthy. This is typically slower than source-level debugging. It is also common for optimized code to be debugged at the source level and for programmer time to be wasted because the information provided by the debugger is inaccurate.

This work describes methods of enriching the compiler-debugger interface to allow a debugger to provide accurate information about sequential optimized code. Part I shows how optimization can change the behavior of a program, describes the problems that optimization introduces into the debugging process, and reviews previous and current research in source-level debugging of optimized code. Part II describes several possible solutions to the problem of providing an accurate call-stack trace when the frame pointers normally present in the run-time stack have been optimized away. This problem is particular to architectures on which there is a stack pointer that is modified in the process of evaluating expressions. One of the solutions has been implemented and results are described. Part III describes a general breakpoint model and a general solution to the *currency determination* problem; the problem of determining whether optimization has affected the value of a variable at a breakpoint in such a way that the value may mislead the user.

2 Optimization May Change the Behavior of a Program

A program compiled with optimization enabled may behave differently from the same program compiled with optimization disabled—that is, when optimization is turned off, the bug may go away. Optimizations are correctness-preserving transformations which, if the compiler is correct, will not change the behavior of a correct program. However, a program that is being debugged is certifiably not a correct program, and correctness-preserving transformations are not guaranteed to preserve the behavior of an incorrect program.

It is a misconception that if optimization changes the behavior of a program, the compiler must be incorrect.[1] There are two circumstances in which correct optimization may change the behavior of a program.

[1]If the compiler is incorrect, there are three options: get a different compiler, get the broken compiler fixed, or work around the bug. In practice the first two options may not be viable. The third option requires the programmer to find the code that causes the compiler bug to show up and replace it with semantically equivalent code on which the compiler functions correctly. The programmer still has to debug the (incorrectly) optimized code! Even if the choice is made to have the compiler fixed, the programmer typically has to debug the optimized code enough to convince the compiler vendor that it is a compiler bug.

```
void uses_uninitialized_variable() {
    int x,y;
    return y;
    }
```

Figure 2.1: Optimization Changes Program Behavior: Example 1

- Loose semantics:
  A language may contain constructs whose semantics allow multiple correct translations with distinct behaviors. Most common general purpose programming languages do contain such constructs. The most commonly known area of "loose semantics" is evaluation order. A correct optimized translation of a program containing code with loose semantics may have different behavior from a correct unoptimized translation of that program.
- Buggy programs:
  A correct optimized translation of a program containing a bug may have different behavior from a correct unoptimized translation of that program. This is a commonly overlooked case that is important because a program that is being debugged is known to have bugs.

It is common for even experienced software engineers to be surprised at the fact that a program can behave one way when not optimized and a different way when optimized *when it has been compiled with a correct compiler*. Figure 2.1 is almost a parody of a very common type of bug that can cause such a behavior change. The bug is using a variable that is not properly initialized. When `uses_uninitialized_variable` is called, a stack frame is allocated for it, within which `x` and `y` are located. It is possible that on some call to `uses_uninitialized_variable`, the stack was left in a state in which `0` was in the location allocated to `x` and `1` was in the location allocated to `y`, in which case `1` is returned. Because `x` is not used, its storage may be optimized away. In the optimized version, `y` is located where `x` falls in the unoptimized version—and `0` is returned.

The bug in Figure 2.2 (writing one byte past the end of `b`) has an effect when the program is not optimized. It is benign ("goes away") when data fetches are optimized by aligning data structures on 4 byte boundaries. If each data object is aligned to a four byte boundary, there will be two bytes of padding between the end of array `b` and character `c`, and the bug will have no effect on program behavior. If data objects are not aligned, there will be no padding between `b` and `c`; `c` will be overwritten. Note that program misbehavior, which is the external evidence of the bug, could occur when the program is *not* optimized and go away when the program *is* optimized by changing the sense of the conditional — that is, by adding another bug. Regardless of which version exhibits the symptom of the bug, the behavior changes depending on the presence or absence of optimization.

Because optimization can change the behavior of a program, it is necessary, upon occasion, to either debug optimized code or never optimize the code.

2.1 Approaches to the Problem

General approaches to the problem have been:

- to restrict the optimizations performed by the compiler to those that do not provoke the problem ([WS78], [ZJ90]),

```
int i;
char b[10], c;

void walk_on_c() {
    c = getchar();
    for (i=0; i<=10; i++) {
        b[i] = '\0';
        }
    if (c == '\0') {
         program misbehaves
        }
    }
```

Figure 2.2: Optimization Changes Program Behavior: Example 2

- to restrict the capabilities of the debugger to those that do not exhibit the problem ([WS78], [Gup90], [ZJ90]),
- to recompile, without optimization, during an interactive debugging session, the region of code that is to be debugged ([FM80], [ZJ90]), and
- to have the compiler provide information about the optimizations that it has performed and to have the debugger use that information to provide appropriate behavior ([WS78], [Hen82], [Zel84], [CMR88], [ZJ90], [Gup90], [PS91], [Coh91], [CM91b], [BHS92], [PS92], [Cop92], [BW93], [AG93a]).

A larger problem is lowering the cost of debugging production quality software. Much if not most production quality software produced in this country is heavily optimized, and the first approach, restricting optimization, would result in compilers that would not get used; their use would degrade the quality of the software. The second approach, restricting the capabilities of the debugger, is clearly undesirable, though possibly preferable to being misled by the debugger. The third approach, recompiling without optimization while debugging, requires a software engineering environment that provides incremental compilation. Such environments are not in general use and even should they become commonplace, the approach is problematic because optimization may change the behavior of the program.

This work follows the fourth approach, using compile-time information about optimization to provide appropriate debugger behavior. Some of the previous work that has taken this approach has resulted in compiler/debugger pairs that are able to provide acceptable behavior when debugging optimized code because the debuggers have been specialized to handle the particular optimizations performed by the compiler. Because much of the industry allows compilers and debuggers to be mixed and matched, solutions that do not require the compilers and debuggers to be tightly coupled are preferable. While not without restrictions, this work is general in that the debugger need not be specialized to a particular set of optimizations.

This work is applicable in the presence of any sequential optimizations that either do not modify the flow graph of the program or modify the flow graph in a constrained manner. Blocks may be added, deleted, coalesced, or copied; edges may be deleted, but control flow may not be radically changed. The constraints are described in detail in Section 17.3. This work applies in the presence of

- local common subexpression elimination
- global common subexpression elimination
- constant and copy propagation
- constant folding
- dead code elimination
- dead store elimination
- cross-jumping
- local instruction scheduling
- global instruction scheduling
- strength reductions
- code hoisting
- partial redundancy elimination
- other code motion
- induction-variable elimination
- loop unrolling
- inlining (procedure integration)

as well as any other optimizations that observe the constraints. As an example of an optimization that does not observe the constraints, it does not apply to the portion of a program that has had loops interchanged.

3 Background

Automatic translation of high-level language programs into machine language was once an artificial intelligence problem. By the late 1960's or early 1970's, there was a well-founded body of scanning, parsing, and type-checking theory and a considerable number of effective code-generation techniques. The research in the area of high-level programming languages has been moving the languages toward the problem domains, providing language constructs that allow programs to describe problems and solutions in ways that are more understandable by the programmer. This moves the languages further from the machine-level description of the computation. The recent research in the area of translation of high-level programming languages to machine languages has been in optimization, which moves the code generated from the program toward the machine-level description of the computation. The "semantic gap", the gap between the source program and the generated code, has been growing.

This has consequences for the process of debugging. Compilation is a mapping from source code to machine code. Debugging is the art of determining from a program's "misbehavior" the source constructs responsible for the misbehavior. Misbehavior is a difference in behavior from intended behavior, and is discernible in the effects of the machine code. This involves following the mapping backward from machine code to source code. The less straightforward the mapping, the more difficult this process becomes.

Assembly-level debuggers have no difficulty with optimized code; they deal only in machine entities (addresses and instructions) and do no association of these entities with source entities (symbol names and source constructs). In recent years, source-level debuggers have become available. Source-level debuggers must be able to associate machine entities and source entities, as they translate user's requests in terms of source entities into actions in terms of machine entities. Programmers can debug code faster with a source-level debugger because it automates much of the reverse mapping process. This is done by having the compiler create a machine-code to source-code mapping, including information about the source code in the object module; information such as the type and address of data objects and the address of the start of the code generated from each source line.

The machine-code to source-code mappings in general use today are sufficient to describe the relatively straightforward mapping from unoptimized code to source code, but they are not sufficient to describe the (potentially convoluted) mapping from optimized code to source code.

4 The Effects of Optimization

Unoptimized code has several characteristics that aid in the ability to do source-level debugging. In unoptimized code

1. statements are translated to a contiguous set of instructions.[2]
2. In straight-line code, the set of instructions for a statement S immediately precedes the set of instructions for the statement following S. This extends in a natural way to statements that involve branching such that if execution is halted at the code for S, the code for statements appearing in the source code before S will have been executed and the code for statements appearing in the source code after S will not have been executed.[3]
3. Each variable has exactly one location.
4. The value in a variable's location matches the that value the source code would lead one to expect the variable to have, at all statement boundaries.

Optimized code does not necessarily have any of these characteristics.

5 The Problems Optimization Causes Source-Level Debuggers

The following low-level debugger capabilities are made difficult by optimization:

1. the association of source code locations with machine code locations, and in particular,
   (a) trap location reporting, and
   (b) breakpoint location determination
2. execution context reporting (for example, providing a call-stack trace)
3. data value reporting, with distinct subproblems
   (a) data location determination
   (b) data currency determination
4. data modification

[2]This is not the case for all constructs. For example, loops often involve two discontiguous sets of instructions, a test and a jump, separated by the body of the loop. Case or switch statements may involve multiple discontiguous sets of instructions for correctly addressing the target. The discontiguity is present in the source code, but it still causes problems in trap location reporting and breakpoint location for these constructs. As described below, these problems are endemic in optimized code.

[3]For source statements in a loop, we can distinguish executions of the statements by subscripting them with the loop iteration value. If execution is halted at S_i (at S in the i^{th} iteration of the loop), $\mathit{Successor(S)}_{i-1}$ will have been executed and $\mathit{Successor(S)}_i$ will not have been executed.

Table 5.1: Line Table

Code Address	Source Line Number
⋮	⋮
3F6	17
3FC	18
404	19
⋮	⋮

5. code modification

At a higher level, how the debugger presents information about the program to the user becomes a more difficult problem in the presence of optimization. Zellweger [Zel84] introduced terms for two methods of removing or ameliorating the confusion introduced into the debugging process by optimization. The preferred method is to have the debugger responses to queries and commands on an optimized version of a program be identical to its responses to the same set of queries and commands on an unoptimized version of the program. This is known as providing *expected* behavior. It may not always be possible to provide expected behavior, so the next best thing is to provide *truthful* behavior, in which the debugger avoids misleading the user, either by describing in some fashion the optimizations that have occurred or by warning the user that it cannot execute a command or give a correct answer to a query.

If a debugger were always able to provide expected behavior, optimization would present no additional interface problems for a debugger. Truthful behavior has attendant interface problems because it requires that the debugger expose the effects of optimization in a way that is meaningful to a possibly naive user. Consider the problem of telling the user in source terms where a program has halted when loops may have been unrolled, fused, and interchanged. For the most part, these interface issues are beyond the scope of this work. Part II concentrates on providing expected behavior in the presence of a particular optimization. Part III shows how to determine when expected behavior is possible (relative to data value reporting) in the presence of a large class of optimizations. Attention is given to producing the information that the debugger needs to expose the effects of optimization when expected behavior is not possible, but little attention is given to the form of the presentation of that information.

5.1 Associating Source Code Locations with Machine Code Locations

Source-level debuggers commonly use a *line table* to associate source code locations with machine code locations. A line table is a table of $\langle$*address*, *line number*$\rangle$ pairs in which *address* is the address of the first instruction generated from the source line at *line number*.

In unoptimized code, all instructions between one code address entry and the next are generated from the referenced source line. In addition, all entries (code addresses and line numbers) are monotonically increasing. If program execution were halted before execution of the instruction at code address entry A associated with source line S, all instructions generated from source lines previous to S and no instructions generated from source lines subsequent to S would have been executed.

```
foo();                call foo                 load r1, q
*p = *q;    =>        load r1, q      =>       load r1, [r1]
                      load r1, [r1]            call foo
                      load r2, p               load r2, p
                      store [r2], r1           store [r2], r1
```

Figure 5.1: A Statement May Have Discontiguous Generated Code

5.2 Trap Location Reporting

When a program terminates due to a trap, the debugger must be able to report the statement in the source program that generated the instruction causing the trap. In the absence of optimization, the line table is sufficient. The debugger determines into which code address set the program counter falls and returns the associated source line number.

In the presence of optimization, the line table is not sufficient. Its use in trap location reporting crucially depends on the instructions generated from a statement being contiguous. Consider the source code, machine code, and transformed machine code shown in Figure 5.1. There is no way to correctly report a trap caused by `p` containing trash and a trap caused by `q` containing trash with a single entry in a line table (assuming that the call to `foo` also has an entry).

This problem is addressed is Section 15.1. It is also addressed in some of the more modern symbol table formats. Examples include the symbol table put out by the MIPS compiler for use by dbx and the symbol table put out by the Convex compiler for use by the Convex debugger Cxdb [Str91], [BHS92].

5.3 Breakpoint Location Determination

When the user requests a breakpoint at a source line L, the debugger must be able to set the breakpoint at an appropriate address. In the absence of optimization, the line table is sufficient. By definition, the code address entry whose corresponding source line entry is L is an appropriate address: all previous lines and no subsequent lines will have been executed when the program breaks.

In the presence of optimization, the line table is again insufficient. Its use in breakpoint location determination depends not only on the instructions generated from a statement being contiguous, but also on the statements being executed in the order they appear in the source code. Optimization can cause source statements to be reordered in their entirety. Reordering of source statements as shown in Figure 5.2 presents a different sort of problem from trap location reporting. In the case of trap location reporting, expected behavior could be provided if the necessary information were available. In this particular example of breakpoint location determination, the line table may contain information about where the code for each statement resides, but it is not clear what to do with the information. Assume the user requests a breakpoint at line 2. If a breakpoint is set before the code for statement 2, the user may be confused by the fact that statement 1 has not yet been executed. If a breakpoint is set after the code for statement 1, the user may be confused by the fact that statement 2 has already been executed.

Additionally, if the instructions for a statement are not contiguous, the first instruction generated from a statement may not be an appropriate choice for the breakpoint location

```
1  a = b;          2  c = d;
2  c = d;    =>    1  a = b;
3  e = f;          3  e = f;
```

Figure 5.2: Statements May Be Reordered

for the statement. The first instruction generated from a statement S may have been moved far from the rest of the instructions generated for S, across the code for other statements, it may have no user-visible effect on program state (it may be part of an address computation, for example). The rest of the instructions for the statement, or at least the ones that result in user-visible effects (such as branches and stores), may not have been moved at all — even if they have been moved, one of them may be a better candidate for the breakpoint location than the first instruction.

This problem is addressed is Section 15. It is also addressed by Zellweger [Zel84], Coutant et al [CMR88], Zurawski and Johnson [ZJ90], Streepy [Str91] and Brooks et al [BHS92], and Wismüller [BW93]. Different approaches allow different debugger capabilities. None of the approaches in the literature, including mine, provide a complete solution.

5.4 Execution Context Reporting

When the debugger gains control of a halted program, it provides the user with information about where the program halted. Most debuggers by default provide source file and line information. From this information, in unoptimized code, the user can determine a great deal about what source code has been executed and what has not. In optimized code, the user can determine considerably less. Worse, the user may be led to believe that code has been executed when it has not, or vice-versa. Simple code motion is enough to cause the latter problem. Loop transformations can compound the problem. This problem is addressed to some extent by Streepy [Str91] and Brooks et al [BHS92].

The user can request more context in the form of a call-stack trace.[4] Two optimizations, inlining and the stack-frame pointer optimization discussed in Part II of this work, can cause a call-stack trace to be inaccurate and misleading. Zellweger [Zel84] shows how to provide expected behavior in the presence of inlining, and Part II shows how to provide expected behavior in the presence of the stack-frame pointer optimization.

5.5 Data Value Reporting

Source-level debuggers commonly use symbol table entries to associate source code data objects with memory locations. Symbol table entries differ from object module format to object module format, but generally contain the name, type, size, and location of a data object. Some symbol table formats require the location information to reference memory (registers are not allowed). Many formats allow only a single location to be associated with a data object, although allowing multiple locations is more common in recent years.

[4]The call-stack trace is also known as a stack trace, stack dump, procedure traceback, or backtrace.

Data Location Determination

In optimized code, a data object may reside in more than one location, or nowhere at all, over the duration of its lifetime. A debugger that retrieves an object's value from the location given in the symbol table entry when the object actually resides elsewhere (or nowhere) may give the user incorrect information.

This problem is addressed by a number of modern symbol table formats, including Microtec Research, Inc.'s format for their Xray debugger [Wan91], the DWARF Debugging Information Format [Sil92], the symbol table put out by a prototype modification of Hewlett Packard's C compiler for the HP9000 series 800 for use by a prototype modification of the debugger for that machine [CMR88], and the symbol table put out by the Convex compiler for use by Cxdb [Str91]. Although the symbol table formats are general, the information stored in them is usually derived from live range information computed for use by the optimizer. This is not optimal for use by the debugger, for reasons described in Section 26. Adl-Tabatabai and Gross [AG93a] show how data-flow analysis can be used to determine optimal variable location range information. Wismüller [BW93] also approaches the problem using data-flow analysis.

How this problem interacts with my solution to the next problem is discussed in Section 26.

Data Currency Determination

Even if a data object's location is correctly determined, that location may not have the value the user expects. For example:

```
Unoptimized                           Optimized
a = expression1;                      a = expression1;
...     /* a is used.     */          ...
a = expression2;
foo(a); /* Last use of a. */          foo(expression2);
```

The user may break at the call to `foo` and ask for the value of `a` in order to find out what was passed to `foo`. If the dead store into `a` has been eliminated, the value in the location associated with `a` may mislead the user.

This problem is addressed for local optimizations only (specifically, register promotion combined with local instruction scheduling) by Coutant et al [CMR88], Adl-Tabatabai and Gross [AG93a], and Streepy [Str91] and Brooks et al [BHS92]. It is also addressed for local and global optimization by Hennessy [Hen82], Wismüller and others [BW93], and by Part III of this work.

5.6 Data Modification

The user may want to modify a data object's value while in the debugger and continue execution of the program using the modified value to modify the program's behavior. Optimization can defeat the process. Consider:

```
#define DEBUG 0
a = DEBUG;
...        /* No assignments to a. */
if (a)
    print();
```

Since the value of `a` is known to be 0 at compile time, the conditional is known to always be false and can be eliminated, along with the call to `print`. If the user sets `a` to 1, it will not cause `print` to be called.

This problem has not been addressed. It is generally considered too difficult, but the data structure developed by Pollock and Soffa for incremental global optimization [PS88], [PS92] should be applicable.

5.7 Code Modification

The user may want to modify existing code or execute arbitrary code while in the debugger. Debuggers are far from uniform in the manner in which they provide this sort of facility. Some do not allow it at all. Others allow anything that can be done by writing to memory or to registers. Some allow any existing procedure to be called, while others provide facilities specifically for patching code, either in assembly or high level languages. In the presence of optimization, the results may be unexpected. In the following example, optimization causes `v` to contain the value it receives from `foo()` rather than the value `5` at *breakpoint*. If, from within the debugger while the program is stopped at *breakpoint*, the user calls a routine that uses `v`, the routine may perform in an unexpected manner.

```
Unoptimized                          Optimized
...                                  ...
v = foo();                           v = foo();
...                                  ...
v = 5;
b = v; /* Last use of v. */          b = 5;
breakpoint—call routine(v) from the debugger
```

Providing expected behavior or useful truthful behavior in the presence of code modification has not been addressed. It is generally considered too difficult.

Now that we have reviewed the problems, let us review the research that has been done in the area.

6 A Survey of Related Work

This section surveys the work that has been done on providing a mapping from machine code to source code to support source-level debuggers that can effectively assist in debugging optimized code. Summaries of published papers are presented in approximately chronological order. Discussions of each paper are headed by the title of the paper.

"Design of the FDS interactive debugging system"

In July of 1978, Warren and Schlaeppi produced an IBM research report [WS78] the intended debugging system for the Research Division's Firmware Development System (FDS). The problem that they attacked was to design an interactive source level debugging system for microcode that would allow full optimization with attendant debugging problems or full debugging with constraints on optimization, without recompilation or modification of the source program. While their system was designed for microcode as opposed to general software, the problems inherent in debugging optimized code are identical, thus their approaches are valid for a general discussion of debugging optimized code.

This paper opened the discussion of how to do source-level debugging of optimized code. It contains a discussion of information needed from the compiler to implement a debugger that provides messages about the effect of optimizations on source code—what has since been termed *truthful behavior.*

"Symbolic Debugging of Optimized Code"

Hennessy published the first journal paper [Hen82] on the topic, and it has been influential. His paper is concerned with currency determination and expected value recovery. When a variable is not current, *expected value recovery* is the act of computing the value the variable would have had if it were current.

Hennessy's work sets the context for future work on debugging optimized code. He introduces the notion that the debugger should, when possible, hide the effects of optimization from the programmer.[5] He introduces the notion of noncurrent and endangered variables, and the notion of recovery of expected values for noncurrent variables.

He describes a model of code generation and local optimization for which he can do currency determination and (often) recovery, augmenting a code generation DAG with information used by his currency determination algorithms. He also considers limited global optimization, with weaker results.

Technical correspondence about Hennessy's work has appeared twice. "A note on Hennessy's *Symbolic Debugging of Optimized Code*", by Wall, Srivastava, and Templin [WST85] corrects an error in the algorithm used to determine the currency of variables, using Hennessy's data structures. "A further note on Hennessy's *Symbolic Debugging of Optimized Code*", by Copperman and McDowell [CM93] points out that his technique assumes that after optimization, programs have only one store to a variable within any basic block. This was a reasonable assumption in 1982 but is a restrictive assumption today.

[5]This approach, when successful, has subsequently been termed providing *expected behavior*.

"An Incremental Programming Environment"

In 1980 Feiler and Medina-Mora produced a report at Carnegie-Mellon University about one part of a software engineering environment, the Incremental Programming Environment (IPE), in which editing, compiling, linking, and debugging are performed on a common representation of a program by integrated tools.

Compilation is incremental on a procedure-by-procedure basis. Debugging is accomplished by modifying the code and (automatically) recompiling the modified procedure with optimization disabled. The language definition is extended to include debugging constructs, and these are programmed in the same manner as other language constructs.

Incremental recompilation has the advantage that debugging can be done without any of the problems discussed in Section 5. The disadvantage is that no debugging support is provided for the situation in which disabling optimization changes the behavior of the program.

"The Debugging of Optimized Code"

This is an unpublished draft by Teeple and Anderson from March 1980. It describes a problem with determining breakpoint locations in the presence of cross-jumping. While their proposed solution was not correct, it contained the genesis of Zellweger's complete and correct algorithm [Zel84], and appears to have contained the genesis of her *invisible breakpoints* and *path determiners*.

"A Systematic Approach to Advanced Debugging through Incremental Compilation"

Fritzson [Fri83] investigated using incremental compilation to supply advanced debugging features. He was not focussed on debugging optimized code, but his system has the advantages and disadvantages of any incremental compilation system with regard to debugging optimized code. His incremental compiler was based on the portable C compiler, thus optimizations were local to a statement. This allowed him to incrementally recompile at statement granularity without needing an optimization history like that of Pollock and Soffa [PS88], [PS92].

"Interactive Debug Requirements"

Seidner and Tindall published IBM's market requirements statement for an interactive debugger [ST83]. It catalogs what a good debugger (as seen from 1983) should not be without, with a good rationale for each suggested feature. It is interesting how many debuggers today are without some of these features. Source-level debugging of optimized code was considered a requirement for an interactive debugger. The paper lists the problems optimization causes for a debugger and the behavior that solutions to the problems should support.

"An Interactive High-Level Debugger for Control-Flow Optimized Programs"

Zellweger [Zel84] produced the first dissertation on debugging optimized code. The first part of her thesis provides a coherent discussion of terminology for debugging optimized code and the related issues. Terminology she introduced includes:

- expected versus truthful behavior
- semantic and syntactic breakpoints
- invisible breakpoints
- path determiners

Expected Versus Truthful Behavior Zellweger defines two properties of debuggers, neither of which holds for commonly used debuggers when applied to optimized code. A debugger that provides *expected behavior* always responds exactly as it would for an unoptimized version of the same program. The effects of optimization are hidden from the user. The debugger must create a view of the (optimized) machine state that conforms to the (unoptimized) source-level program state.

A debugger that provides *truthful behavior* avoids misleading the user. For a given query or attempted action,

- it may do nothing, stating that any response would be incorrect due to optimization,
- it may provide a response but warn that the response may be incorrect due to optimization, or
- its response may include a description of the relevant effects of optimization.

The effects of optimization are described to the user. The user now has the task of creating a view that conforms to the (unoptimized) source-level program state. The debugger may still be able to do much of the machine-state to source-level program state translation.

A debugger that provides neither expected not truthful behavior is likely to give incorrect and/or confusing responses to queries about an optimized program. Even truthful behavior may be confusing to a user unfamiliar with the possible effects of optimization. Expected behavior is clearly the superior choice. However, expected behavior may be impossible to provide. For example, if induction variable elimination has removed all references to a variable V from the machine code for a loop, and the compiler has left no information about how to reconstruct the variable's value, the debugger cannot provide its expected value if asked for it within the loop. The ability to provide expected behavior depends on the information provided by the compiler. In some cases it is impractical to provide the information necessary to allow a debugger to provide expected behavior. In general, the investigation into source-level debugging of optimized code involves determining how to provide expected behavior, determining when that is impossible or impractical, and if impossible or impractical, determining what sort of truthful behavior is appropriate.

Semantic and Syntactic Breakpoints A *semantic* breakpoint for a statement is located at the machine code generated from the statement. It is called semantic because the breakpoint is associated with where the statement "happens". A *syntactic* breakpoint location is chosen according to the syntactic positional relationship of source statements. In unoptimized code, semantic and syntactic breakpoint locations are identical. However, if a code motion optimization has been performed such that the code for a statement S has been moved relative to other statements, a semantic breakpoint will be placed wherever the code for S ends up, while a syntactic breakpoint will be placed wherever the code for S started out. Given a sequence of statements S_1, S_2, ..., S_n, for two statements S_i and S_j, if $i < j$ then the syntactic breakpoint for S_i will never follow the syntactic breakpoint for

S_j.[6] This condition does not hold for semantic breakpoints. Semantic breakpoints require knowing where the code for a statement is moved to, and problems arise if the instructions generated from a statement are not contiguous in the final machine code. Zellweger discusses five possible refinements of the semantic breakpoint definition for noncontiguous code generated from a single statement S. The breakpoint could be set:

1. on the first instruction generated from S,
2. on the first instruction of each contiguous set of instructions generated from S,
3. on the first instruction generated from S that corresponds to any user-visible change in execution state,
4. on the first instruction generated from S that corresponds to any change to a user-visible variable,
5. on the first instruction generated from S that corresponds to the "core semantic effect" of S, where the core semantic effect would ideally by defined by the language.

Invisible Breakpoints An *invisible breakpoint* is a breakpoint set by the debugger, at which the debugger takes some action and continues execution of the program without giving control to the user.

Path Determiners A *path determiner* is Zellweger's technique for using an invisible breakpoint to determine what execution path is taken.

Control Flow Optimizations The main body of Zellweger's work describes algorithms for providing expected behavior (most of the time) in the presence of cross-jumping and procedure inlining. In the presence of inlining, extra work is required to display a call-stack trace that alerts the user that an inlined routine is active. This is the only research in the literature that deals with a problem similar to the one I address in Part II. Zellweger has successfully implemented her algorithms in the Cedar programming environment at Xerox PARC.

"DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code"

At Hewlett Packard, Coutant, Meloy, and Ruscetta [CMR88] modified a compiler and debugger to produce a prototype called DOC that can debug the optimized code the compiler produces. The debugger was designed to handle global register allocation, induction variable elimination, constant and copy propagation, and delay slot scheduling.

A major goal was to inform the debugger of a variable's location at a given point in a program. The compiler provides the debugger with live-range information, including the location(s) a variable is mapped to within a live range. It also provides information about whether a variable is current, which can be done inexpensively because no global code motion is done. Eliminated induction variables are handled by storing a recovery function, which is a function from the value of a named compiler temporary to the value of the eliminated user variable which the debugger can evaluate.

[6] The syntactic breakpoint locations for S_i and S_j may be in the same location. This will occur if both statements have been moved.

“Incremental Global Reoptimization of Programs ...”

In two articles [PS88], [PS92], Pollock and Soffa explore the potential of incremental optimization. The earlier paper focusses on incrementally reoptimizing when an optimization has made it impossible to satisfy a debugging request. The later paper focusses on incrementally reoptimizing when a program is edited.

They develop a data structure in which an optimization history is recorded, and use it to incrementally recompile and reoptimize the program in response to debugging requests or program edits. The optimization history contains information about the extent of an optimization (what regions of code or further optimizations it affects), so that when an optimization must be reversed, any dependent optimizations can also be reversed.

While it has not been investigated in this regard, their data structure should be applicable to the data modification problem—at least to allow truthful behavior. I have not seen any other approaches that might apply to this problem.

“A Unified Approach to the Debugging of Optimised Programs ...”

Shu [Shu89], [Shu91], [Shu92], [ST83], did his dissertation on debugging optimized code. I cite his work for completeness, but I only recently became aware of it and am not familiar with the content. From my limited acquaintance with his work, it appears that he has concentrated on breakpoint mapping and formalizing the effects of optimization in a composable set of rewrite rules.

“Debugging Code Reorganized by a Trace Scheduling Compiler”

Gupta [Gup90] investigated debugging code that has been automatically parallelized by a trace scheduling compiler. To allow the compiler freedom to exploit any parallelism present in the program, “the power of the debugger is compromised”. Monitoring commands must be compiled into the program. The debugger can then enable and disable any of these, and monitor those variables or expressions, at those points in the program (which might be specific loop iterations) that were selected for monitoring at compile time.

“Debugging Optimized Code With Expected Behavior”

At the University of Illinois at Urbana-Champaign, Zurawski and Johnson [ZJ90] modified the standard Smalltalk debugger and an optimizing compiler for Typed Smalltalk. They are always able to provide expected behavior by a combination of compiler-generated information, incremental compilation, and restricting points at which a debugger can take control of a program.

“Source Level Debugging of Automatically Parallelized Code”

Cohn [Coh91] describes a method for providing a sequential view of programs that have been automatically parallelized for distributed memory MIMD machines by transformations that can be decomposed into *thread splitting* and sequential optimizations. He defines functions that the debugger can often use to reconstruct a sequential state that corresponds to the execution state of a halted parallelized program. He defines techniques that can be used when no sequential state corresponds to the current execution state of the program, to minimize the cost of achieving a debuggable state (the worst case involves re-executing the program with synchronization code).

"Debugging Parallelized Code Using Code Liberation Techniques"

Pineo [PS91] describes a different method for providing a sequential view of automatically parallelized programs. Global renaming is used to convert the sequential program to single assignment form. Subsequently, parallelizing transformations can be performed. Finally, the large storage requirements of single assignment code are reduced by reclaiming names that are not useful for parallelization or debugging.

"A New Approach to Debugging Optimized Code ..."

At Convex, Streepy [Str91] and Brooks et al [BHS92] break with the by now conventional view that the debugger should hide the effects of optimization. Their view is that the debugger should aid the user in understanding what is actually happening in the presence of aggressive optimization and parallelization. To this end, they describe an enriched compiler-debugger interface and a debugger user interface that can visually display the effects of optimization. Their debugger CXdb incorporates capabilities first published in Coutant et al [CMR88], handling induction variable elimination and providing live range and variable location information (although in a different format). From a research perspective, their contribution is a finer granularity in the mapping from source entities to machine code addresses. They have eliminated the line table used for mapping source lines to code addresses and replaced it with a mapping between *source units* and sets of address ranges. A source unit can be an expression, a statement, a block, a loop, or a routine. This allows stepping at any of these granularities, as well as accurate trap location reporting. Their method of stepping involves breaking once for each contiguous sequence of instructions generated from a unit. This is a significant departure from the line-by-line stepping strategy employed by many debugger users, which involves breaking once per line. In Section 15 I present a breakpoint model that does not require such a departure. The two models can be combined to provide additional debugging capabilities without losing existing capabilities.

"Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging ..."

Adl-Tabatabai and Gross [AG92], [AG93a], [AG93b] address the data location problem. This problem has been addressed by others [CMR88], [Wan91], [Str91], [BHS92], but typically by providing live-range information to the debugger. This may unnecessarily truncate the range across which a variable is available, because the live-range information computed by the compiler terminates at the last use of a variable, not at the point where it no longer resides in memory or a register. Adl-Tabatabai and Gross show how data-flow analysis can be used to determine the actual availability of a variable. This is discussed further in Section 26 and is addressed by Wismüller [BW93].

"Source-Level Debugging of Optimized Programs Using Data Flow Analysis ..."

Wismüller is currently investigating data location, currency determination and expected value recovery (with Bemmerl [BW92], and with Berger [BW93]). His approach is similar to my approach to currency determination. The scope of his work is quite ambitious. As of this writing [Wis93] his breakpoint model is more general than that presented in Section 15, he incorporates recovery into his currency determination technique, and he addresses the data location problem. He does not address optimizations that modify the shape of the flow graph. If his work comes to successful fruition, it will be a valuable addition to the field.

7 A Partial Survey of Today's Compilers and Debuggers

I have claimed that optimized code causes debuggers to give misleading responses, while noting that there is work being done in the area, both in academia and industry. This section is intended to give a few data points regarding how some commonly available compilers and debuggers respond in the presence of optimization.

Two simple programs were used as probes. The first of these, shown in Figure 7.1, investigates debuggers' call-stack-trace facility in the presence of inlining. The second, shown in Figure 7.2, investigates debuggers' breakpoint and data display facilities in the presence of constant and copy propagation and dead store elimination.[7] It is far from a complete examination of these facilities, but does show some common unexpected behavior.

Clearly, debugger responses depend on the debugger. They also depend on the compiler—on the generated code and the symbol table information. Compilers and debuggers were chosen strictly on the basis of ease of availability (to me).

Compilers and Debuggers Surveyed On personal computers, I have coupled compilers and debuggers from the same vendors:

- Borland's Turbo C/C++ compiler and Borland's Turbo debugger, running on Microsoft's OS/2 on an IBM pc.

[7]Minor editing of these programs was necessary to satisfy all the surveyed compilers. None of the changes had any semantic import.

Figure 7.1: Test Code with Inlining

```
#include<stdio.h>

void a() {
    }

inline int b() {
    a();
    return 1;
    }

int c(int x) {
    int y = b();
    return x+y;
    }

int d() {
    return c(5);
    }

int main(unsigned argc, char **argv) {
    printf("%d\n", d());
    return 1;
    }
```

```
 #include <stdio.h>
 int main(unsigned argc, char **argv) {
 /* int main(argc, argv) unsigned argc; char **argv; {*/
 int w, x, y, z = 0;

 x = atoi(argv[1]);

 printf("%d\n", x);

 x = 5;

 y = x;

 if (y > 2) {
     printf("y > 2");
     }
 else {
     printf("y <= 2");
}

 if (z) {
     printf("z");
}
 else {
     printf("not z");
     }

 }
```

Figure 7.2: Test Code with Constant and Copy Propagation and Dead Store Elimination

- MetaWare's High C compiler and MetaWare's `mdb` debugger, running on MS-DOS with Phar Lap's DOS Extender on an IBM pc.

On the UNIX system I have mixed and matched debuggers and compilers because that is a common mode of operation on that system. All tests were done on Sun Sparcstations.
Compilers:

- Sun's Ansi C compiler `acc`
- The Portable C compiler `cc`
- ATT's Cfront C++ translator `CC`
- The Free Software Foundation's C compiler `gcc`

Debuggers:

- dbx
- The Free Software Foundation's debugger gdb

7.1 Survey Results

Historically, compilers have either produced debugging information or optimized the generated code, but not both.[8] This trend is changing but has not yet disappeared. The following occur when attempting compile to with both -g and -O:

- `cc` gives the message
  `cc: Warning: -O conflicts with -g. -O turned off.`
- For High C, `-g` suppresses inlining (with a warning) and silently suppresses dead-store elimination.
- For acc, `-O` silently suppresses production of debugging information.

The Call-Stack Trace Facility I surveyed the call-stack trace facility in the presence of inlining by setting a breakpoint in routine `a()` of the program shown in Figure 7.1 and requesting a call-stack trace. If expected behavior is provided, when execution is suspended in `a()`, a call-stack trace will show that `a()` was called from `b()`, `b()` was called from `c()`, `c()` was called from `d()`, and `d()` was called from `main()`. The optimization is to expand `b()` inline, so `b()` has no stack frame in the optimized version. A call-stack trace that relies solely on information in the call stack will not provide expected behavior. `b()` will not appear in the call-stack trace, which will suggest that `a()` was called from `c()` rather than from `b()`, and that `b()` is not active.

Inlining is not a part of ANSI Standard C, so `acc` was not among the compilers surveyed. In the other cases, routine `b` was expanded inline by the compiler (or translator/compiler pair, in the case of `CC`/`cc`).

- High C/`mdb`: I did not get to the point of requesting a call-stack trace, because with `-g`, `b` was not expanded inline, and without `-g`, I could not set the breakpoint via source-level debugging (I could not set a breakpoint at `a()` by function name.)
- `CC/cc/dbx`: `b()` does not appear in the call-stack trace.
- `CC/cc/gdb`: `b()` does not appear in the call-stack trace.
- `gcc/dbx`: `b()` does not appear in the call-stack trace.
- `gcc/gdb`: `b()` *does* appear in the call-stack trace. Of the compiler/debugger pairs surveyed, this is the only one that provides expected behavior for its call-stack trace facility.
- Turbo C++/Turbo Debugger Neither `b()` nor `c()` appear in the call-stack trace. the call-stack trace suggests that `a()` was called from `d()`, and `d()` was called from `main()`. The compiler optimizes the stack frame of `d()` as described in Part II of this work, causing the additional unexpected behavior in the call-stack trace.

The Breakpoint and Data Display Facilities I surveyed the breakpoint and data display facilities in the presence of constant and copy propagation and dead store elimination via the program shown in Figure 7.2. The optimized code generated from this program, for each compiler, is shown (in source terms) in Figure 7.3. High C produces code equivalent to that produced by `acc`, `cc`, and `gcc` when `-O` is specified and `-g` is not. As mentioned above, when both are specified, dead-store elimination is suppressed in order to allow debugger

[8] By convention, `-g` is the flag passed to the compiler to cause it to produce debugging information and `-O` is the flag that enables optimization.

```
     High C                 acc, cc, gcc           Turbo C

2    int main(...)  {       int main(...)  {       int main(...)  {
6    x = atoi(argv[1]);     x = atoi(argv[1]);
8    printf("%d\n", x);     printf("%d\n", x);     printf("%d\n", atoi(argv[1]));
10   x = 5;
12   y = x;
15   printf("y > 2");       printf("y > 2");       printf("y > 2");
25   printf("not z");       printf("not z");       printf("not z");
28   }                      }                      }
```

Figure 7.3: Test Code with Constant Propagation, Copy Propagation, and Dead Store Elimination

users to examine the values of variables. This has the advantage of aiding debugger users and the disadvantage that the code runs slower and is not identical to the code produced when `-g` is not specified.

To investigate the breakpoint facility, I set a breakpoint at a line of code that was eliminated by optimization and ran the program. None of the debuggers surveyed provided expected behavior. Truthful behavior requires the debugger to warn the user, at the point that a breakpoint is set at such a line, that the request cannot be honored as given, and to describe the action taken (for example, that the breakpoint was actually set at some other line).

- High C/`mdb`: The breakpoint command was accepted with no suggestion that it was not possible to stop at the referenced line. The breakpoint was reached at a subsequent line (the syntactic breakpoint for the eliminated line), with a message displaying the line number of the line the program actually halted at.
- `acc`/`gdb`: Because `-O` suppresses debugging information, my attempt to set a breakpoint at line 12 produced the message:
  ```
  No line 12 in file "const-prop.c"
  ```
  Attempts to set breakpoints at other lines produced similar messages. Because the suppression of debugging information is silent, this was unexpected and misleading.
- `acc`/`dbx` `dbx` exited immediately upon loading the compiled program, after producing this message:
  ```
  warning:  stab entry unrecognized:  name ,ntype 62, desc 0, value 0
  warning:  stab entry unrecognized:  name ,ntype 38, desc 0, value 0
  warning:  stab entry unrecognized:  name ,ntype 38, desc 0, value 0
  warning:  stab entry unrecognized:  name Xt ; g ; 0 ; V=2.0,ntype 3c,
  desc 0, value 2bf5522d
  dbx:  internal error:  unexpected value 98 at line 2620 in file
  ../common/object.c
  ```
- `cc`: Because `-g` suppresses optimization, I was unable to test the source-level breakpoint facility of any debuggers with code compiled with `cc`. Because the suppression of optimization is made explicit with a warning at compilation time, this was expected and not misleading (though not helpful).

- `gcc/dbx`, `gcc/gdb`: The breakpoint command was accepted with no suggestion that it was not possible to stop at the referenced line. The breakpoint was reached at a subsequent line (the syntactic breakpoint for the eliminated line), with a message displaying the line number of the line the program actually halted at.
- Turbo C/Turbo Debugger Source lines that have executable code associated with them are shown in a different color from lines that do not. An attempt to set a breakpoint at a line that has no code associated with it has no effect (the lack of a stop-sign glyph makes it clear that the attempt to set the breakpoint failed). Breakpoints can be set at a line that has executable code associated with it, and the program halts at that line. Of the compiler/debugger pairs surveyed, this is the only one that provides truthful behavior for its breakpoint facility.

To investigate the data display facility, I ran the same program (giving it the argument `8`) past the point that the assignment to `y` should have taken place. I then attempted to display `w`, `x`, `y`, and `z`. Expected behavior would be to display whatever value happened to be on the stack for `w` (we see that expected behavior is not always helpful either) and the values `5`, `5`, and `0`, for `x`, `y`, and `z`, respectively. If data has been eliminated (as is the case for most compilers in this instance), truthful behavior would be to say so.

- `acc/gdb`: Because `-O` suppresses debugging information, my attempt to display `w` by name produced the message:
  `No symbol "w" in current context.`
  Attempts to display other variables produced similar messages. This would have been unexpected and misleading, but I'd already failed to set breakpoints at the source level, which warned me.
- `acc/dbx` No debugging could be attempted because `dbx` could not load the program.
- `cc`: Because `-g` suppresses optimization, I was unable to test the source-level data display facility of any debuggers with code compiled with `cc`.
- `gcc/dbx`, `gcc/gdb`: `gcc` eliminated `w`, `y`, and `z`. Attempts to display them by name produced the message:
  "*symbol name*" `is not defined`
  from `dbx` and the message:
  `No symbol` "*symbol name*" `in current context.`
  from `gdb`. For both debuggers, my attempt to display `x` by name gave the value `8` (`x` is noncurrent). This is unexpected and misleading behavior.
- Turbo C/Turbo Debugger Because copy propagation eliminated the use of `x` in line 8, the assignment to `x` was eliminated, thus `Turbo C` eliminated all four variables. Attempts to display them by name produced a message saying that they were unavailable because optimization had eliminated them. This is truthful behavior.
- High C/`mdb`: Because `-g` suppresses dead store elimination, all variables were available for display. The values given were `4055`, `5`, `5`, and `0`, for `w`, `x`, `y`, and `z`, respectively. This is expected behavior.

Though far from an exhaustive test either in breadth (of compilers and debuggers surveyed and facilities investigated) or depth (of optimizations performed), this brief survey gives a feel for how today's compilers and debuggers respond in the presence of optimization:

- Some compiler/debugger pairs provided expected or truthful behavior for some examined facilities.
- Some compiler/debugger pairs did not provide expected or truthful behavior for any examined facilities.

- No compiler/debugger pair provided expected or truthful behavior provided for all examined facilities.

It is also worth noting that neither expected nor truthful behavior always corresponds to the most helpful behavior.

Part II

Producing an Accurate Call-Stack Trace in the Occasional Absence of Frame Pointers

8 Producing an Accurate Call-Stack Trace in the Occasional Absence of Frame Pointers

Debuggers have the capability of displaying the current execution context as a list of active routines and their arguments, in reverse order of invocation. Many terms have been used to describe this list of active routines, including call-stack trace, stack trace, stack dump, procedure traceback and backtrace. I use the term call-stack trace in the face of a lack of unanimity of usage.

On architectures in which there is a stack pointer that is modified in the process of evaluating expressions, this facility relies on information provided by code within each active routine. If this code is eliminated due to optimization, the call-stack trace will contain inaccurate and incomplete information, which may mislead the user. It would be preferable to provide either an admittedly incomplete stack trace containing only accurate information or a stack trace that is identical to that which would be produced if the optimization had not been performed.

Providing a call-stack trace requires the ability to locate the callers's stack frame and symbol table information given the called routine's stack frame. Typically a call-stack trace displays the arguments of active routines but does not display local variables. A debugger may also allow the user to change the apparent context of execution so that any chosen active routine can be treated as the focus of debugging. The user can request the display or modification of arguments or local variables of the routine that is the focus of debugging. To fulfill such requests, the debugger needs the same capability that it needs to provide a call-stack trace: the ability to locate the callers's stack frame and symbol table information given the called routine's stack frame. Subsequently, only providing an accurate call-stack trace is discussed. In particular, I do not further discuss the display of local variables.

This problem has not been discussed in the literature. Zellweger [Zel84] provides a solution to the problem of providing an accurate call-stack trace in the presence of procedure integration (inlining).

I show how an optimizing compiler and interactive debugger can cooperate so that the debugger can provide an accurate call-stack trace while making subroutine calls as inexpensive as possible.

9 The Call-Stack Trace

In order to provide a call-stack trace, a debugger uses information that is provided by the standard code sequence that implements a subroutine call, termed *the calling sequence*. In the presence of an optimization of the calling sequence, some of the information currently used by debuggers will be missing, causing the debugger to provide an inaccurate call-stack trace.

Terminology

At any point in an executing program, some sequence of routines is active. The naming convention used within this paper for such a sequence is F_0, F_1, ..., F_n where F_0 is the first routine called and F_n is the currently executing routine. Thus for an arbitrary active routine F_i, its caller is F_{i-1} and the active routine that it called is F_{i+1}.

In the figures, *ip* denotes the instruction pointer register, *fp* denotes the frame pointer register, and *sp* denotes the stack pointer register,

9.1 The Calling Sequence

In a running program, the currently executing routine must have access to its arguments, the return address, local variables, and compiler temporaries. The standard method in many of today's machine architectures for providing such access is to provide storage for each active routine on a procedure call stack. The storage associated with the routine is known as the routine's *stack frame*. A machine register contains a pointer to the base of the currently executing routine's stack frame, and the arguments and local variables are accessed as offsets from that pointer. The pointer is called the *frame pointer*, and the machine register that by convention contains the frame pointer is called the *frame pointer register* (this name is sometimes shortened to *frame pointer* as well—context is used to distinguish the two).

When one routine F_{i-1} calls another routine F_i, F_{i-1} (typically) pushes F_i's arguments on the stack. The call instruction itself pushes the return address on the stack. Code within F_i, called F_i's *prologue*, gets the machine ready for the body of F_i to execute. This includes (but is not limited to) making space for F_i's local variables and providing access to F_i's local variables and parameters. Access to F_i's local variables and parameters is provided by setting the frame pointer register to point to F_i's stack frame. However, after F_i has completed and control has been returned to F_{i-1}, the frame pointer register must contain F_{i-1}'s frame pointer. F_{i-1}'s frame pointer is therefore saved in F_i's stack frame prior to modifying the frame pointer register to point to F_i's stack frame. Space for F_i's locals is allocated on the stack by adjusting the stack pointer. Immediately prior to returning, code within F_i pops F_{i-1}'s frame pointer from the stack into the frame pointer register. See Figure 9.1 for an example of a standard calling sequence.

9.2 Optimization of the Calling Sequence

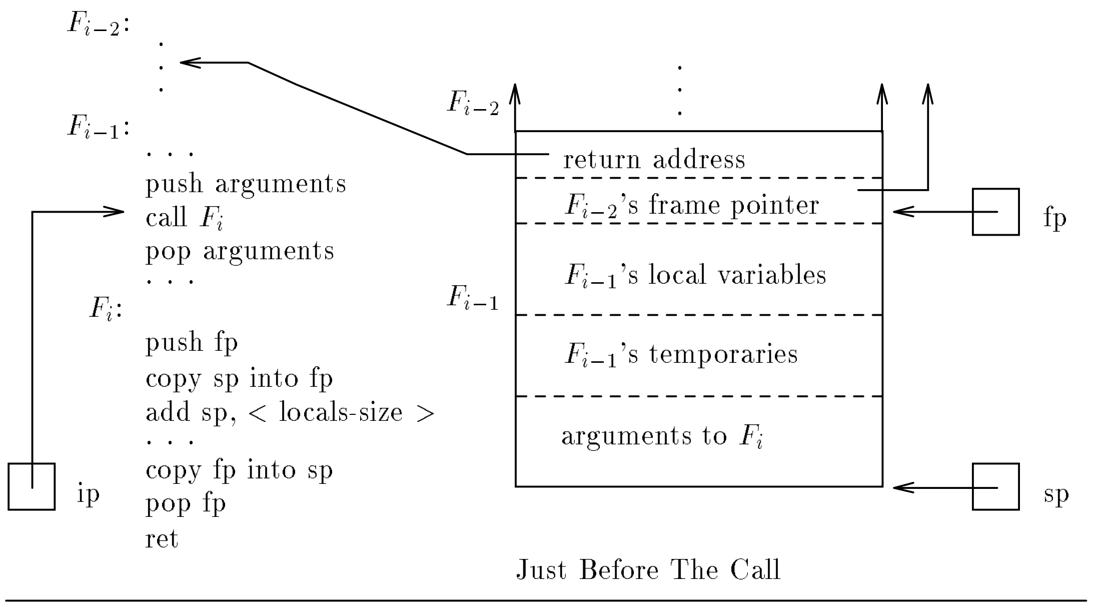
Under some circumstances, the code that pushes F_{i-1}'s frame pointer and sets the frame pointer register to point to F_i's stack frame is unnecessary overhead and can be eliminated. Figure 9.2 gives an example of a calling sequence upon which this optimization has been performed. If F_i is optimized in this manner and F_{i+1} is not, F_{i+1} will save F_{i-1}'s frame pointer, not F_i's as in the unoptimized case. See Figure 9.3 for an example of such a situation.

This optimization is possible when the frame pointer register is not used in F_i's code. This register is used for two purposes:

1. to access F_i's arguments and local variables, and
2. to restore the stack pointer to the position following F_{i-1}'s return address (in preparation for the execution of F_i's return instruction).

Clearly, if F_i has no arguments or local variables, or has them but does not reference them,[9] it will not use its frame pointer for the first purpose. Even if F_i does reference its locals or arguments, a compiler often has enough information to reference them through the stack pointer rather than the frame pointer, although doing so may add to the complexity of the compiler. In addition, a compiler often has enough information to restore the stack pointer to the position following F_{i-1}'s return address without using the frame pointer. Correct code for F_i can be produced without saving F_{i-1}'s frame pointer unless at some instruction

[9] A routine may have arguments and/or local variables but not access them due to carelessness, consistency requirements on a set of routines, or the use of stub routines during the development process.



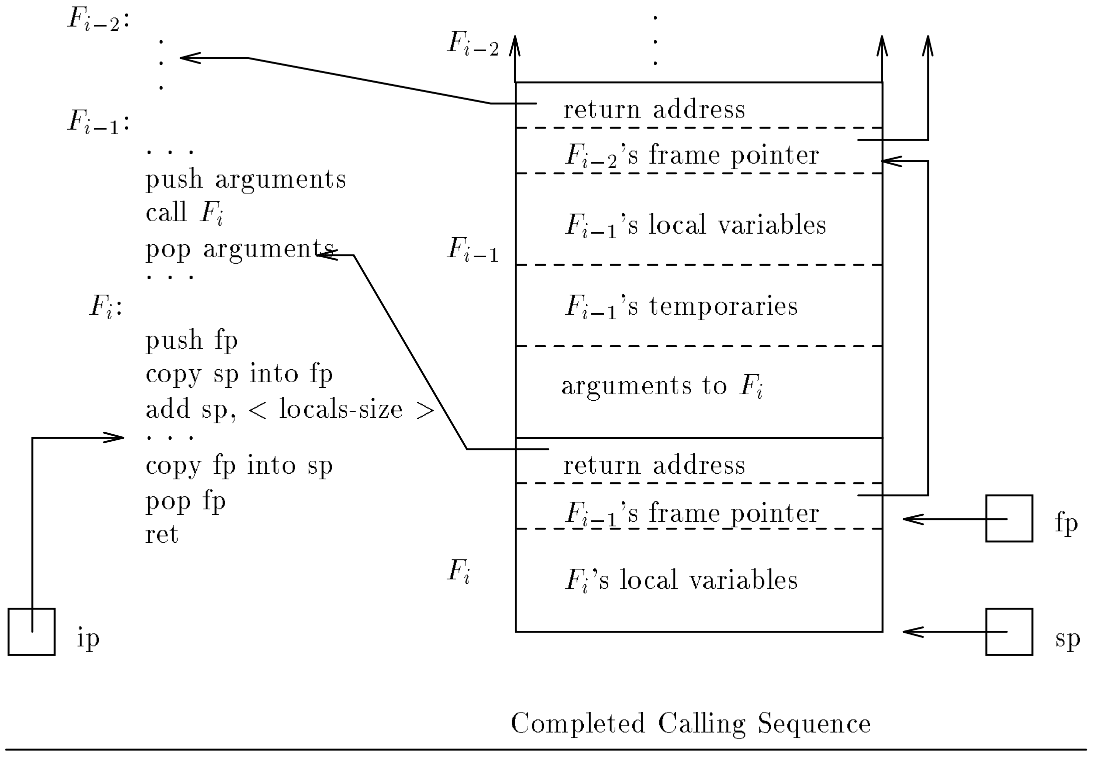
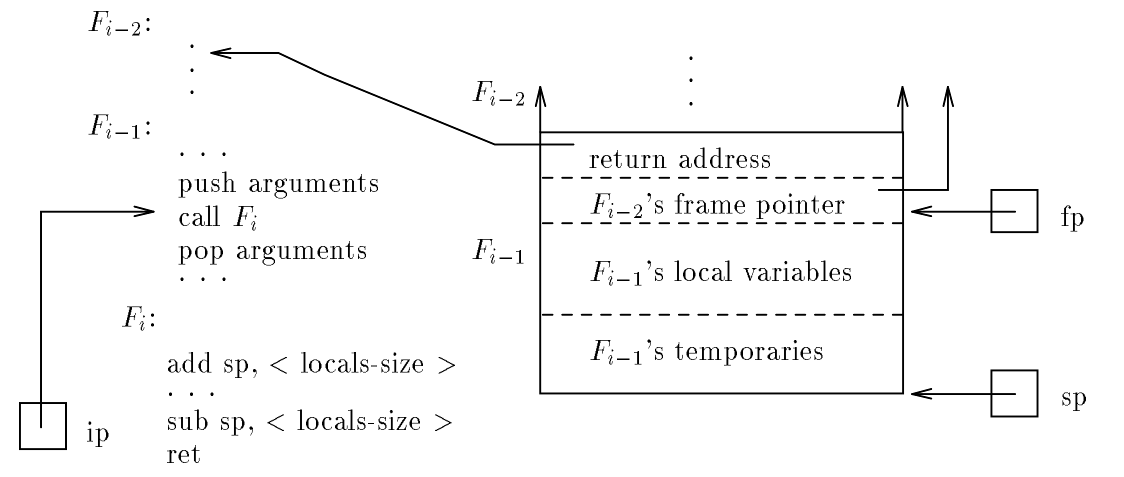


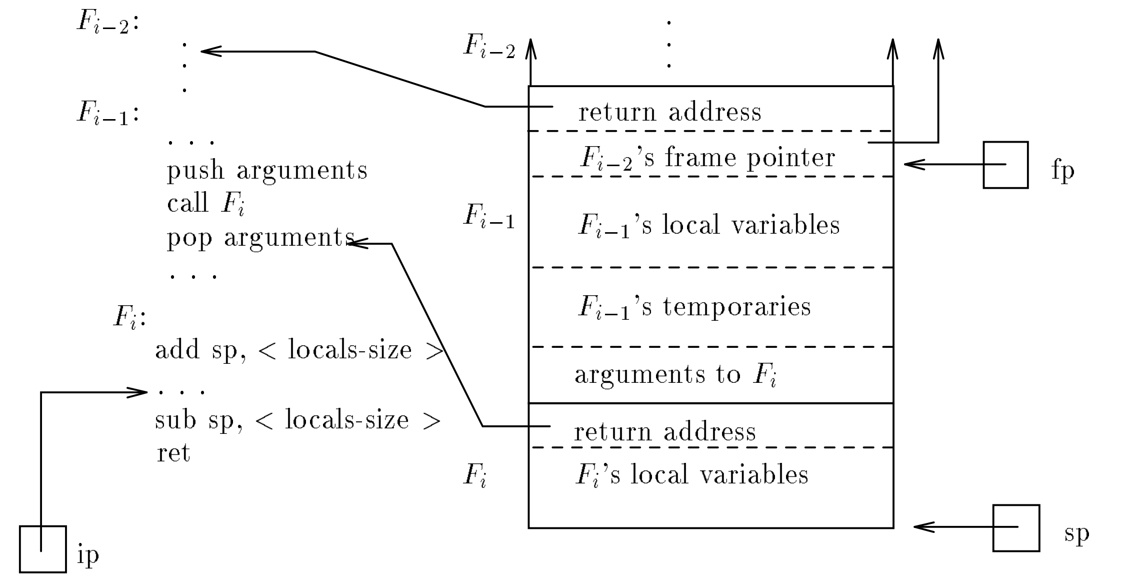
Figure 9.1: Unoptimized Calling Sequence

within F_i, F_i's frame is not constant in size across all calls to F_i. This is infrequent, but happens if the stack is used for dynamic allocation or if the stack pointer is modified along one execution path but not along another.[10] Note that the compiler cannot reference local

[10]One way that the stack pointer can be modified along one execution path but not along another is if parameters are not popped immediately following a call (a routine may be called on one path and not on another; its parameters affect the size of the stack). Another way is if storage for a (conditionally executed)



Just Before The Call



Completed Calling Sequence

Figure 9.2: Optimized Calling Sequence

variables and arguments through the stack pointer in the same circumstances that it cannot restore the stack pointer to the position following F_{i-1}'s return address without using the frame pointer.

Such optimization of a routine's prologue and epilogue is most commonly done for routines that have neither parameters nor local variables, because less analyis is needed on the part of the compiler.

local block is allocated when the block is entered but not deallocated when the block is exited. The stack pointer may not need to be adjusted for each such allocation or set of parameters because an unoptimized prologue restores the stack pointer by setting it to the value in the frame-pointer register.

9.3 Debugger Use of Frame Pointers

The frame pointers that have been pushed onto the call stack by calling sequences form a linked list of pointers to active routine stack frames, with the value in the frame pointer register heading the list. In the unoptimized case, this list contains a pointer to the frame of every active routine, and the frame-pointer register points to the frame of the currently executing routine. A pointer to a routine's frame can be used to locate and access its arguments.

Given an address within a routine, the debugger can find the name and parameter list of the routine by looking in the symbol table. The code address used to find the symbol table entry for the currently executing routine is the address in the instruction pointer when the breakpoint is reached.[11] The code address used to find the symbol table entry for each routine F_i that is not currently executing is the return address stored in the stack frame of routine F_{i+1}.

Let us consider in detail how the debugger will construct the call-stack trace. In following the general description given here, it may be helpful to refer to the example call stack in Figure 9.1. The debugger begins with the currently executing routine F_n. An address within F_n is available from the instruction pointer. From that address, the parameter list and name of F_n are retrieved from the symbol table. The values of the arguments to F_n are available through the frame pointer register. They are displayed formatted according to the type information in the parameter list retrieved from the symbol table.

The return address in F_n's stack frame is an address within the previously invoked routine F_{n-1}. The debugger uses that address to retrieve the parameter list and name of F_{n-1} from the symbol table. The stored frame pointer in F_n's stack frame points to F_{n-1}'s stack frame. The values of the arguments to F_{n-1} are retrieved by the debugger through this stored frame pointer, formatted appropriately, and displayed. The debugger repeats this process, using information found in the stack frame of the just-displayed routine to display that routine's caller until all routines have been displayed. I will call this the *Fchain* method.

An algorithm to construct a call-stack trace using the *Fchain* method is given in Figure 9.4.

10 The Problem

If one or more of the frame pointers have been optimized away, a debugger using the *Fchain* method will construct a call-stack trace that is inaccurate. If a single frame pointer (for routine F_i) has been optimized away, the debugger will construct a call-stack trace that associates F_i's name with the stack frame of F_{i-1} (formatting the values found there according to the symbol table entry for F_i), and neither F_{i-1}'s name nor F_i's arguments will appear. If the frame pointers for routines F_i through F_{i+j} have been optimized away, the debugger will construct a call-stack trace that associates F_{i+j}'s name with the stack frame of F_{i-1} and no information about F_i through F_{i+j-1} will appear. Figure 9.3 shows a call stack containing four active routines, one of which has had this optimization performed on it. The call-stack trace produced by the *Fchain* method on this call stack is:

[11] A debugger saves the values that are in the machine registers when it takes control at a breakpoint, thus when we use the terms "instruction pointer", "frame pointer register", and "stack pointer register" we are actually referring to the debugger's copy of the values that were in these registers when the breakpoint was reached.

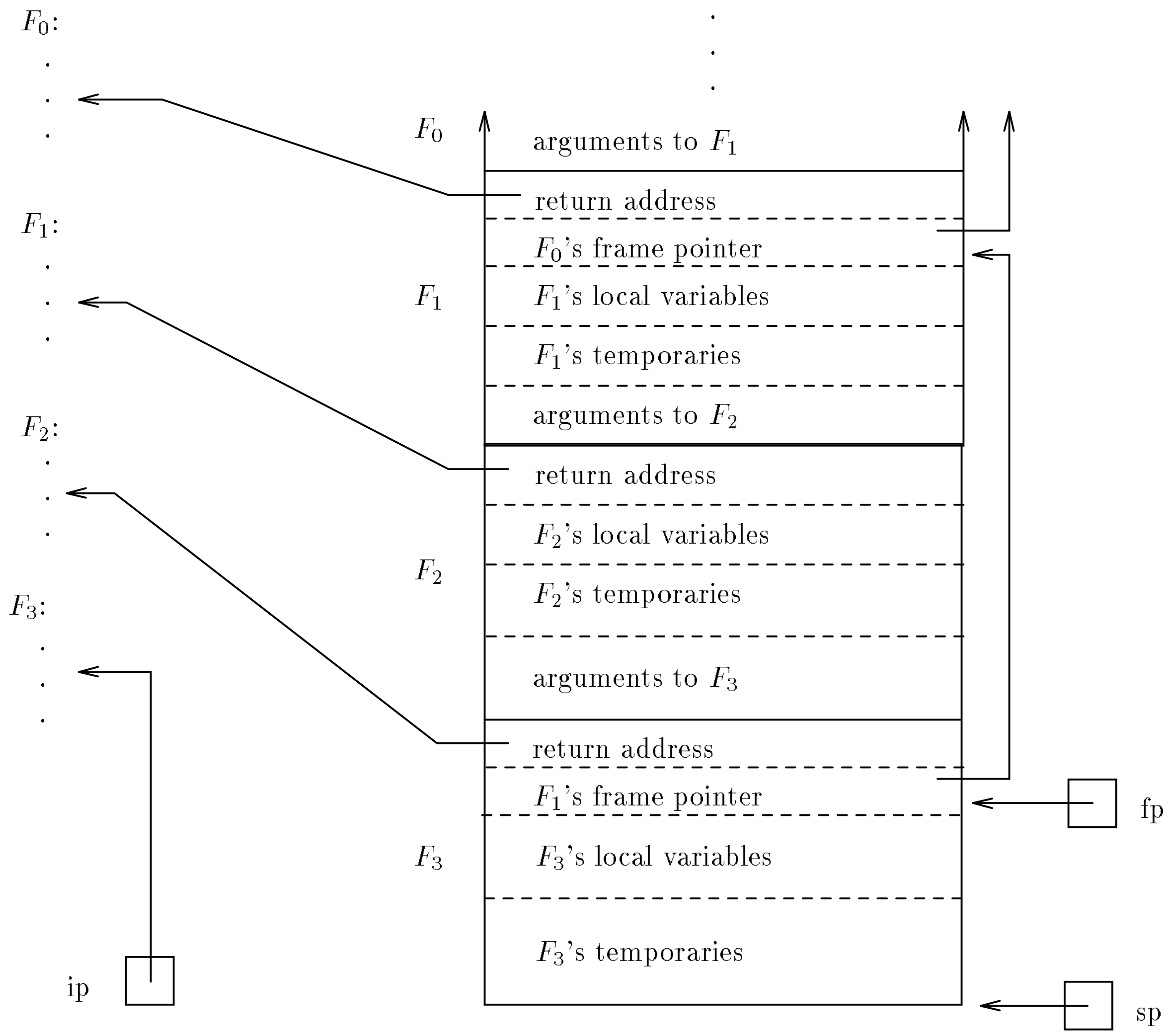


Figure 9.3: Optimized Call-Stack

```
F₃'s name(F₃'s arguments)
F₂'s name(garbage)
F₀'s name(F₀'s arguments)
```

Although four routines are active, the call-stack trace contains only three entries, one of which is inaccurate.

11 Solutions

The general approach to the problem is to have the debugger use an alternative method of constructing the call-stack trace that does not rely on the frame pointers in the call stack. Several solutions are presented.

Figure 9.4: Debugger Algorithm to Display a Call-Stack Trace using the *Fchain* Method

In the following algorithm, *ip* is the instruction pointer register, *fp* is the frame pointer register, and we assume the debugger has the following routines available to it:

- *get-symbol-table-information*, which takes a code address and determines which symbol table entry corresponds to the routine containing that address, then returns the name and parameter type list from the symbol table entry,
- *display-routine-entry*, which takes a routine name, a parameter type list, and a frame pointer, and finds the arguments in the stack frame pointed to by the frame pointer, formats them according to the type list, and displays the routine name and appropriately formatted arguments,
- *get-return-address*, which takes a frame pointer and returns the return address that is stored in the stack frame pointed to by that frame pointer, and
- *get-frame-pointer*, which takes a frame pointer and returns the frame pointer that is stored in the stack frame pointed to by that frame pointer.

The termination condition given here is when the "main" routine (the entry point of the user's code) has been displayed. This is somewhat arbitrary; actual termination conditions are system dependent.

Algorithm *Fchain*-Call-Trace:

```
routine-address ← ip
frame-pointer ← fp
repeat
    name, parameter-types ← get-symbol-table-information(routine-address)
    display-routine-entry(name, parameter-types, frame-pointer)
    routine-address ← get-return-address(frame-pointer)
    frame-pointer ← get-frame-pointer(frame-pointer)
  until name = "main"
```

11.1 The Debugger Maintains Its Own Frame Pointers

The debugger can maintain its own copy of the information that it currently gets from the call stack. The debugger can use *invisible breakpoints* [Zel84] or *program patching* [CT93], [BK92], [CH91], [Kes90] to collect the information. An invisible breakpoint is a breakpoint at which the debugger halts the executing program, takes some action, and continues execution without ever giving control to the user. Program patching can accomplish the same result without the high cost of context switches between the executing program and the debugger. Simply put, the debugger inserts code into the program to take the desired action at the desired points. The information must be maintained in the data-space of the executing program. While program patching is a general technique applicable beyond debugging, its use here is limited to implementing fast invisible breakpoints. Therefore I use the terminology 'invisible breakpoint'.

As we have seen, in order to construct the call-stack trace, the debugger needs for each routine F:

- an address of some instruction within F, which it uses to locate the symbol table entry for F, and
- a pointer to the base of F's stack frame, where it finds F's arguments.

Recall that the first instruction in a routine's prologue pushes the caller's frame pointer into the current stack frame (following the return address). Once that push has occurred, the stack pointer register points to the location that by convention is considered the base of the routine's stack frame. If the debugger were to take control immediately after the first instruction in the prologue, it could make a copy of the value in the stack pointer register, which would give it a pointer to the base of the stack frame, and it could make a copy of the value in the instruction pointer register, which would give it an address of an instruction within the routine. By setting an invisible breakpoint at the second instruction in each routine, the debugger can get the information that it needs for a call-stack trace.

Note that if the debugger set its invisible breakpoint at the *first* instruction in the prologue, it can still get the information that it needs for the call-stack trace. As in the above case, the instruction pointer contains an address within the routine. The stack pointer register contains at this point a value that must be offset by the size of the frame pointer that is about to be pushed in order to get a pointer to the base of the stack frame.

This scenario assumes an unoptimized stack frame. We are interested in the optimized case, when the push of the caller's frame pointer into the stack frame does not occur. However, we have just seen that the debugger can get the information it needs by setting its invisible breakpoint at the first instruction in the prologue. Since the debugger can get this information before the instruction executes, it clearly doesn't matter whether that instruction saves the caller's frame pointer.

The debugger must set a breakpoint at the first instruction of every routine. Each time a routine is called, the debugger copies two pieces of information into its own workspace.[12] It uses this information rather than the information that may (not) be stored in the call stack to construct the call-stack trace.

Clearly, for this to be correct, the debugger must also set an invisible breakpoint at the return instruction of every routine so that it can remove the information for the about-to-return routine from its workspace—otherwise, it would be maintaining not a call-stack trace, but a subroutine-call history. That is, the debugger must store this information in a stack of its own, pushing when a routine is called and popping when the routine returns. I call the debugger's stack the *dstack*. Assuming the debugger correctly maintains its dstack, it can use the information therein just as it would have used the corresponding information that it would find in the program's call stack, to examine the symbol table and to access parameters and local variables. We will call this the *Dstack* method.

The locations at which invisible breakpoints must be set are call and return instructions. The locations of the call instructions are available from the symbol table. Either additional compiler support is needed to ensure that the locations of the return instructions are also placed in the symbol table or the debugger must scan the executable and locate them before executing the program.

In today's most common software production environments, using the debugger involves explicitly invoking the debugger before running the to-be-debugged program—all debugging is *expected debugging activity* [Zel84]. In a growing number of software production environments, the debugger is always available and the program need not be restarted to be debugged. In such an environment, unexpected debugging activity is possible—for example, a program trap or user interrupt may invoke the debugger on a running program. In this

[12]If invisible breakpoints are implemented with program patching technology, this workspace must be in the address space of the executing program.

case, the dstack has not been maintained and a stack trace may be inaccurate. It would not make sense for the *Dstack* method to be employed by default in such an environment, because although it would provide for an accurate stack trace in the case of unexpected debugging activity, the overhead cost would be greater than the savings from the original optimization. In such an environment, the user should be given a choice between restarting the program with the *Dstack* method enabled, or enabling the *Dstack* method at the time the debugging activity begins. In the latter case, a stack trace would be accurate for subroutines called subsequently, but might be inaccurate for those already on the stack.

11.2 A Cheaper Method: The Debugger Maintains Only the Missing Frame Pointers

Subroutine calls are extremely common. The *Dstack* method has a considerable amount of overhead with two invisible breakpoints per call. If setting the invisible breakpoints involves context switches between the executing program and the debugger, this would be an untenable amount of overhead. If the invisible breakpoints are implemented with program patches, the overhead is two very small subroutine calls for each existing call in the program. The *Dstack* method and the *Fchain* method can be combined to lower the overhead. The dstack is entirely redundant if the frame pointer is set up by every routine. It is partially redundant if the frame pointer is optimized away by some routines. Redundancy can be limited by only creating entries in the dstack for routines that do not set up a frame pointer. The compiler can tell the debugger which routines set up frame pointers. The symbol table entry for a routine must be extended to include a field with a boolean entry recording the presence or absence of a frame pointer. There is now overhead of two invisible breakpoints per call only for those routines that do not use a frame pointer.

Even so, the dstack remains partially redundant. It contains only frame-pointers that are not present in the call stack. But each dstack entry also contains an address. The address in the entry for F_i is redundant with the return address which has been placed in the stack frame of F_{i+1} by the call instruction. The dstack can therefore be simplified to contain only frame pointers. I will call this the *Dstack/Fchain* method.

Consider how the debugger will construct the call-stack trace given an optimized call stack and a dstack. The basic difference is where a pointer to the stack frame of each routine is found. Note that the frame pointers stored in the call stack still form a linked list, but now they chain only stack frames of routines that use frame pointers. If F_i uses a frame pointer (indicated by F_i's symbol table entry), then either the frame pointer register or the frame pointer in the stack frame of subsequently called active routine F_{i+j} points to F_i's stack frame, where F_{i+j} is the next routine that uses a frame pointer. If F_i does not use a frame pointer, then the dstack contains a pointer to its stack frame.

An algorithm to construct a call-stack trace for an optimized call stack using the *Dstack/Fchain* method is given in Figure 11.1.

11.3 Non-Local Gotos

Commonly used procedural languages provide some form of jump from the middle of a routine to the middle of some other previously invoked active routine, a jump that is not simply a return to the calling routine. Sometimes this is provided in the language as part of the *goto* facility (as in Pascal), and it is from there that the name non-local goto was

```
Algorithm Dstack/Fchain-Call-Trace:
    routine-address ← ip
    call-stack-frame-pointer ← fp
    temp-dstack ← dstack // copy dstack so pops are not destructive
    dstack-frame-pointer ← pop(temp-dstack)
    repeat
        name, parameter-types ← get-symbol-table-information(routine-address)
        if (uses-frame-pointer(routine-address))
            frame-pointer ← call-stack-frame-pointer
            call-stack-frame-pointer ← get-frame-pointer(frame-pointer)
        else
            frame-pointer ← dstack-frame-pointer
            dstack-frame-pointer ← pop(temp-dstack)
        display-routine-entry(name, parameter-types, frame-pointer)
        routine-address ← get-return-address(frame-pointer)
      until name = "main"
```

Figure 11.1: Algorithm to Display a Call-Stack Trace in the Occasional Absence of Frame Pointers using the *Dstack/Fchain* Method

coined—it is a `goto` whose target is not local to the routine that the `goto` is in. In other languages, library routines perform the same function (e.g., in C, `setjmp` and `longjmp`). Other languages provide some form of exception handling, which can have the same effect.

Any of these forms of non-local goto cause difficulties for the *Dstack* and *Dstack/Fchain* methods. In order to maintain correctness in its dstack, the debugger must reach invisible breakpoints at both the beginning and the end of each routine that does not set up a frame pointer. If a non-local goto occurs, the end of such an active routine may never be encountered, since the routine does not perform a normal return. Instead, the return is performed for it by restoring the machine to an earlier saved state. There is a problem if a routine F_i that does not set up a frame pointer has been called and has not returned, and the target of the goto is code in a routine F_{i-j} called earlier than F_i. F_i will not return normally, the invisible breakpoint at F_i's return statement will not be reached, and the debugger will not pop its entry for F_i from the dstack. Let us assume there is a routine F_{i-j-k} that was called earlier than F_{i-j} (thus is active after the non-local goto) and F_{i-j-k} does not set up a frame pointer. After the non-local goto, the debugger is asked for a call-stack trace. An example of such a situation is shown in Figure 11.2. The debugger algorithm displays the call-stack trace entries accurately until it attempts to display F_{i-j-k}. It determines from the symbol table entry for F_{i-j-k} that F_{i-j-k} has not set up a frame pointer, so it uses the entry on the top of dstack to continue the construction of the call-stack trace. This, of course, is the wrong entry—it is the entry for F_i. Two solutions to this problem are given below.

One Solution: Cleaning Up the Dstack

If immediately after a non-local goto was executed the debugger took control, it could remove any inappropriate entries from the top of the dstack. Any entry with a frame pointer

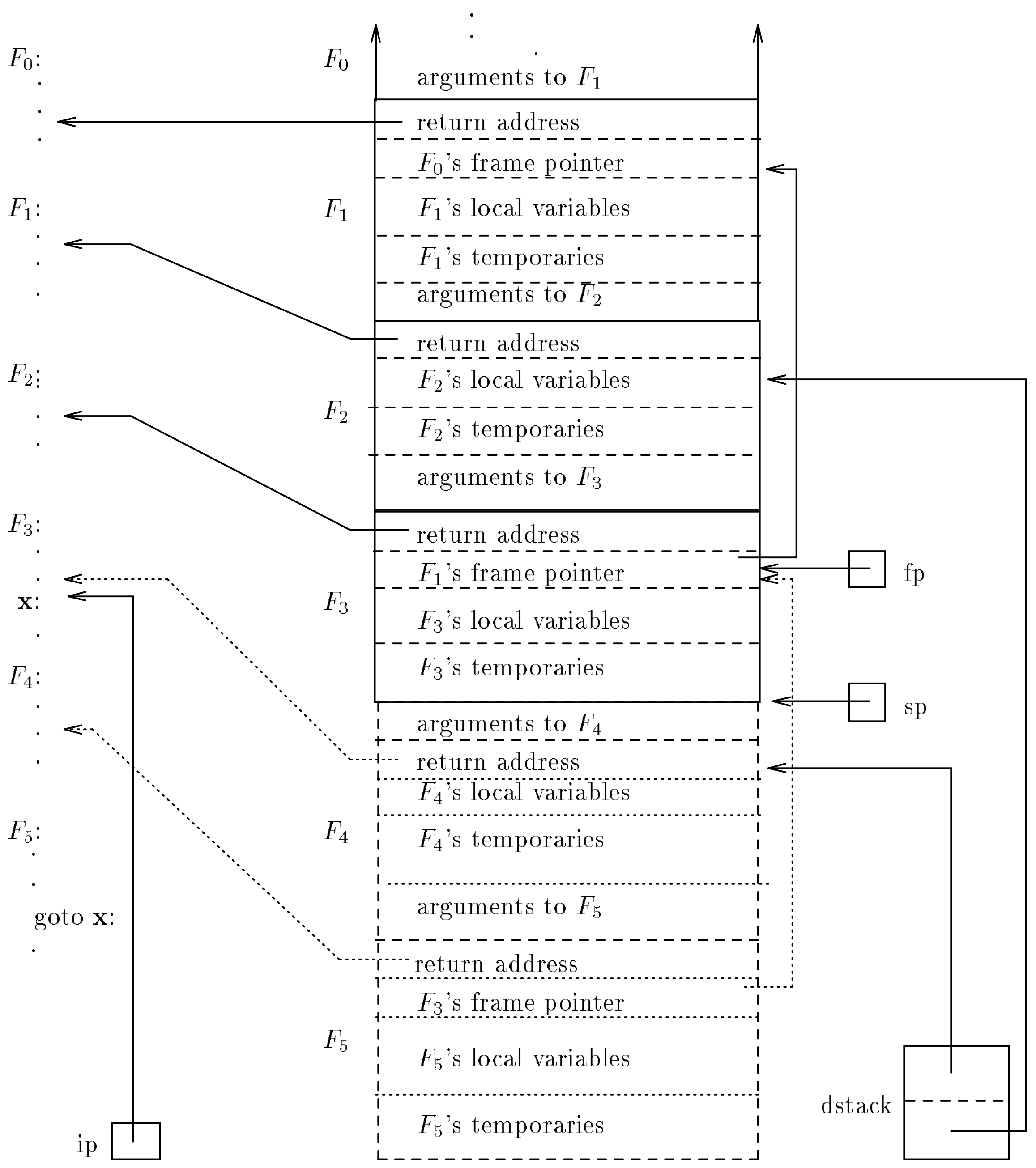


Figure 11.2: Call-Stack After Non-Local Goto

pointing higher on the stack than the value in the stack-pointer register is an entry that should be removed, since such entries must be for routines that have "returned" via the non-local goto.[13] The debugger must take this action right away, since subsequent subroutine calls will cause the stack to grow.

For languages like C, in which non-local gotos are implemented by library routines, the debugger can set an invisible breakpoint at the library routine that implements the goto,

[13]On many architectures the stack grows down, so higher on the stack may mean a lesser value in the frame pointer.

and when it reaches such a breakpoint, the debugger can clean up the dstack. A similar technique may work for languages with exception handling as well.

Languages with direct jumps in the code require some work on the part of the compiler if the invisible breakpoint cleanup solution is to be used. The debugger must be told where to set its invisible breakpoints, so the compiler must provide a list of such addresses to the debugger.

Another Solution: Leaving Redundancy in the Dstack

In the *Dstack* method, because the dstack alone is used to construct call-stack traces, each entry must contain both a frame pointer and a code address. I noted above that when both the call stack and the dstack are used, the code addresses in the dstack are redundant. This redundancy can be exploited.

In constructing a call-stack trace, when the debugger needs to use a dstack entry, it already has a code address A within the routine that is to be displayed next (the code address that was the return address found in the stack frame of the routine that was previously displayed). If the code address in the current dstack entry and A are not addresses within the same routine, the debugger can simply ignore the dstack entry and go on to the next entry. Whenever the invisible breakpoint for the return of a routine that does not use a frame pointer is reached, all dstack entries must be popped until (and including) the entry whose code address is in the same routine as the return instruction.[14]

This scheme does not require that the debugger take control immediately after a non-local goto, and thus does not require any special effort on the part of the compiler. It does increase the expense of constructing a call-stack trace, since more code addresses must be matched with the routines that contain them.

11.4 A "Free" Method: The Compiler Does the Work

Compile-time information can be used to solve this problem with no run-time overhead. I thank Dr. Polle Zellweger and an anonymous reviewer for the essential insight that the size of the caller's stack frame at each call site can be recorded by the compiler and looked up by the debugger.

A routine may contain numerous calls to other routines. Assume routine A calls routines B and C. The size of A's stack frame at the point of the call to B may differ from the size of its stack frame at the point of the call to C. However, the size of A's stack frame at the point of the call to B may be the same every time A is active. If the size of a routine's frame is constant at the point that it makes a call, the size can be determined by the compiler. The return address stored in the stack frame of the called routine is a unique identifier of the call. A table *Fsize* of <*return address, size of caller's stack frame at point of call*> pairs can be provided by a compiler. We can use such a table to compute the appropriate position for each frame pointer without looking in the run-time stack (thus optimization of the calling sequence does no harm). Assuming a lookup function *get-frame-size* for this table that takes a return address and returns the associated stack frame size, and function

[14] If an invalid entry is on the top of the dstack when a call-stack trace is being constructed, it can be popped from the dstack then. However, an entry may be buried on the dstack because subsequent to the non-local goto other routines were called.

get-return-address as defined in Figure 9.4, the position FP_i of the (logical) frame pointer for the i^{th} function on the stack can be determined by:

$$FP_i = FP_{i+1} - \textit{get-frame-size}(\textit{get-return-address}(FP_{i+1})) \tag{11.1}$$

The algorithm given in Figure 9.4 can be used to display the call-stack trace if function *get-frame-pointer* is redefined to return FP_i as defined in equation 11.1.[15] No invisible breakpoints need to be set by the debugger. I call this the *Fsize* method. The *Fsize* method is elegant but incomplete:

1. it fails for routines whose stack-frame size is not constant at the point of a call made by the routine, and
2. it fails if the return address of an active routine is not available. Once a frame pointer is available, the return address of all previously invoked routines can be found. However, if the routine that was executing when the debugger gained control (hereafter the *current* routine) does not set up a frame pointer, the frame-pointer register cannot be used to find the return address of its caller. That return address is present on the stack, but its location is not exactly identified.

When the Frame Size Varies

If the size of a routine's stack frame at the point of a particular call is not constant, the compiler must save the caller's frame pointer, that is, it cannot perform the optimization on the prologue and epilogue of the called function. The chain of frame pointers stored in the stack can be used to locate stack frames of such routines while the *Fsize* table can be used to locate stack frames of routines that do not save the frame-pointer register. I call this the *Fsize/Fchain* method, and it is the solution that I implemented.

An algorithm to construct a call-stack trace for an optimized call stack using the *Fsize/Fchain* method is given in Figure 11.3.

When the Current Routine Has No Frame Pointer

If the current routine does not set up a frame pointer, the location of the return address of its caller is not exactly identified. One option is to search the stack for the return address. The search is bounded on one side by the value in the stack pointer register and on the other by the value in the frame pointer register. The stack search may be complex (it may have to be done more than once, starting at different alignments), but the return address is guaranteed to be on the stack. There is a chance of finding a value in the stack that matches an *Fsize* return-address entry but is not the caller's return address. This chance is probably fairly small (unless the program is a compiler or debugger), but is nonzero. The probability of such an error can be decreased considerably by checking the consistency of the resulting stack trace. In a consistent stack trace, the return address of the entry for function F_i points to an instruction following a call to F_i. If a consistent stack trace that

[15]The problem would be much simpler if the size of A's stack frame were identical for all calls made by A. The stack-frame size could be included in A's symbol table entry, eliminating the need for *Fsize*. Unfortunately, if the stack pointer is used to reference temporaries, fixing the stack frame size would require modification of the stack pointer prior to and following calls, which would cost more than is saved by optimizing the calling sequence. However, in some architectures, such as Mips, the stack pointer is not used in this manner, and this approach is taken [Cor91].

```
Algorithm Fsize/Fchain-Call-Trace:
    routine-address ← ip
    call-stack-frame-pointer ← fp
    repeat
        name, parameter-types ← get-symbol-table-information(routine-address)
        if (uses-frame-pointer(routine-address))
            frame-pointer ← call-stack-frame-pointer
            call-stack-frame-pointer ← get-frame-pointer(frame-pointer)
        else
            frame-pointer ← frame-pointer - get-frame-size(get-return-address(frame-pointer))
        display-routine-entry(name, parameter-types, frame-pointer)
        routine-address ← get-return-address(frame-pointer)
      until name = "main"
```

Figure 11.3: Algorithm to Display a Call-Stack Trace in the Occasional Absence of Frame Pointers using the *Fsize/Fchain* Method

accounts for the entire stack cannot be constructed starting with a potential return address found during a search of the stack, that potential return address is discarded and the search continues.

12 Implementation

I have a prototype implementation of the *Fsize/Fchain* method using MetaWare Incorporated's High C compiler and The Free Software Foundation's gdb debugger, running on an MC68000 based Sun workstation.

Compiler modifications involved adding approximately 50 lines of code spread across seven source files, none of it inside loops. The 'pushes fp' bit was integrated into the existing symbol table format at no space cost. *Fsize* takes eight bytes per call (four bytes for the return address, four bytes for the frame size) plus four bytes per file (to record the number of entries). One *Fsize* table per source file is written to the program's data space as static data and a dbx stab is used to allow the debugger to locate the beginning of each table.

I used compiler benchmarks to compare compilation times and object module sizes for the *Fchain* and *Fsize/Fchain* methods. Table 12.1 gives the time to compile one benchmark program consisting of 13,511 lines of C code (in 44 source files), using each method. The table also gives the sum of the sizes of the object modules produced. For both compilation time and object module size, the increase due to the *Fsize/Fchain* method is given as a percentage of the time or size associated with the *Fchain* method. Individual object-module size increases varied from less than one per cent to 34 per cent. No attempt was made to optimize for space.[16]

[16]There is considerable potential for space optimization in *Fsize*, at the cost of added complexity in reading the table. The return address could be entered as an offset from the function entry point. Typical stack frame sizes can be expressed in a few bits rather than four bytes. If *Fsize* were optimized for space, the *Fsize* version could probably be implemented in half the space of our existing implementation.

	Total Compilation Time	Sum of Object Module Sizes
Fchain	804.3 seconds	490987
Fsize/Fchain	825.4 seconds	589113
increase	2.5 per cent	20 per cent

Table 12.1: Compilation times and object module sizes for the *Fchain* and *Fsize/Fchain* methods. A compiler benchmark consisting of 44 source files containing a total of 13,511 lines of C code was compiled.

Debugger modifications involved adding approximately 150 lines of code spread across five source files. The time the modified debugger spent on call-stack traces was not perceptibly different from the time spent by an unmodified debugger.

13 Solution Summary and Comparison

The enabling technology for producing an accurate call-stack trace in the occasional absence of frame pointers is either the *dstack* or *Fsize*. In this section we summarize and compare the *Dstack/Fchain* and *Fsize/Fchain* methods. First we break the methods down by what is required of the compiler and debugger. We include the effect on the symbol table format, because it is often the limiting factor in what information is passed from the compiler to the debugger.

- *Dstack/Fchain*
  The debugger must be able to determine where the first instruction in each routine is located, and where the routine exits (return instructions) are located. The debugger could determine this information by scanning the executable code. This would slow debugger start-up, but would allow this method to be used with no compiler support. We assume, however, that the task of producing this information would fall to the compiler. In addition, the debugger must be able to determine whether a routine's prologue pushes a copy of the frame pointer register. As before, while the debugger could determine this information by scanning the executable code, we assume that the compiler will be responsible for supplying the information to the debugger.
  - Symbol Table Format
    The symbol table already has a place for the location of the first instruction in a routine. It must be augmented to include the routine exit locations, and to include a 'pushes fp' bit encoding whether a routine's prologue pushes a copy of the frame pointer register.
  - Compiler
    The compiler must set the 'pushes fp' bit and record each routine exit location in the augmented symbol table.
  - Debugger
    The debugger must read the augmented symbol table and place invisible breakpoints at the entry and exit of each routine whose 'pushes fp' bit is off, prior to running the target program. On reaching an invisible breakpoint at routine entry, the debugger must push the stack pointer onto the dstack; on reaching an

invisible breakpoint at routine exit, the debugger must pop the dstack. The debugger must use dstack entries to locate frames of these routines, and use frame pointers saved in the call stack to locate frames of other routines.

- *Fsize/Fchain*
  - Symbol Table Format
    The symbol table must be augmented to include the *Fsize* table of <*return address*, *framesize*> tuples. It must also include the 'pushes fp' bit in the entry for a routine, as above.
  - Compiler
    For each call, the compiler must record in *Fsize* the caller's framesize at the point of the call instruction along with the address of the following instruction. For each routine, the compiler must set the 'pushes fp' bit in the routine's symbol table entry.
  - Debugger
    The debugger must use *Fsize* entries instead of frame pointers saved in the call stack to locate frames of routines whose 'pushes fp' bit is off. The debugger must use frame pointers saved in the call stack to locate frames of other routines. Because *Fsize* is indexed by return addresses, if the current routine's 'pushes fp' bit is off, the debugger must either
    1. print its name only, warn the user that the stack trace may be incomplete, and provide a stack trace beginning with the routine whose frame is pointed to by the frame-pointer register, or
    2. search the stack for a return address that appears in the first field of some *Fsize* entry; the debugger must use that *Fsize* entry to locate the current routine's stack frame, and if that entry does not give a consistent call-stack trace, the search must be repeated.

Next we summarize the advantages and disadvantages of these method relative to each other.

- *Dstack/Fchain*
  - Disadvantages
    This method requires additional complexity in the debugger. If invisible breakpoints involve context switches, then when invisible breakpoints are set, the *Dstack* method has an overhead of hundreds, perhaps thousands, of instructions per call of routines that do not save frame pointers in their stack frame. If invisible breakpoints do not involve context switches, then when invisible breakpoints are set, the *Dstack* method has an overhead of between ten and fifty instructions per call of routines that do not save frame pointers in their stack frame. When invisible breakpoints are not set, the *Dstack* method has no run-time overhead. The symbol table entry for every routine must be read, and the invisible breakpoints set, before the *Dstack* method can be used. This can cause a significant delay in debugger start-up. The location of routine exits must be available to the debugger. This either adds to the complexity of the symbol table and the compiler and to the size of the object modules and executable (typically 4 bytes per routine for the routine exit and one 'pushes fp' bit per routine) or adds to the start-up time and complexity of the debugger.

  - Advantages
    The *Dstack/Fchain* method can be implemented in a debugger without a dependence on compiler support. It can always provide an accurate call-stack trace.
- *Fsize/Fchain*
  - Disadvantages
    This method requires additional complexity in the symbol table, the compiler, and the debugger. The size of the object modules and executable increases significantly. Compilation time increases a little, as does the time required to read the symbol table. In the case that the current routine does not save the frame-pointer register, the method is either incomplete or may be inaccurate.
  - Advantages
    The time cost to the debugger of a larger symbol table is more than offset by the ability to read the table on demand: unlike the *Dstack/Fchain* method, the *Fsize/Fchain* method only requires the symbol table entry for a routine to be read if the routine is active when the call-stack trace is performed. The method has no run-time overhead.

The method used today by interactive debuggers to provide a call-stack trace relies on a frame pointer being set up within the stack frame of each active subroutine. I have described several ways to support a debugger's call-stack trace facility in the circumstance that, due to optimization, some routines do not set up a frame pointer. These methods vary in their costs: there is a trade-off between run-time overhead for the debugger and required symbol table and compiler support.

Part III

Currency Determination

14 Introduction to Currency Determination

A source-level debugger should have the capability of setting a breakpoint in a program at the executable code location corresponding to a source statement. When a breakpoint at some point P is reached, presumably the user wishes to examine the state of the program, often by querying the value of a variable V. Commonly available debuggers, upon receiving such a query, will display the value in V's storage location. Unfortunately, this value may be misleading due to optimization. For example, due to a code motion optimization, an assignment to V may have been done earlier than the source code would lead one to expect. Since one aspect of debugging is examining potential anomalies, the debugger user may expend time and effort attempting to determine why V contains the value that has been displayed when the source code suggests that V should contain some other value.

Figure 14.1 is an example of such a situation caused by constant propagation followed by dead store elimination. Assume that the only use of `x` following the assignment of `constant` to `x` is the assignment of `x` to `y`. Constant propagation removes that use of `x` as shown in the second column of the figure. With that use eliminated, the assignment of `constant` to `x` may be eliminated as shown in the third column. If a breakpoint is reached anywhere following the eliminated assignment to `x` and the debugger is asked to display the value of `x`, typical debuggers will display `expression`. The user, looking at the original source code, may be confused by the fact that the displayed value is not `constant`, or may believe wrongly that the value being assigned to `y` is `expression`. At such a breakpoint, `x` is called *noncurrent* [Hen82], and determining whether optimization has caused a variable's value to be misleading is called *currency determination*. Some solution to the problem of currency determination is necessary for providing expected or truthful behavior.

Optimization may also introduce confusion over where execution is suspended in the program being debugged. The straightforward mapping of statement boundaries onto machine-code locations in unoptimized code is insufficient for optimized code.

The source-level debugger user probes the state of a halted executable while looking at the source code from which it was compiled. Much of the user's activity consists of inference based on the source code and the state information provided by the debugger. This state information includes the location at which execution is halted and the values of variables. One implicit assumption is that the value of each variable in the halted executable corresponds one-to-one to the value that would be predicted by examining the source code and knowing the relevant context, such as within which iteration of a loop execution is suspended. Another implicit assumption is that the location at which execution is halted corresponds to a location in the source code specified by the user. These assumptions may be violated by the presence of optimization, and the inferences that the user draws may be incorrect.

Figure 14.1: Potentially Confusing Optimizations

Original Source Code	After Constant Propagation	After Dead-Store Elimination
`x = expression;`	`x = expression;`	`x = expression;`
`...`	`...`	`...`
`x = constant;`	`x = constant;`	
`...`	`...`	`...`
`y = x;`	`y = constant;`	`y = constant;`
`...`	`...`	`...`

Section 15 presents a mapping from statements to executable code that allows the user to break at a statement and step source statement by source statement in an optimized program. Using this mapping, subsequent sections present a currency determination technique to determine whether a variable has the value the user would expect when execution is suspended at one of these breakpoints. In response to a query about a variable V, this work enables a debugger to

- display V's value without comment if it can be determined that optimization has not given it a misleading value, or
- display V's value with a warning if optimization may have given it a misleading value. The warning can describe how the variable may have gotten the misleading value. The debugger can distinguish the cases in which V is known to have an unexpected value from the cases in which (because of unknown control flow) it may or may not, and adjust the warning accordingly.

15 Breakpoint Model

In an unoptimized translation of a program, code is generated for every source code statement in the order in which it appears in the source code, and the code generated from most statements is contiguous.[17] It is possible to halt unoptimized code at a point that corresponds exactly to a statement boundary in the source code by halting at (before execution of) the first instruction generated from the statement. When execution is suspended at statement S in unoptimized code, all "previous" statements have completed, that is, all code that was generated from statements on the path to S has been executed. No "subsequent" statements have begun, that is, no code that was generated from any statement on the path from S (including code generated from S itself) has been executed. Because of the straightforward nature of the translation, the value in each variable's location matches the value of the variable that would be predicted by a close reading of the source code. Users not versed in optimizing technology expect these characteristics to hold when execution is suspended at a statement boundary.

The state of a suspended program is the context in which debugging takes place, called the *actual debugging context*. In contrast, the *expected debugging context* is the state that would be predicted by an examination of the source code of a program suspended at an identified point. The actual debugging context matches the expected debugging context for an unoptimized program suspended at a statement boundary.

15.1 Treatment of Program Traps

The actual debugging context may not match the expected debugging context, even for unoptimized code, if the program halts on a non-statement boundary, which can happen due to a trap. A program may trap in the middle of an update to a variable, leaving that variable in a decidedly unexpected state. The most important piece of information when a program traps is "What statement caused the trap?", that is, which statement generated the instruction that trapped. This information can be provided by tagging each instruction with a reference to the statement that generated it. This can be encoded in a table by listing the address of the first instruction of each set of contiguous instructions generated from a source statement with a reference to that source statement, thus the trap location reporting problem can be solved by a simple extension of the line table currently emitted by most compilers. The remainder of this work considers programs that are suspended at source-level user-specified locations (breakpoints) only.

15.2 Debugger Capabilities at a Breakpoint in Optimized Code

Optimization may well make it impractical to provide the user with the expected debugging context. Because code may be reordered or eliminated and the instructions generated from a given source statement may not be contiguous, when execution is suspended at statement S in optimized code, no matter what code location is chosen to represent S, some

[17] Code generated from looping or branching statements is typically not contiguous. However, this lack of contiguity is present in the source code as well as the generated code. It can cause debugging anomalies in unoptimized code. For example, placing a breakpoint at a C **for** loop can cause several commonly available debuggers to either break once before loop entry or break each time through the loop, depending on the presence or absence of initialization code.

of the code from previous statements may not yet have been executed and some of the code from subsequent statements may have been executed early.

The debugger user makes inferences based upon the source code and the state of the halted program. This is problematic for debugging optimized code because the inferences are also based upon the implicit assumption that the actual debugging context is equivalent to the expected debugging context.

Of course, it is not possible to prevent a user from making invalid inferences, regardless of the presence of optimization. The best the debugger can do is provide a means of determining when optimization has broken an otherwise valid chain of inference, that is, when an inference that would be valid in the absence of optimization is invalid in its presence. To this end, the debugger acts satisfactorily upon optimized code if at a breakpoint it can report the ways in which the actual debugging context differs from the expected debugging context.

At a breakpoint, the user should be informed of salient differences between the actual debugging context and the expected debugging context. If the user asks to see the value of a variable, the debugger should offer information as to whether its value would be misleading, and why. The user should be able to ask whether a given statement has been executed out of order, and if so, whether it has been executed early or will be executed late. These capabilities allow the user the same power to probe the state of an optimized program at a breakpoint that is available currently for unoptimized programs, because they license valid inferences based on the source code and the state of the suspended program and they provide information that can be used to prevent invalid inferences.

Only those effects of optimization that affect the validity of the user's inferences need to be reported by the debugger. As noted by Coutant et al [CMR88], much of the optimization performed upon a program is irrelevant to the user. It is only optimization that affects user-visible entities, such as source code variables and statement flow-of-control, that the user needs to be informed about. Informing the user of optimization on compiler temporaries is likely to make the debugging job harder, not easier. The same is true of optimization of code generated from the right-hand-side of assignments—the store of the result affects the state of the program as seen from the source-level view, but how that result is computed does not affect the source-level view of program state. Similarly, optimization of an expression whose result determines the outcome of a conditional branch should be invisible to a user debugging at the source level if the branch itself is unaffected.[18] Many statements that start earlier in optimized code than in unoptimized code do so due to code motion of parts of the statements (such as address computations) that are irrelevant to the user's inquiry. Though the optimization of these statements does cause the actual debugging context to differ from the expected debugging context, it does not invalidate user inferences, therefore it is not necessary for the debugger to report that these statements have begun early. Statements that begin early due to source-level-invisible optimization but that otherwise exhibit no source-level-visible effects from optimization are not considered to be executed out of order.

[18]There are circumstances in which it is important for the debugger to reveal the effects of optimization at this level of detail, such as allowing the user to track down a code-generation bug. In such circumstances, it is appropriate to shift to machine-level debugging.

15.3 Breakpoint Locations (Representative Instructions)

Commonly, when setting a breakpoint on a statement, the debugger user wants to break exactly once each time the statement is executed at some location that corresponds to the statement boundary. This is problematic for optimized code, but not providing or closely approximating this capability puts a heavy burden on the user not well-versed in optimizer technology. The capability is necessary to support two common debugging strategies: running until a selected statement is reached, and stepping through the program statement by statement.

In Section 15.1's treatment of program traps, every instruction generated from a statement is associated with that statement. This is possible and appropriate because the program may trap at an arbitrary location that is mapped back to the source code. A breakpoint is specified in source terms and must be mapped onto the machine code. It is inappropriate to associate every instruction generated from a statement with that statement for the purposes of setting breakpoints, because if the instructions are not contiguous, many breakpoints may be reached for a single statement. In contrast, Streepy [Str91] and Brooks et al [BHS92] describe a source-code/breakpoint-location mapping that allows breakpoints to be set at various levels of granularity, including expressions, basic blocks, and subroutines. In the debugger described by Streepy and by Brooks et al, when a statement is selected as the level of granularity, a breakpoint is set at the beginning of each sequence of contiguous instructions generated from the statement.

Under the mapping described in this section, the instruction generated from a statement S that best corresponds to the statement boundary is selected to represent S, and is called the *representative instruction* for S. The address of this instruction is a breakpoint location for S.[19] Where no confusion will result, the representative instruction itself may be referred to as the breakpoint location. The mapping described herein is not in conflict with that described by Streepy [Str91]; each enables debugger capabilities missing from the other. This paper does not concern itself further with breakpoints for language entities other than statements, except to state that the results hold in the presence of such breakpoints.

The choice of a machine instruction as the breakpoint location for a statement should be based on why the user wants to break at that statement. It may be that the user sets a breakpoint at some statement within a loop because it looks like a convenient place to see how the program state is changing on subsequent iterations of the loop. There may be nothing about the chosen statement relevant to the user's purpose except its location within the loop. If that statement were moved out of the loop by optimization, it would be appropriate to set the breakpoint where it used to be, so the breakpoint would be reached each time through the loop. On the other hand, the user may set a breakpoint at some statement to check the values of variables used in an expression in that statement. In that case, if the statement were moved out of the loop by optimization, it would be appropriate

[19]In the most common case, a single instruction will serve as the breakpoint location for a statement. Statements with multiple side effect on user variables will require multiple breakpoint locations, one for each side effect. Optimizations that cause code duplication may require breakpoint location duplication as well—procedure integration (inlining), partial redundancy elimination, and loop unrolling are examples. Even in unoptimized code some statements may require more than a single instruction to represent their breakpoint locations. Loop constructs are an example. The appropriate location to break the first time (before the loop is entered) may be at a different instruction than the appropriate location to break subsequently (each time through the loop).

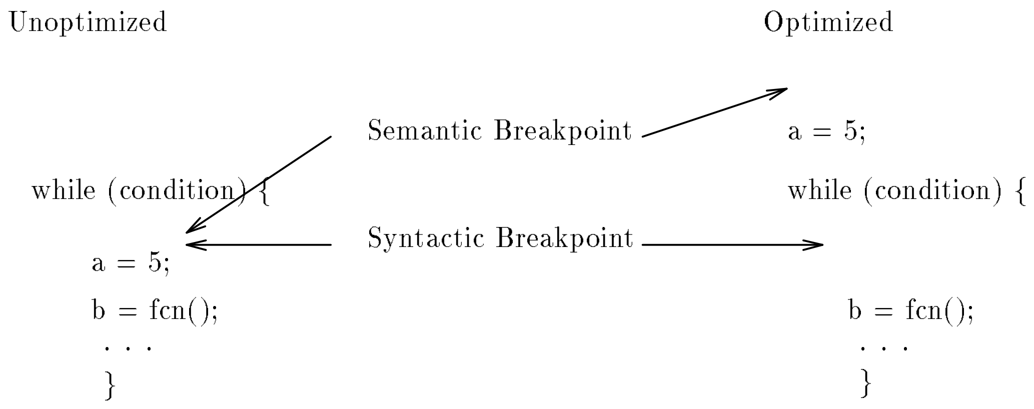


Figure 15.1: Semantic and Syntactic Breakpoint Locations

to set the breakpoint where it ended up, so the values the debugger displays are the actual values used in the expression.

Zellweger [Zel84] introduced the terms *syntactic* and *semantic* breakpoints. If no code motion or elimination has occurred, these are identical. In the presence of code motion or elimination, the order in which syntactic breakpoints are reached reflects the syntactic order of source statements; the syntactic breakpoint for statement n is prior to or at the same location as the syntactic breakpoint for statement $n + 1$. It will be at the same location if the code for n is moved or eliminated. If the code generated from statement n is moved out of a loop, a syntactic breakpoint for n remains inside the loop.

The semantic breakpoint location for a statement is the point at which the action specified by the statement takes place. This does not preserve any particular order. If the code generated from a statement is contiguous, the semantic breakpoint location is the location at which the code for the statement has ended up. If the code generated from statement S is discontiguous, the semantic breakpoint location is the location at which the instruction chosen to represent S has ended up.

Figure 15.1 provides an example of the syntactic and semantic breakpoints for a loop from which optimization has moved an invariant statement.

The choice of a breakpoint location for a statement S affects the correspondence between the actual debugging context and the expected debugging context considerably. Zellweger [Zel84] has a discussion of possible semantic breakpoint locations for statements whose generated code is discontiguous. The view taken in this work is that the best breakpoint location for a programming language construct is the location that corresponds most closely to the source level view of the program: the breakpoint location for a statement should be the address of the instruction that most closely reflects the effect of the statement on user-visible entities (program variables and control flow). For each construct in a programming language, the breakpoint location (equivalently, the representative instruction) should be chosen appropriately.

For statements involving program-variable updates, the instruction that stores into the variable is the right choice. A "store" in this context need not be a store into a memory location. It can be a computation into a register, or a register copy, if that is the instruction that implements the semantics of the source statement. This is illustrated by Figure 15.2, which gives a fragment of source code and an optimized sequence of instructions that could result. One might want to break at statement (2) and examine `a`. If the first instruction generated from a statement is the representative instruction for that statement, a breakpoint

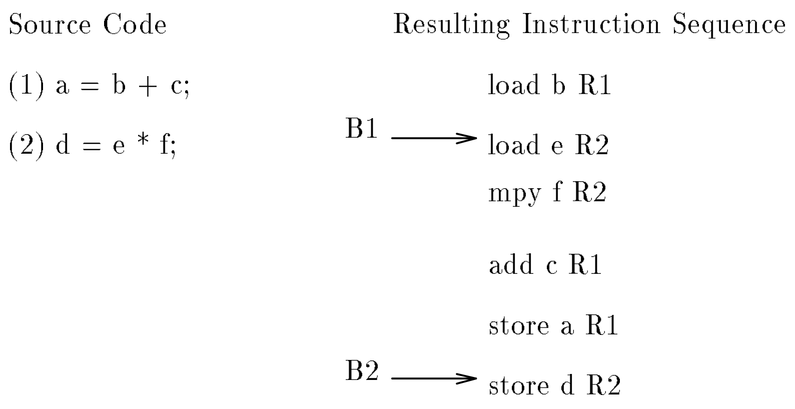


Figure 15.2: Breakpoint Location Choices for Statement (2)

```
Unoptimized                 Optimized

                            R1 = b + c * d
a = x;                      a = x;
if (b + c * d)              if (R1)
    e = a;                      e = a;
else                        else
   e = -a;                     e = -a;
```

Figure 15.3: The Branch is a Sequence Point for Dependences

at statement (2) would suspend execution at B1, resulting in examining `a` when it has not yet had `b + c` stored into it. If, instead, the store instruction is the representative instruction for an assignment, the breakpoint will be reached at B2 and the store into `a` will have occurred. (Note that it may not be possible to choose a representative instruction for statement (2) that gives expected results: for example, if the optimizer has reversed the order of the stores into `a` and `d`, then either `a` or `d` will have an unexpected value.)

For control-flow statements (branching or looping constructs), the instruction that accomplishes the control transfer (typically a conditional branch) is the appropriate choice; it provides a natural sequence point for program dependences. Consider the code fragment in Figure 15.3. The computation of `(b + c * d)` can be computed before the assignment into `a`, however, the jump to the **then** or **else** case must follow the assignment if correctness is to be maintained.

15.4 A Summary of the Proposed Breakpoint Model

A debugger may have the capability of suspending the execution of a program at an arbitrary instruction. The results described in the remainder of this paper hold not at arbitrary instructions but at syntactic breakpoints. The entire set of breakpoints (syntactic and semantic) is the set of representative instructions as described above: for a variable modification that appears in the source code, the store into the variable is the associated breakpoint. An assignment that has side effects will have more than one associated breakpoint. For branching and looping constructs, the branch instruction is the associated breakpoint. The C statement

```
if ((i = j++) == k)
```

has three representative instructions (and therefore three breakpoint locations), one at the store into `j`, one at the store into `i`, and one at the branch to the **then** or **else** case. Choosing the store as the breakpoint location for variable modifications is crucial to the correctness of the work presented in the remainder of the paper. Additional breakpoints, such as those described by Streepy [Str91], could easily be incorporated into this model.

The proposed breakpoint model supports both syntactic and semantic breakpoints. This does not increase the number of breakpoint locations, but it affects the mapping between source-level specifications of breakpoints and breakpoint locations. A source-level specification of a breakpoint is a specification of its type (syntactic or semantic) and a reference to a statement or side effect within a statement. This work does not specify a user interface, so it does not describe the form of such a reference.[20]

The remainder of this work addresses the problem of currency determination, assuming the use of this breakpoint model, restricted to syntactic breakpoints. Section 25 discusses the problems semantic breakpoints introduce for currency determination.

[20] An implementation could accept a statement reference (such as a line number) and set breakpoints at every valid breakpoint contained therein. The user would not need to specify the type of breakpoint nor the side effect within a statement. However, for some statements the debugger would gain control more than once during the execution of the statement, and the location at which the debugger gains control may not be the location the user expects. As always, the debugger should provide enough information that the user is not misled. The advantage of this scenario is that user that is naive about optimization can still use the debugger effectively. The debugger could even gently educate the naive user about the different types of breakpoints.

16 Currency

When the user asks the debugger to display the value of a variable, the user is misled if optimization has caused the value displayed to be different from the value that would be predicted by examining the source code.

The *actual value* of a variable V when execution is suspended at a breakpoint is the value in V's storage location. A variable's *expected value* when execution is suspended at a breakpoint is the value that would be predicted by examining the source code and knowing the relevant context, such as within which iteration of a loop execution is suspended. Abstractly, this would-be-predicted value is the value that would be given to the variable if the program were running on a machine whose instruction set is the source language.

In unoptimized code, at each breakpoint the expected value of every variable is identical to its actual value. In optimized code, as we have seen, the actual value of a variable at some point may differ from its expected value at that point. Hennessy [Hen82] introduced the terms *current*, *noncurrent*, and *endangered* to describe the relationship between a variable's actual value and its expected value at a breakpoint. This relationship is described on the basis of a static analysis, one that has no information about how the breakpoint was reached.

Informally, a variable V is *current* at a breakpoint B if its actual value at B is guaranteed to be the same as its expected value at B no matter what path was taken to B. Examples of current variables are given in Figures 16.1 and 16.2. All examples use program flow graphs. Nodes in the flow graphs represent basic blocks and edges represent basic block connectivity. For clarity of exposition, the example graphs are minimal (for example, there is at most one instruction within a basic block). The language of the examples includes assignment (`a = x` denotes the assignment of `x` into `a`) and a distinguished symbol `bkpt` which represents the instruction at which the breakpoint has been reached. Assignment instructions with the same right hand side assign the result of the equivalent computations into the left hand side; this is how the relationship between assignments in the unoptimized code and assignments in the optimized code is shown. While a statement in a source language that corresponds to either an assignment or a breakpoint may compile to more than a single machine instruction, assignments and breakpoints appearing in flow graphs are referred to as instructions, because a single representative instruction is chosen for each statement.

The examples should be understood as flow graph pieces that contain all the relevant information about a variable at a breakpoint. An example flow graph is representative of the family of flow graphs that contain the example graph with arbitrary other edges, nodes, and instructions, so long as these additional elements do not change which definitions of shown variables reach shown points within the example graph.

Figure 16.1 shows the simplest case of a variable that is current at a breakpoint. There is a single assignment into `a` prior to the breakpoint, and this assignment is unaffected by optimization. There is only one way to reach `bkpt` in both versions of the program, and in both versions, along the only path to `bkpt`, `a` receives its value from the same assignment.

A variable may be current at a breakpoint even if optimization has affected assignments into the variable. Figure 16.2 shows a case in which an assignment into `a` has been moved. Variable `a` is still current at `bkpt`, because the code motion has not changed the fact that along each path `a` receives its value from the same assignment in the unoptimized and optimized versions of the program.

V is *noncurrent* at B if its actual value at B may differ from its expected value at B no matter what path is taken to B (though the two values may happen to be the same

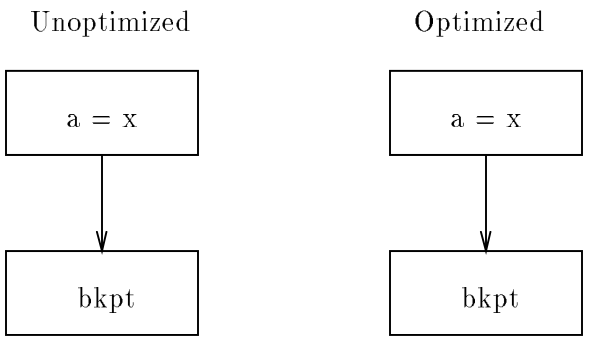


Figure 16.1: Variable `a` is current at `bkpt`

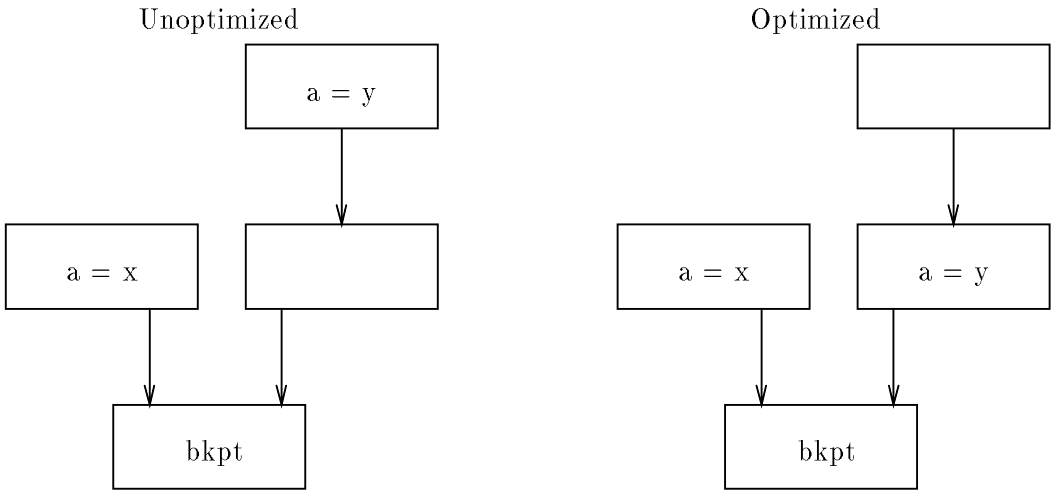


Figure 16.2: Variable `a` is current at `bkpt` in the presence of relevant optimization

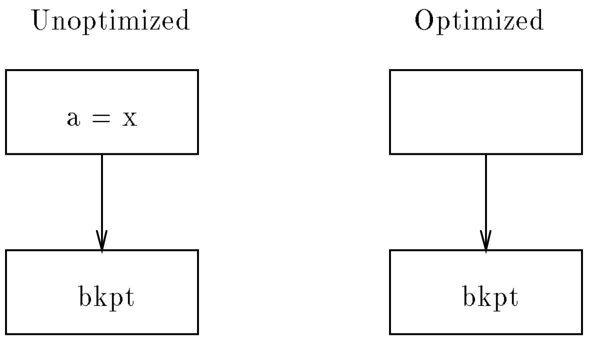


Figure 16.3: Variable `a` is noncurrent at `bkpt`

on some particular input). Figure 16.3 is a simple example of a noncurrent variable, and could be a result of dead store elimination. There is only one way to reach `bkpt` in both versions of the program. There is a single assignment into `a` prior to the breakpoint in the unoptimized code, but in the optimized code there is no corresponding assignment into `a` along the only path to `bkpt`.

Code motion can also make a variable noncurrent. In Figure 16.4, the assignment into `a` reaches `bkpt` in the unoptimized code but does not reach `bkpt` in the optimized code, thus `a` is noncurrent at `bkpt`. Clearly there is a corresponding situation in which the store into

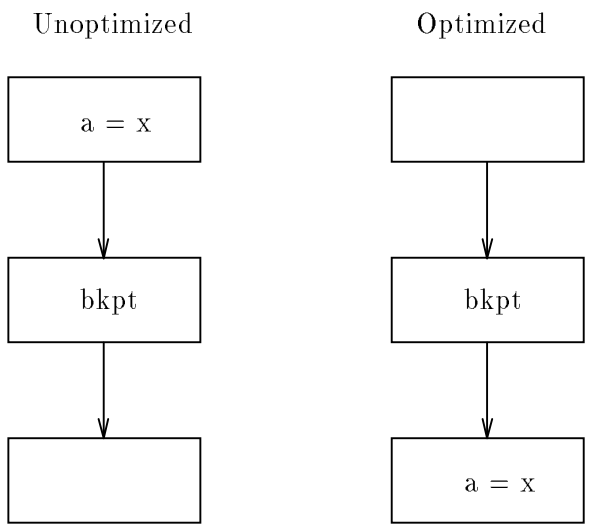


Figure 16.4: Variable `a` is noncurrent at `bkpt` due to code motion

`a` does not move but the breakpoint does. Motion of the breakpoint raises questions that are explored in Section 25.

V is *endangered* at B if there is at least one path to B along which V's actual value at B may differ from its expected value at B. Endangered includes noncurrent as a special case.

In Figure 16.5, along the left-hand path the assignment into `a` that reaches `bkpt` in the unoptimized code corresponds to the assignment into `a` that reaches `bkpt` in the optimized code, but along the right-hand path this is not the case. `a` is endangered by virtue of the right-hand path, and is not noncurrent by virtue of the left-hand path.

The use of the terms current and noncurrent extends to particular paths: in Figure 16.5, `a` is current along the left-hand path and noncurrent along the right-hand path. When execution is suspended at `bkpt` during some particular run of the program, `a` is either current or noncurrent, depending on the path taken to `bkpt`. However, static analysis cannot determine which, because knowledge of the path taken is absent. A debugger that has no access to execution history information can do no better than static analysis. Complete information about the execution path taken could be large, and collecting it could be invasive and time consuming. An open problem, termed *dynamic currency determination*, is how a debugger can collect the minimal information needed to determine whether an endangered variable is current or noncurrent when execution is suspended at a breakpoint. This work assumes such information is unavailable to the debugger.

In order to talk about V's currency along a particular path, a path must be defined in such a way that it makes sense in both the unoptimized and optimized versions of the program, as optimization may modify the program's flow graph.

Definition 16.1: A *path-pair* p is a pair $< p_u, p_o >$ where p_u is a path through the flow graph of an unoptimized version of a program and p_o is a path through the flow graph of an optimized version of the same program such that p_u and p_o are taken on the same set of inputs.

Because a path describes an entire execution, call blocks are expanded. A call blocks appears on a path, followed by the blocks visited in the called subroutine, followed by the call block's successor.

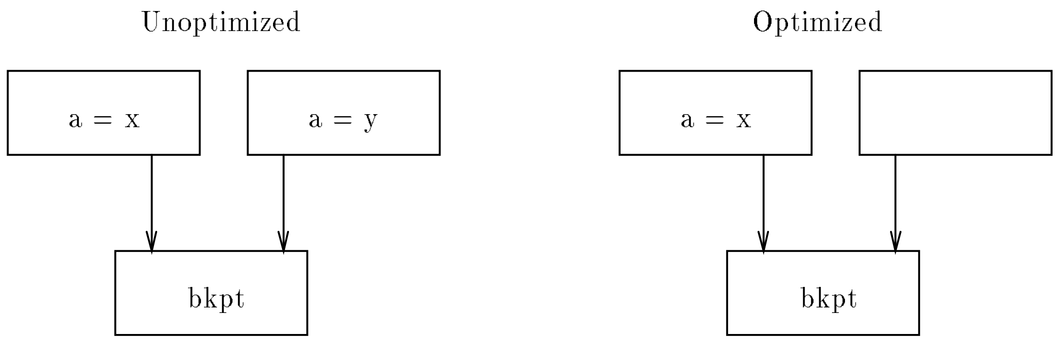


Figure 16.5: Variable `a` is endangered at `bkpt`

I have been using the term 'unoptimized version' as if there were a canonical unoptimized translation of a program, and similarly I have used the term 'optimized version' as if there were a canonical optimized translation. Of course, there are no such canonical translations. What is necessary is that there be a mapping between these 'versions'. The nature of the mapping will be discussed in Section 17.3. For the purposes of this section, simply assume all versions are produced by the same (correct) compiler, which has the same information available to it whether producing an unoptimized version or an optimized version. An implementation would most likely use a single compilation to produce unoptimized intermediate code, providing all necessary knowledge about what I refer to as the unoptimized version without actually generating machine code from it. It would then optimize that intermediate code to produce the optimized version.

Parts of a path-pair are of interest, i.e., a path-pair to a breakpoint or a path-pair from one point to another.

Definition 16.2: A *path-pair p to a block B*, where B is visited in both versions, is a sub-path-pair of a path-pair p' where p_u is a prefix of p'_u ending in an occurrence of B and p_o is a prefix of p'_o ending in the same occurrence of B.

Definition 16.3: A *path-pair p from block A to block B*, where A and B are visited in both versions, is a sub-path-pair of a path-pair p' where p_u is a subsequence of p'_u starting at an occurrence of A and ending at an occurrence of B and p_o is the subsequence of p'_o starting at the same occurrence of A and ending at the same occurrence of B.

I speak loosely of a path-pair to a breakpoint, or a path-pair from one representative instruction to another. In these cases, I mean a path-pair to the block containing the breakpoint, or from the block containing one representative instruction to the block containing the other.

I have made a simplifying assumption that a variable resides in a single location throughout its lifetime. A discussion of the interplay between currency determination and multiple locations appears in Section 26. Relaxing this assumption is a topic for future research. Both assignments to a variable and side effects on that variable modify the value stored in that variable's location. These terms do not distinguish whether the source code or generated code is under discussion. Furthermore, they do not distinguish between unoptimized generated code and optimized generated code. These distinctions are needed in this work because it compares reaching definitions computed on unoptimized code with reaching definitions computed on optimized code. Henceforth the term *assignment* refers to assignments and side effects in the source code.

It is convenient to have a term *definition* that can denote either an assignment or its representative instruction in unoptimized code. This does not introduce ambiguity because either one identifies the other, and the order of occurrence is the same in the source code and unoptimized code generated from it. In contrast, the term *store* denotes a representative instruction for an assignment in optimized code. As with definitions, a store has a corresponding assignment, but unlike definitions, an assignment may have no corresponding store, and the order of occurrence of stores in the machine code may differ from the order of occurrence of assignments in the source code.

An optimizing compiler may be able to determine that two assignments to a variable are equivalent and produce a single instance of generated code for the two of them, or it may generate multiple instances of generated code from a single assignment. Such optimizations essentially make equivalent definitions (or stores) indistinguishable from one another. We will be concerned with determining whether a store that reaches a breakpoint was generated from a definition that reaches the breakpoint. If definitions d and d' are equivalent, and store s was generated from d while s' was generated from d', the compiler is free to eliminate s' so long as s reaches all uses of d'. To account for this, s needs to be treated as if it was generated from either d or d'.

Definition 16.4: A *definition of* V is an equivalence class of assignments to V occurring in the source code of a program that have been determined by a compiler to represent the same or equivalent computations, or the set of representative instructions generated from members of such an equivalence class in an unoptimized version of the program.

Definition 16.5: A *store into* V is the set of representative instructions occurring in an optimized version of a program that were generated from any member of the equivalence class denoted by a definition.[21]

We can now formally define some of the terms described previously. The following definitions assume that the breakpoint location is the same in the optimized and unoptimized version, that is, either that the representative instruction for the statement at which the breakpoint is set has not been moved by optimization or that a syntactic breakpoint has been specified.

The definition of current is complicated by the fact that an assignment through a pointer (or similar alias) must not kill previous assignments, because the pointer might not be pointing at the variable of interest. In the presence of aliases, a sequence of definitions of a variable V and a sequence of stores into V might reach a breakpoint B along a given path p—this is treated in detail in Section 19. Only one definition (store) in the sequence is the last to assign into V, but with static analysis it is not known which one, because it is not known which pointer is an alias for V. If some definition through a pointer is an alias for V, the semantics of the program may require that some other definition through a pointer is also an alias for V (e.g., if the value of the pointer has not changed). Also, we assume that on a given input, a pointer in the optimized version points to the same thing as the same pointer in the unoptimized version. This implies that if some definition through a pointer is an alias for V, the semantics of the program may require that some store through a pointer is also an alias for V.

[21] A store is an equivalence class by the same equivalence relation applied to definitions (having been determined by a compiler to represent the same or equivalent computations).

Definition 16.6: A definition d of V is *turned off* if it is assumed not to be an alias for V, and d is *turned on* if it is assumed to be an alias for V. If d is an alias for V, all definitions and stores program semantics thereby require to be aliases for V are turned off if and only if d is turned off.

A symmetric treatment of *turned off* applies to stores, exchanging the roles of definitions and stores above.

The i^{th} definition of V along a path is written d_i and the i^{th} store into V along a path is written s_i.

Definition 16.7: A store s_i *qualified-reaches a point Bk with definition* d_j *along a path p to Bk* if d_j reaches Bk along p and if, when every d_t, $t > j$, that follows d on p is turned off (consequently turning off other definitions and stores), and all other transparent definitions of V are on, s_i is the store that assigns into V along p. A definition d_i *qualified-reaches a point Bk with store* s_j *along a path p* similarly, exchanging the roles of stores and definitions in the previous sentence.

Definition 16.8: In the absence of assignments through aliases: a variable V is *current at a breakpoint B along path-pair p* iff the store into V that reaches B along p_o was generated from the definition of V that reaches B along p_u.

In the presence of assignments through aliases: V is *current at a breakpoint B along path-pair p* iff for all instances of definitions d_j and instances of stores s_i such that either s_i qualified-reaches B with d_j or d_j qualified-reaches B with s_i along p, s_i is generated from d_j.

Clearly, assignments through aliases are a crucial part of most procedural programming languages. Because of the complexity they introduce, a treatment of currency determination in their absence is given in Section 17, and throughout that section the simpler form of the definition can be assumed. The solution is then generalized to handle assignments through aliases in Section 19.

Definition 16.9: *V is endangered at B along p* if it is not current at B along p.

Definition 16.10: In the absence of assignments through aliases: *V is noncurrent at B along p* iff no store into V that reaches B along p_o was generated from a definition of V that reaches B along p_u.

In the presence of assignments through aliases: V is *noncurrent at B along p* iff for all instances of definitions d_j and instances of stores s_i such that either s_i qualified-reaches B with d_j or d_j qualified-reaches B with s_i along p, s_i is not generated from d_j.

Definition 16.11: *V is current at B* iff V is current at B along each path-pair to B.

Definition 16.12: *V is endangered at B* iff it is endangered at B along at least one path-pair to B.

Definition 16.13: *V is noncurrent at B* iff V is noncurrent at B along each path-pair to B.

We turn now to how to determine which state of currency a variable is in at a breakpoint.

17 Currency Determination

This section describes how to determine a variable's currency at a breakpoint. My approach to currency determination involves solving a set of a dataflow equations. This requires control flow information and within-basic-block ordering information for definitions in the unoptimized program, and the same information for stores in the optimized program. It also requires a mapping between the unoptimized and optimized control flow information.

In this section, we assume that no aliasing is present in the program. This simplifies the presentation. One consequence of this assumption is that only one definition and one store may reach a breakpoint along a single path. In Section 19 we present the mechanisms needed to allow aliasing.

Section 17.1 introduces *paired reaching sets*, which make up the data in the dataflow computation. Section 17.2 discusses the data structure on which the computation is performed, and its relationship to the unoptimized and optimized versions of a program. Section 17.4 discusses the aspects of the dataflow computation that are particular to the problem of currency determination, Section 17.5 gives an algorithm for computing paired reaching sets at block boundaries, Section 17.6 extends it to compute paired reaching sets at breakpoints, and Section 17.7 describes how the results are used to determine the currency of a variable. Appendices B and C.1 offer proofs of the correctness of the work presented in this section.

17.1 Paired Reaching Sets

I have introduced the terms definition and store to distinguish an assignment occurring in the source or unoptimized code from an assignment occurring in the optimized (or machine) code. I am now going to use definition/store pairs to convey some information about both a definition and the store generated from it.

Definition 17.1: A *ds-pair* is a pair (d, s), where d is a definition of a variable V and s is a store into V.

Ds-pairs bear the same relation to the currency determination dataflow computation that definitions bear to a standard reaching definitions computation. I allow 'dotting into' a ds-pair: if P is the ds-pair (x, y), $P.d$ is the definition element x and $P.s$ is the store element y, giving the tautologies $(x, y).d = x$ and $(x, y).s = y$.

Because both definitions and stores are represented, the set of ds-pairs that reaches a breakpoint provides complete information about what should reach and what does reach the breakpoint. These sets are called **P***aired* **R***eaching* **S***ets*: PRS_B^V is the set of ds-pairs relevant to V that reach a breakpoint B. Loosely, $(d, s) \in \mathrm{PRS}_B^V$ means d is a definition of V that should reach B and s is a store into V that does reach B. More precisely, given a definition d of V and a store s into V, independent of whether s was generated from d, I will show that:

- $(d, s) \in \mathrm{PRS}_B^V$ means there is a path-pair p such that d reaches B along p_u and s reaches B along p_o.

I will also show how to compute PRS_B^V, and subsequently I will show that,

- V is current at B iff $\forall(d, s) \in \mathrm{PRS}_B^V$, s was generated from d;
- V is endangered at B iff $\exists(d, s) \in \mathrm{PRS}_B^V$ such that s was not generated from d;
- V is noncurrent at B iff $\forall(d, s) \in \mathrm{PRS}_B^V$, s was not generated from d.

Caveat:

An infeasible path in a flow graph is one that cannot be taken in any execution, and a feasible path is one that can be taken in some execution. Infeasible paths introduce conservative error under this currency determination technique. The claims I have just made hold for programs without infeasible paths. If (d, s) is in PRS_B^V by virtue of an infeasible path, and s was not generated from d, this technique will claim that V is endangered though it may be current. However, we can do no better than the compiler, and like the compiler, we must make the conservative assumption that all paths are feasible. More precisely:

- V is current at B if $\forall(d, s) \in \mathrm{PRS}_B^V$, s was generated from d;
- V is endangered at B iff $\exists(d, s) \in \mathrm{PRS}_B^V$ such that s was not generated from d and (d, s) was placed into PRS_B^V by virtue of a feasible path-pair (an untestable condition);
- V is noncurrent at B if $\forall(d, s)$ placed into PRS_B^V by virtue of a feasible path-pair, s was not generated from d.

The claims are weaker, but in defense of the technique let me point out that the conservative error is in the data (which is the program), not in the modelling of the program or in the currency determination algorithm. One could try to define the weakness away by arguing that people cannot in all cases distinguish feasible paths from infeasible paths, so the formalization of human expectation in the terms current, endangered, and noncurrent should not distinguish them either. Rather than define it away, I will ignore it, sticking with the original formulation of the claims, with the understanding that the results may be conservative when infeasible paths contribute to them.

17.2 The Flow Graph Data Structure

The relationship between the optimized and unoptimized code must be captured in a data structure that can be used by the currency determination algorithm. It may be possible to perform currency determination using information produced by a compiler about an unoptimized version of a program and information produced by a (possibly different) compiler about an optimized version of the same program. We assume that the information used to do currency determination on a compilation unit is produced in a single compilation. Also, information about the unoptimized version of the program is taken from the compiler's intermediate representation of the program prior to the optimizing phases (independent of whether code is generated for an unoptimized version), thus the full facilities of the compiler are available to produce information about the relationship between the unoptimized 'version' and the optimized version.

Currency determination needs the following:

- The assignments that constitute a definition,
- the 'generated from' relationship between definitions and stores (from these two pieces of information, the machine-code instructions that constitute a store can be determined);
- the execution order of statements and side effects within a basic block, for blocks in both the optimized and unoptimized versions, and
- the correspondence between the flow graphs for each version.

The particular encoding of the information is not important here. One possible encoding is described in Section 22. Here we assume that the first three items are available and discuss the fourth.

Some data structures must represent the flow graphs for each version and the correspondence between them. Hereafter, the term *source flow graph*, or simply *source graph* refers to the flow graph for the unoptimized version, and *object flow graph*, or *object graph* refers to the flow graph for the optimized version. *DS-graph* refers to the data structure used to map between them, upon which the data flow computation is performed.

The DS-graph construction is constrained such that a node in the DS-graph is derived from a block in the source graph, from a block in the object graph, or from both. The constraints are described in Section 17.3. A node B in the DS-graph *selects* the block(s) it is derived from: B selects at most one basic block B_u in the source graph and B selects at most one basic block B_o in the object graph. B contains ordering information about definitions occurring in B_u in a definition list. If no block B_u is selected, B contains an empty definition list. Similarly, B contains ordering information about stores occurring in B_o in a store list—if no block B_o is selected, B contains an empty store list. A path p through the DS-graph selects p_u where p_u is the sequence of blocks in the source graph selected by the sequence of nodes in p, and p selects sequence p_o in the object graph similarly. We shall see that by DS-graph construction, p_u forms a path through the source graph and p_o forms a path through the object graph, and $< p_u, p_o >$ is a path-pair.

The correspondence between blocks and edges in the source and object graph is immediate (represented by the identity mapping) when optimization has not caused the object graph to differ in shape from the source graph. When optimization causes changes in the object graph, these changes must be reflected in the DS-graph.

If $(d, s) \in \mathrm{PRS}_B^V$ is to mean that there is a path-pair $< p_u, p_o >$ such that d reaches B along p_u and s reaches B along p_o, then the path through the DS-graph that caused (d, s) to be placed into PRS_B^V must select $< p_u, p_o >$.

A feasible path in the DS-graph is one that selects paths that can be taken in some execution. The DS-graph will contain infeasible paths (i.e., paths that select path-pairs through which execution cannot proceed) if the object graph contains infeasible paths. These paths may contribute information that makes the results of currency determination conservative, i.e., a variable may appear to be endangered due to an infeasible path, as discussed in Section 17.1. It would clearly be preferable to construct the DS-graph so that it contains only feasible paths. Unfortunately, it is not possible in general to determine which paths through a flow graph are feasible.

Definition 17.2: A DS-graph is *valid* if and only if every path p through the DS-graph denotes a path-pair, that is, the pair of paths $< p_u, p_o >$ selected by p is a path pair, and every feasible path-pair is denoted by some path through the DS-graph.

For the purposes of this discussion, we assume that for each node in the DS-graph,

- there is a list of the definitions that are in the block in the source graph selected by the node, in execution order. Because this list contains definitions, I call it the *definition list.* This is equivalent to a list of statements and side effects that appear in that block, in the order in which they appear in the source code.
- there is a list of the stores that are in the block in the object graph selected by the node, in execution order. Because this list contains stores, I call it the *store list.* This is again equivalent to a list of statements and side effects that appear in that block, but execution order in the optimized version is not equivalent to source order.

The definition list must order definitions relative to possible breakpoint locations. Similarly, the store list must order stores relative to possible breakpoint locations. As discussed in Section 15.3 I believe that definitions and stores are appropriate breakpoint locations for assignments and side effects, and this work assumes that definitions, stores, and breakpoints have a uniform representation. If breakpoints at non-statement-level granularity (e.g., expressions or loops) are to be considered, or if a different choice is made for the representative instruction for assignments and side effects, the algorithms presented here will work properly assuming that the definition and store lists order definitions and stores relative to whatever representative instructions are chosen as breakpoint locations. (The choice of representative instructions models the debugger user's expectation. No choice in optimized code provides a perfect reflection of the behavior of unoptimized code.)

These assumptions guarantee that if a node appears in a path through the DS-graph, the code in the unoptimized version represented by that node is the code that would be executed (in proper order) if the block represented by that node were visited on that path, and we have the analogous guarantee relative to the optimized version.

A node n in the DS-graph may select a block in one version that does not exist in the other version, in which case n does not select any block in the latter version. The DS-graph is constructed so that the appropriate list (definition or store) is empty for n. For example, if a loop pre-header were introduced by optimization, the DS-graph will have a node that selects the pre-header in the object graph and does not select any block in the source graph. That node will have an empty definition list.

Store lists reference a subset of instructions, with some additional information. In an implementation, the store lists could be contained in the object graph blocks rather than in the DS-graph nodes. The definition lists must be kept in the DS-graph nodes because some optimizations require definition lists to be copied in such a way that each copy of a definition is in a distinct equivalence class. For simplicity of presentation, the discussion assumes that both definition and store lists reside in DS-graph nodes.

17.3 Constraints on the DS-Graph

The DS-graph begins isomorphic to the source graph, with definition lists replacing the code within blocks, and where each node selects the block it maps to. Before any optimizations change its shape, it is clearly valid.

Some notation:

V^S is the set of vertices and E^S is the set of edges in the source graph.

V^D is the set of vertices and E^D is the set of edges in the DS-graph.

V^O is the set of vertices and E^O is the set of edges in the object graph.

DS-Graph Creation Rule:

- A DS-graph is created before any optimization has been performed. At the same time, the object graph is created. The object graph is a copy of the source graph. For each block B_u in the source graph, a DS-graph node B is created such that B selects B_u and B_o. The definition list for B is abstracted from the code in B_u and copied to the store list for B. For each edge $(h_u, t_u) \in E^S$, edge (h_o, t_o) is placed in E^O. and edge (h, t) is placed in E^D. The name of a block in the source graph is subscripted by u and the name of a block in the object graph is subscripted by o, thus B_o is a copy of B_u.

Any changes made to the shape of the flow graph by optimization are reflected in changes to the shape of the DS-graph and are constrained by the graph transformations below, which maintain the validity of the DS-graph. The currency determination method presented in this work applies only to optimizations that change the flow graph in a manner that can be modelled by iterative application of these transformations. A transformation on the object graph and the accompanying transformation on the DS-graph are described together. A transformation is applied to the DS-graph only when the accompanying transformation is applied to the object graph, thus a transformation is applied only if it is semantically valid.

Let O_i denote the object graph prior to a graph transformation, and O_{i+1} denote the transformed object graph. Let DS_i denote the DS-graph prior to the graph transformation, and DS_{i+1} denote the transformed DS-graph. Let S denote the source graph, which is unchanged by transformations.

A *block boundary marker* is a distinguished instruction used to represent a position in the code stream. A block boundary marker may be selected by a node in the DS-graph, and is considered to be a block in the object graph. When two blocks A and B are coalesced, a block boundary marker M is placed in the code stream of the resulting block C following the last instruction from A and preceding the first instruction from B. Suppose A was selected by node n_A and B was selected by node n_B prior to the coalescing of A and B. Subsequent to coalescing, C is selected by n_A, and n_A's store list comprises the instructions from the top of C down to M. M is selected by n_B, and n_B's store list comprises the instructions following M to the bottom of C (or to the next block boundary marker).

Oselects is a set of functions from DS-graph nodes to object graph blocks that returns the block selected by the argument node. Each such function applies to a particular DS-graph and the corresponding object graph, and is indexed as DS and O are indexed. Thus $Oselects_i(n) = n_o$ means that node n in DS_i selects block n_o in O_i. *Sselects* is similar but maps nodes to source graph blocks. $Oselects_{i+1}(n) = Oselects_i(n)$ and $Sselects_{i+1}(n) = Sselects_i(n)$ unless explicitly changed. For clarity, nodes or blocks created during transformations will be distinguished by an overbar (e.g., $\overline{v}$).

Some of the graph transformations copy nodes in the DS-graph when at first glance it seems unnecessary. The reason for these copies is that when transformations are composed, operations are done on all subpaths in the DS-graph that correspond to an edge in the object graph. If a node in the DS-graph were on more than one such subpath, the transformations could not successfully be composed.

Graph Transformations:

1. Introducing a block.

   An example of this transformation is shown in Figure 17.1.

   The object graph transformation:

   A block is introduced between a complete bipartite subgraph of O_i with vertex sets H^O (for *H*ead) and T^O (for *T*ail). It is typical (but not necessary) for either H^O or T^O to be a singleton set.[22]

   Let $\overline{b}$ be the block introduced by the transformation.

   $O_i = (V^O, E^O)$, $H^O \subseteq V^O$, $T^O \subseteq V^O$, and $H^O \cap T^O = \emptyset$

   $Del = \{(h,t) | h \in H^O, t \in T^O\}$

[22]If a single condition governs the exit of a basic block, T^O can have cardinality at most two, since blocks in T^O end up as successors of the introduced block.

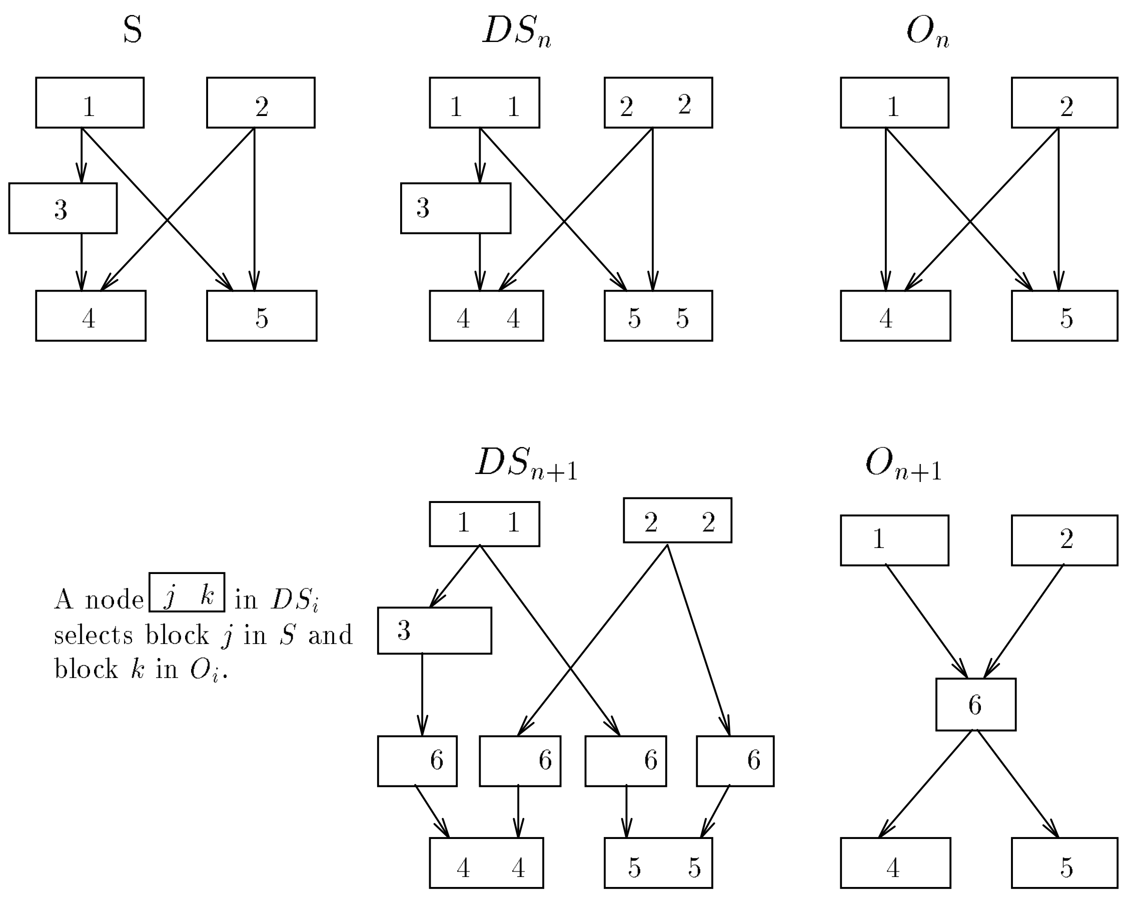

S
DS_n
O_n
DS_{n+1}
O_{n+1}
A node $\boxed{j\ \ k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.1: Graph Transformation 1: introducing a block. In the object graph, a block is introduced between a complete bipartite subgraph. In DS_n, there is a path corresponding to each edge deleted in the object graph transformation. A node selecting the introduced block is added to the end of each such path.

$Add_H = \{(h, \overline{b}) | h \in H^O\}$
$Add_T = \{(\overline{b}, t) | t \in T^O\}$
$O_{i+1} = (V^O \cup \{\overline{b}\}, (E^O - Del) \cup Add_H \cup Add_T)$
Maintaining the semantics of a program imposes the following constraint on this transformation of the object graph: path p_i is taken through O_i on input I if and only if path p_{i+1} is taken through O_{i+1} on I, where p_{i+1} is derived from p_i by replacing each edge (h, t) where $h \in H^O$ and $t \in T^O$ with the subpath $< h, \overline{b}, t >$.

The DS-graph transformation:

A subpath in the DS-graph may correspond to an edge in the object graph. Node $\overline{n}$ (selecting $\overline{b}$) is introduced at the end of each such subpath.

$DS_i = (V^{DS}, E^{DS})$

$SubPaths = \{s | s = <v_1, v_2, \ldots, a_s{=}v_{length(s-1)}, b_s{=}v_{length(s)}>, v_j \in V^{DS},$
$Oselects_i(v_1) \in H^O, Oselects_i(b_s) \in T^O$, and $Oselects_i(v_k) = Null$ for $1 < k < length(s)\}$

$NewNodes = \{\overline{n}_s | s \in SubPaths\}$

$Oselects_{i+1}(\overline{n}_s \in NewNodes) = \overline{b}$

$Sselects_{i+1}(\overline{n}_s \in NewNodes) = Null$

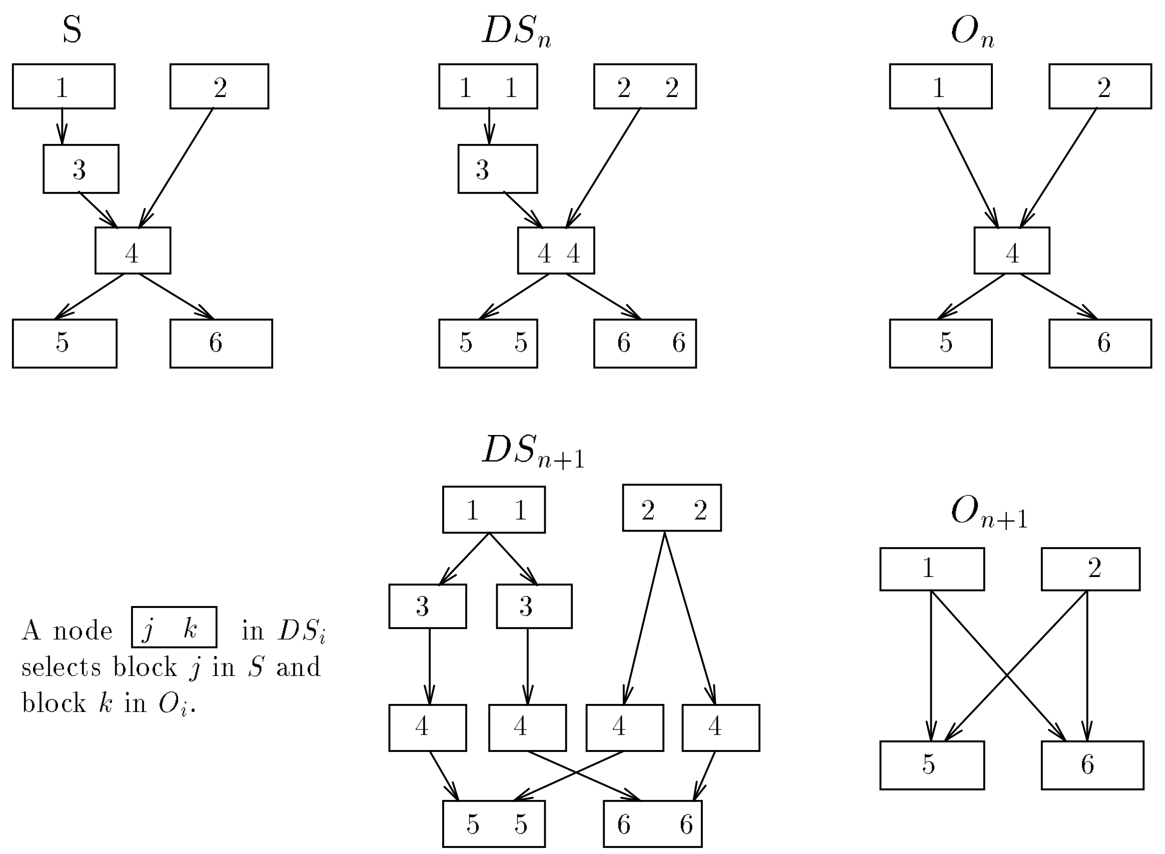

S
DS_n
O_n
DS_{n+1}
O_{n+1}
A node $\boxed{j \quad k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.2: Graph Transformation 2: deleting a block. In the object graph, predecessors and successors of the deleted block form a complete bipartite subgraph in the resulting graph. In the DS-graph, one path is constructed for each edge introduced in the object graph transformation. Nodes are duplicated so that each such path selects the requisite blocks in the source graph.

$DelEdges = \{(a_s, b_s) | s \in SubPaths\}$

$NewEdges = \{(a_s, \overline{n}_s) | (a_s, b_s) \in DelEdges\} \cup \{(\overline{n}_s, b_s) | (a_s, b_s) \in DelEdges\}$

$DS_{i+1} = (V^{DS} \cup NewNodes, (E^{DS} - DelEdges) \cup NewEdges)$

2. Deleting a block.

   An example of this transformation is shown in Figure 17.2.

   The object graph transformation:
   When a block is deleted, its predecessors and successors form a complete bipartite subgraph in the resulting object graph. This is the inverse of the previous transformation. It is typical (but not necessary) for either H^O or T^O to be a singleton set.[23]

   Let b be the block deleted by the transformation.

   $O_i = (V^O, E^O)$

   $H^O = \{h | (h, b) \in E^O \text{ and } h \neq b\}$

   $T^O = \{t | (b, t) \in E^O \text{ and } t \neq b\}$

[23] If a single condition governs the exit of a basic block, T^O can have cardinality at most two, since blocks in T^O are successors of the deleted block prior to the transformation.

$Del = \{(h,b)|(h,b) \in E^O\} \cup \{(b,t)|(b,t) \in E^O\}$

$Add = \{(h,t)|h \in H^O \text{ and } t \in T^O\}$

$O_{i+1} = (V^O - \{b\}, (E^O - Del) \cup Add)$

Maintaining the semantics of a program imposes the following constraint on this transformation of the object graph: if path p_i is taken through O_i on input I, then path p_{i+1} is taken through O_{i+1} on I, where p_{i+1} is derived from p_i by replacing each subpath $< h, b, t >$ with the subpath $< h, t >$.

The DS-graph transformation:

New paths are constructed in the DS-graph corresponding to the new edges in the object graph. There may be nodes on the path from the node that selects a predecessor of the eliminated block to the node that selects a successor of the eliminated block that do not select any object blocks. These nodes, as well as the node that selects the eliminated block, are duplicated so that there is one copy for each new path. Unlike the transformations on the object graph, this DS-graph transformation is not the inverse of the previous transformation.

Let $v_{s,j}$ denote the j^{th} node on subpath s.

$DS_i = (V^{DS}, E^{DS})$

$SubPaths = \{s|s =< h_s = v_{s,1}, v_{s,2}, \ldots, t_s = v_{s,length(s)} >, Oselects_i(h_s) \in H^O,$
$\quad Oselects_i(t_s) \in T^O, \exists j \text{ such that } 1 < j < length(s) \text{ and } Oselects_i(v_{s,j}) = b,$
$\quad \text{and } Oselects_i(v_{s,l}) = Null \text{ for } l \neq j \text{ and } 1 < l < length(s)\}$

$DelNodes = \{v_{s,j}|s \in SubPaths \text{ and } 1 < j < length(s)\}$

$NewNodes = \{\overline{v}_{s,j}|v_{s,j} \in V^{DS}, s \in SubPaths \text{ and } 1 < j < length(s)\}$[24]

$Oselects_{i+1}(\overline{v} \in NewNodes) = Null$

$Sselects_{i+1}(\overline{v} \in NewNodes) = Sselects_i(v)$

$DelEdges = \{e|e \text{ is on some } s \in SubPaths\}$

$NewEdges = \{(\overline{v}_{s,j}, \overline{v}_{s,j+1})|v_{s,j} \in V^{DS}, v_{s,j+1} \in V^{DS}, \text{ and } s \in Subpaths,$
$\quad \text{for } 1 < j < length(s)\} \cup \{(h_s, \overline{v}_{s,2}), (\overline{v}_{s,length(s)-1}, t_s)|s \in Subpaths\}$

$DS_{i+1} = ((V^{DS} - DelNodes) \cup NewNodes, (E^{DS} - DelEdges) \cup NewEdges)$

3. Deleting an edge.

An example of this transformation is shown in Figure 17.3.

The object graph transformation:

Let (h,t) be the edge deleted by the transformation.

$O_i = (V^O, E^O)$

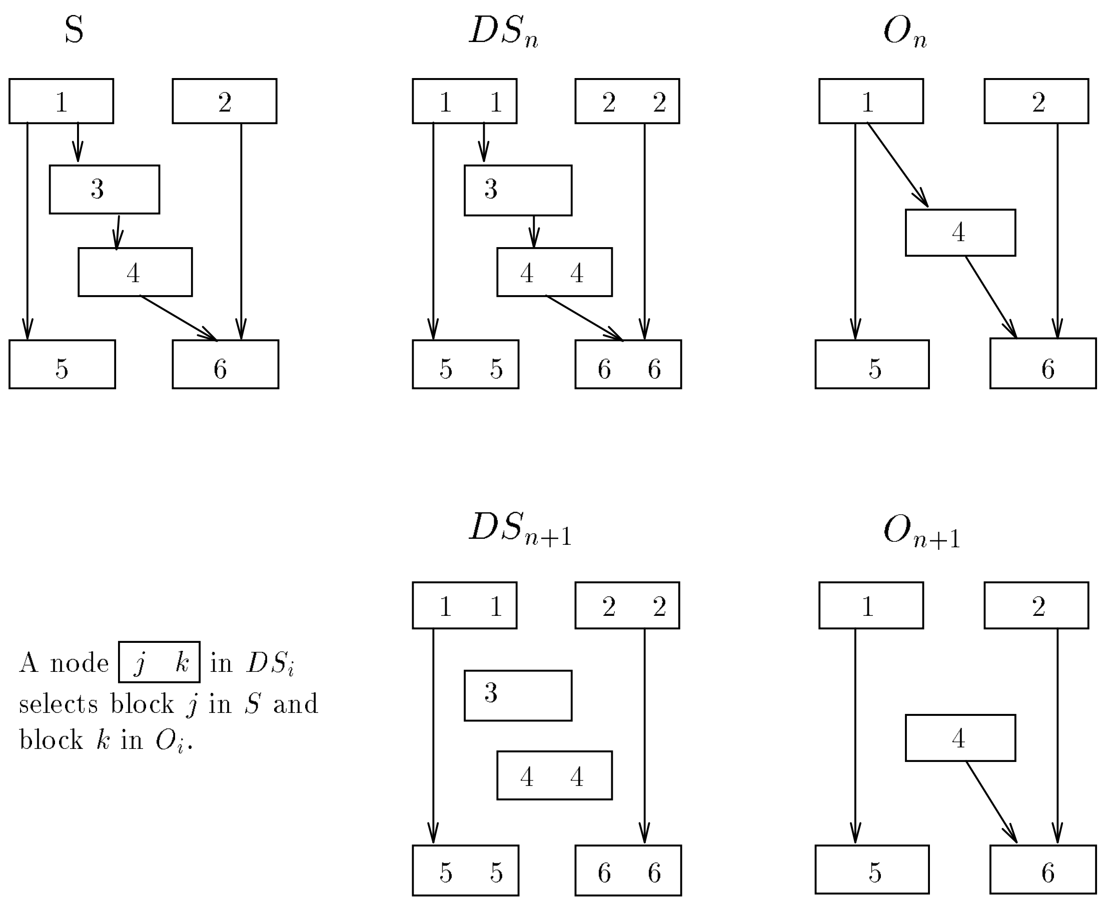
$O_{i+1} = (V^O, E^O - \{(h,t)\})$

Maintaining the semantics of a program imposes the following constraint on this transformation of the object graph: there is no input that causes a path through O_i containing the subpath $< h_o, t_o >$ to be taken.

The DS-graph transformation:

When an edge is eliminated from the object graph, the corresponding edges are eliminated from the DS-graph, and the closure of edges out of subsequentlysubsequentlyreachable nodes are also eliminated from the DS-graph.

[24] $v_{s,j}$ may be on more than one subpath in *SubPaths*. Thus $v_{s,j} = v_{s',j'}$ does not imply that $s = s'$ or $j = j'$. However, a copy is made for each subpath in *SubPaths*, so $\overline{v}_{s,j} = \overline{v}_{s',j'}$ does imply that $s = s'$ and $j = j'$.



A node $\boxed{j \quad k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.3: Graph Transformation 3: deleting an edge. An edge is deleted in the object graph. Paths selecting that edge in DS_n are deleted, and all edges out of nodes that are subsequently unreachable are deleted.

$DS_i = (V^{DS}, E^{DS})$

$\mathit{SubPaths} = \{s | s = < v_1, v_2, \ldots, v_k >, v_j \in V^{DS}, \mathit{Oselects}_i(v_1) = h,$
$\quad \mathit{Oselects}_i(v_k) = t$, and $\mathit{Oselects}_i(v_j) = \mathit{Null}$ for $1 < j < k\}$

$\mathit{DelPaths} = \{e | e$ is on some $s \in \mathit{SubPaths}\}$

$\tau(X) = X \cup \{(n, succ) | \forall (pred, n) \in E^{DS}, (pred, n) \in X\}$

$\mathit{DelEdges}$ = the closure of $\mathit{DelPaths}$ under τ

$DS_{i+1} = (V^{DS}, E^{DS} - \mathit{DelEdges})$

4. Coalescing two blocks into a single block.

An example of this transformation is shown in Figure 17.4.

The object graph transformation:

Let '|' be a code concatenation operator, Code be a function from blocks to the code they contain, and M be a block boundary marker.

Let h be the first of the to-be-coalesced blocks, and t be the second of the to-be-coalesced blocks: t is eliminated by the transformation.

$O_i = (V^O, E^O)$

$\mathit{Code}_{i+1}(h) = \mathit{Code}_i(h) | M | \mathit{Code}_i(t)$

$\mathit{Del} = \{(h, t)\} \cup \{(t, x) | (t, x) \in E^O\}$

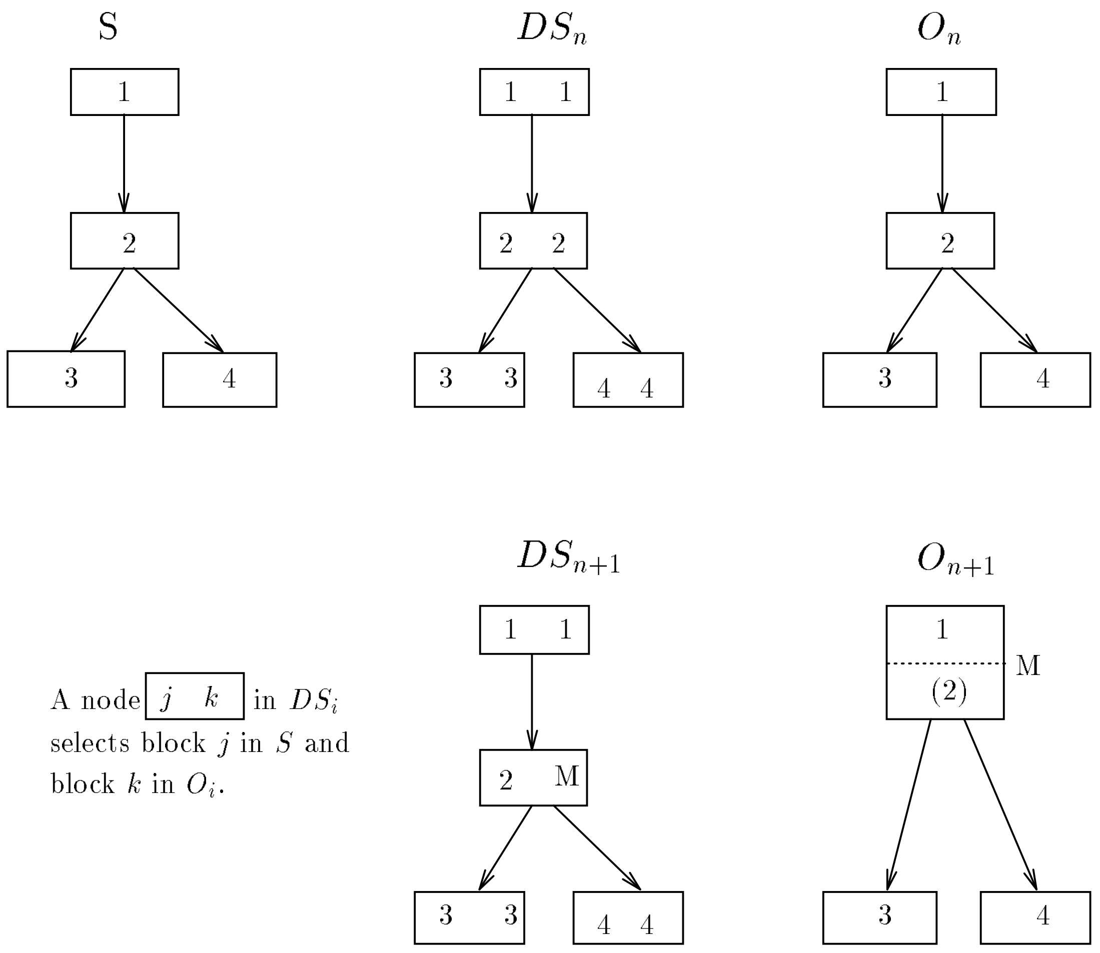

S
DS_n
O_n
DS_{n+1}
O_{n+1}
1
2
3
4
1 1
2 2
3 3
4 4
2 M
M
(2)
A node $\boxed{j \quad k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.4: Graph Transformation 4: coalescing two blocks.

$Add = \{(h, x) | (t, x) \in E^O\}$

$O_{i+1} = (V^O - \{t\}, (E^O - Del) \cup Add)$

Maintaining the semantics of a program imposes the following constraint on this transformation of the object graph: there is no input that causes a path through O_i containing the subpath $< h, x >$, for $x \neq t$, to be taken, and there is no input that causes a path through O_i containing the subpath $< x, t >$, for $x \neq h$, to be taken.

The DS-graph transformation:

Without adding or removing any nodes or edges, modify the selection of nodes so that a path through the DS-graph selects the right path through the object graph.

$DS_i = (V^{DS}, E^{DS})$

For $t \in V^{DS}$ such that $Oselects_i(t) = t$, let $Oselects_{i+1}(t) = M$

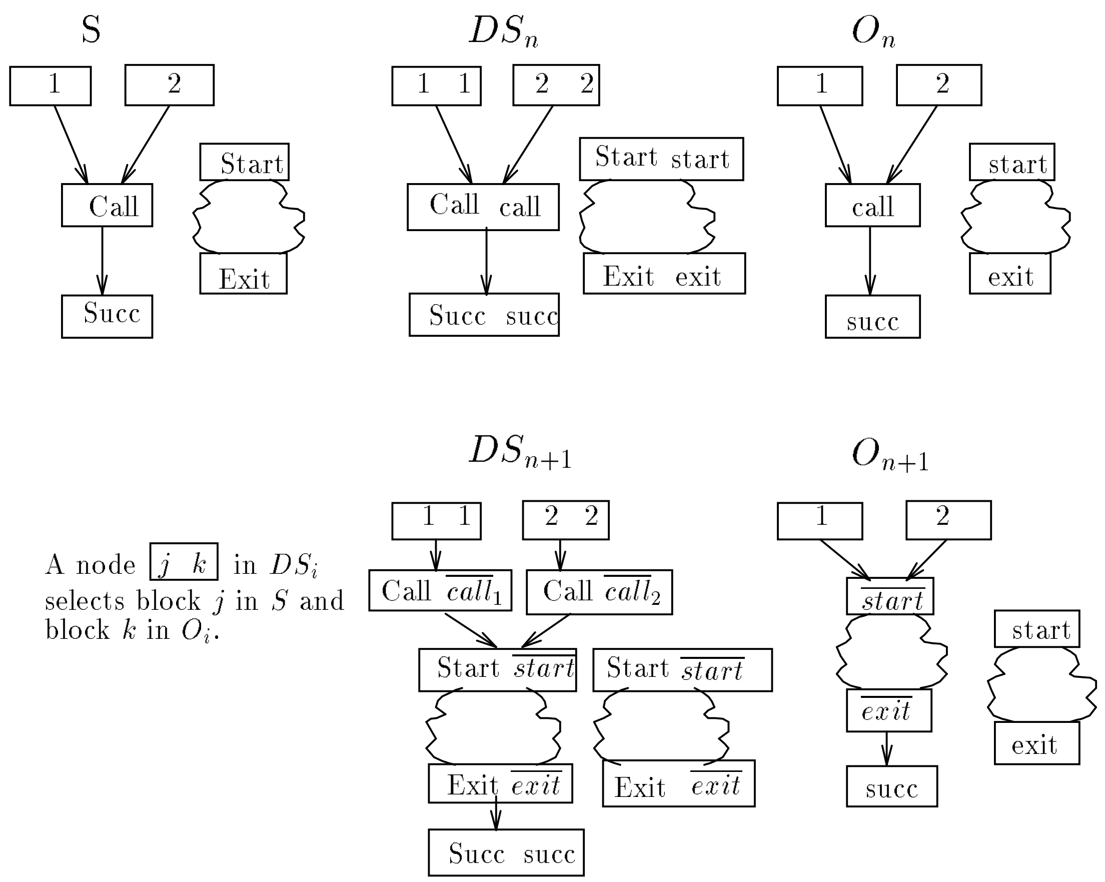
$DS_{i+1} = (V^{DS}, E^{DS})$

5. Inlining a subroutine.

   An example of this transformation is shown in Figure 17.5.

   The object graph transformation:

   The object graph must model a call as a basic block, and this exposition assumes that subroutines have single exits.



A node $\boxed{j\ \ k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.5: Graph Transformation 5: inlining a subroutine. The duplication of the call node is necessary to preserve DS-graph characteristics needed by other transformations.

Let *start* be the entry block of the subroutine being inlined, and *exit* be the exit block of the subroutine. Let *call* be the block containing the call, and *succ* be the successor of *call*. Because a call comprises a block, there can be no conditional branch and thus *succ* is the only successor of *call*.[25]

$O_i = (V^O, E^O)$

$Del = \{(x, call) | (x, call) \in E^O\} \cup \{(call, succ)\}$

$NewBlocks = \{\overline{n} | n = start$, or n is reachable from $start\}$

$Add = \{(\overline{h}, \overline{t}) | \overline{h} \in NewBlocks$ and $\overline{t} \in NewBlocks$ and $(h, t) \in E^O\} \cup$
$\{(x, \overline{start}) | (x, call) \in E^O\} \cup \{(\overline{exit}, succ)\}$

$O_{i+1} = ((V^O - \{call\}) \cup NewBlocks, (E^O - Del) \cup Add)$

The DS-graph transformation:

Like the object graph transformation, the DS-graph transformation copies a subgraph and grafts it in the place of the call. Unlike the object graph transformation, the DS-graph transformation keeps a copy of the call node on each path that enters the inlined routine.

[25]By transformation 4, *call* and *succ* could be coalesced. We can assume that call blocks and their successors are not coalesced prior to inlining the call, because the compiler would have to undo the coalescing in order to do the inlining. After inlining, of course, coalescing can take place.

Unlike the introduction of a new node in transformation 1, where the new node does not select a block in the source graph and has an empty definition list, a node created by this transformation does select a block in the source graph, and has the definition list of that block, modified so that a copied definition is not in the same equivalence class as the original.

Let *Start* be the entry node of the subroutine being inlined, and *Exit* be the exit node of the subroutine. Let *Call* be the node containing the call, and *Succ* be the (sole) successor of *Call*.

$DS_i = (V^{DS}, E^{DS})$

$DelEdges = \{(x, Call) | (x, Call) \in E^O\} \cup \{(Call, Succ)\}$

$InlinedNodes = \{\overline{n} | n = start$, or n is reachable from $Start\}$

$CallNodes = \{\overline{Call_i} | 0 \leq i \leq$ the in-degree of $Call\}$

$Oselects_{i+1}(\overline{n} \in InlinedNodes) = Oselects_i(n)$

$Oselects_{i+1}(\overline{Call_i} \in CallNodes) = Null$

$Sselects_{i+1}(\overline{n} \in InlinedNodes) = Sselects_i(n)$

$Sselects_{i+1}(\overline{n} \in CallNodes) = Sselects_i(Call)$

$NewEdges = \{(\overline{h}, \overline{t}) | \overline{h} \in InlinedNodes$ and $\overline{t} \in InlinedNodes$ and $(h, t) \in E^{DS}\} \cup$
$\{(\overline{Exit}, Succ)\} \cup \{(\overline{Call_i}, Start) | \overline{Call_i} \in CallNodes\} \cup$
$\{(x, \overline{Call_i}) | x$ is the i^{th} predecessor of $Call$ and $\overline{Call_i} \in CallNodes\}$

$DS_{i+1} = (V^{DS} \cup CallNodes \cup InlinedNodes, (E^{DS} - DelEdges) \cup NewEdges)$

6. Unrolling a loop.

An example of this transformation is shown in Figure 17.6.

The object graph transformation:

This exposition assumes that loops are structured so that a loop has a single successor and that there are no jumps into the middle of a loop. The transformation can be extended to loop structures that do not conform to these constraints.

Let a loop in the object graph be described by a set N_O^L of blocks with a few distinguished blocks: e is the loop entry and b is the bottom of the loop so that (b, e) is the back edge. All other blocks in N_O^L are within the loop. *succ* is the loop successor. The loop is unrolled I times, requiring I copies of blocks in N_O^L; these are distinguished by superscripts.

$O_i = (V^O, E^O)$

$N_O^L =$ the set of blocks in the loop, as described above.

$NewBlocks = \{\overline{n}_i | n \in N_O^L$ and $0 < i \leq I\}$

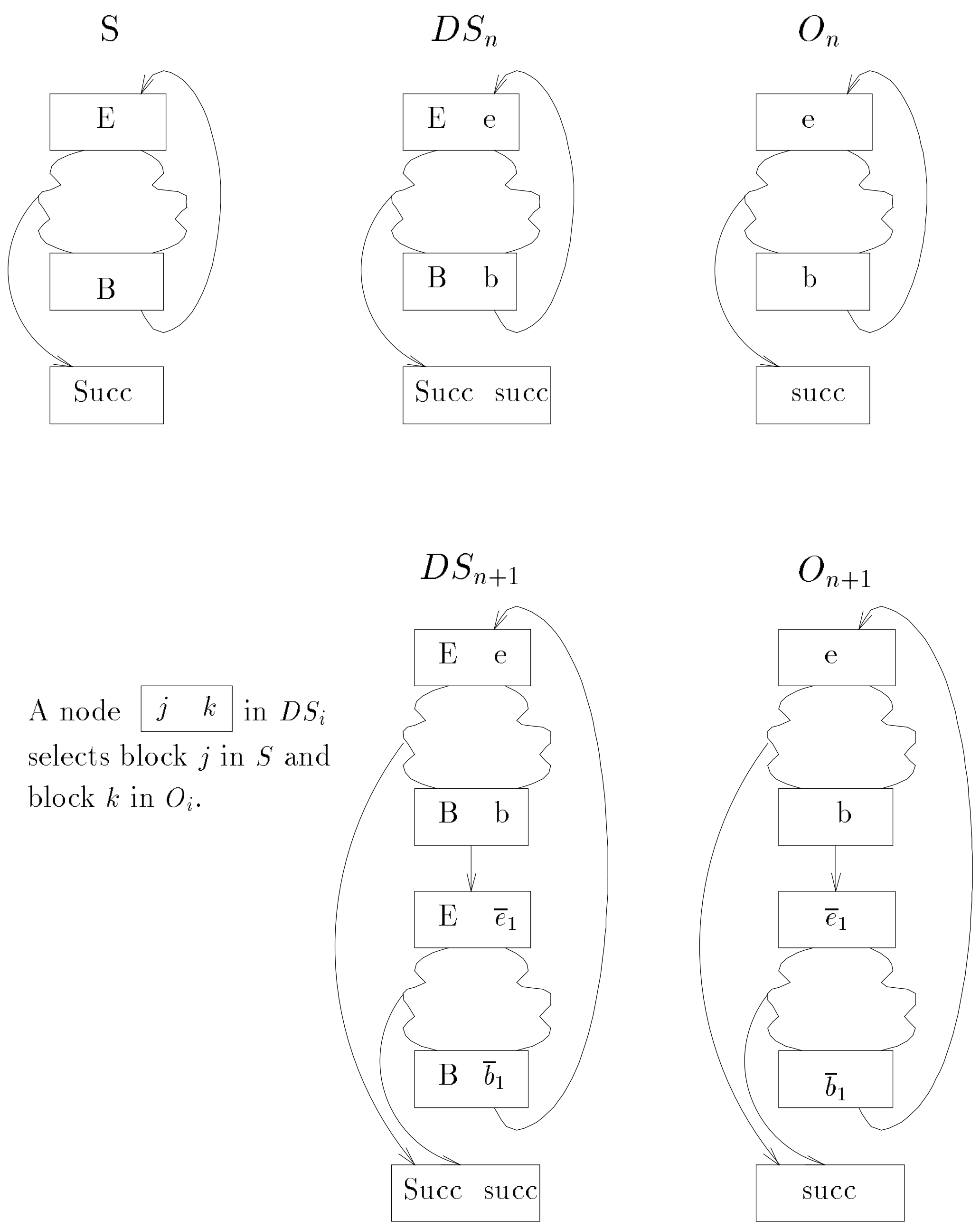
$Add = \{(\overline{h}_i, succ) | \overline{h}_i \in NewBlocks$ and $h \in N_O^L$ and $(h, succ) \in E^O\} \cup$
$\{(b, \overline{e}_1), (\overline{b}_I, e) | \overline{e}_1 \in NewBlocks$ and $\overline{b}_I \in NewBlocks\} \cup$
$\{(\overline{b}_i, \overline{e}_{i+1}) | 0 < i < I$ and $\overline{b}_i \in NewBlocks$ and $\overline{e}_{i+1} \in NewBlocks\}\} \cup$
$\{(\overline{h}_i, \overline{t}_i) | \overline{h}_i \in NewBlocks$ and $\overline{t}_i \in NewBlocks$ and $(h, t) \in N_O^L$ and $\overline{h}_i \neq \overline{b}_i\}$

$O_{i+1} = (V^O \cup NewBlocks, (E^O - \{(b, e)\}) \cup Add)$

The DS-graph transformation:

As with the object graph, when the loop is unrolled I times, I copies of the subgraph of the DS-graph comprising the loop are made.

Unlike the introduction of a new node in transformation 1, where a new node does not select a block in the source graph and has an empty definition list, a node created



A node $\boxed{j \quad k}$ in DS_i selects block j in S and block k in O_i.

Figure 17.6: Graph Transformation 6: unrolling a loop. The loop is unrolled once here. The unshown body of the loop may differ between the source graph, object graph, and DS-graph due to previous tranformations, but varies between O_n and O_{n+1}, and between DS_n and DS_{n+1}, only in the manner shown.

by this transformation does select a block in the source graph, and has the definition list of that block, modified so that a copied definition is not in the same equivalence class as the original.

A loop in the DS-graph is derived from a loop in the object graph. It is nonetheless necessary to name a few of the nodes within a loop: let E be the loop entry, B be the bottom node of the loop, and $Succ$ be the loop successor.

$DS_i = (V^{DS}, E^{DS})$

$N^L_{DS} = \{v_j | v_j \in V^{DS}$, ($< v_0, v_1, \ldots, v_k >$ is a subpath in DS_i
such that $Oselects_i(v_0) \in N^L_O$, $Oselects_i(v_k) \in N^L_O$, and
$Oselects_i(v_l) = Null$ for $0 < l < k)\}$, and $0 \leq j \leq k$

$NewNodes = \{\overline{n}_i | n \in N^L_{DS}$ and $0 < i \leq I\}$

$Top = \{\overline{E}_i | E$ is the loop entry and $0 < i \leq I\}$

$Bottom = \{\overline{B}_i | B$ is the bottom node of the loop and $0 < i \leq I\}$

$NewEdges = \{(\overline{h}_i, Succ) | \overline{h}_i \in NewNodes, h \in N^L_{DS}$, and $(h, Succ) \in E^{DS}\} \cup$
$\{(B, \overline{E}_1), (\overline{B}_I, E) | \overline{E}_1 \in Top$ and $\overline{B}_I \in Bottom\} \cup$
$\{(\overline{B}_i, \overline{E}_{i+1}) | 0 < i < I, \overline{B}_i \in Bottom$, and $\overline{E}_{i+1} \in Top\} \cup$
$\{(\overline{h}_i, \overline{t}_i) | \overline{h}_i \in NewNodes, \overline{t}_i \in NewNodes, (h,t) \in N^L_{DS}$, and $\overline{h}_i \neq \overline{B}_i\}$

$Oselects_{i+1}(\overline{n} \in NewNodes) = Oselects_i(n)$

$Sselects_{i+1}(\overline{n} \in NewNodes) = Sselects_i(n)$

$DS_{i+1} = (V^{DS} \cup NewNodes, (E^{DS} - \{(B, E)\}) \cup NewEdges)$

7. No optimization other than those described in one of the other object graph transformations may modify control flow in a way that changes which block is entered on a particular input. See Section 21 for a mechanism that may allow truthful (but not expected) behavior in the presence of optimizations that violate this constraint.

Given a valid DS-graph, if any of these modifications are performed, the result is a valid DS-graph. This is shown in Appendix B. For this currency determination technique to work, optimizations that modify the shape of the flow graph may do so only in ways allowed by these rules. Optimizations that change control flow are defined (by Rule 7) to modify the shape of the flow graph. It follows that optimizations that do not modify the shape of the flow graph preserve the validity of the DS-graph.

Many optimizations modify the flow graph only in ways that can be modelled by these DS-graph modifications, but arbitrary optimization is not modelled. For example, recognizing bubblesort in the unoptimized version and replacing it with quicksort in the optimized version clearly involves graph modifications beyond the scope of these transformations. Loop interchange is disallowed because it changes the order in which blocks are entered.

Leaving such high-level optimization aside, consider simply adding an edge to the flow graph. This must be reflected in the DS-graph, but an edge in the DS-graph denotes potential execution in both the optimized and unoptimized version. If an edge could be added to the DS-graph arbitrarily, a path resulting from its introduction might select a sequence of blocks in the source graph that is not a path through the source graph. Information about definitions that reach along such a path through the DS-graph would be wrong.

The apparent introduction of such a new path by an optimization may be an artifact of how an optimization is modelled. For example, cross-jumping can be implemented by introducing a jump. Assume blocks A and B share successors, that A contains the code sequence pq, and that B contains the code sequence q. A jump to B can be inserted following

p in A. This could be modelled as eliminating q in A, deleting the original edges out of A, and adding an edge from A to B. It cannot be modelled that way by iterative application of the graph-modification rules. It could also be modelled as introducing a new block C whose predecessors are A and B and whose successors are the successors of A and B, and moving q into C (from both A and B). It can be modelled in this fashion by iterative application of the graph-modification rules.[26]

A valid DS-graph is the graph on which the dataflow computation is performed. We turn now to the nature of that computation.

17.4 Dataflow on DS-Pairs

The Gen DS-Pair

Analogous to the Gen set of standard data-flow algorithms, a Gen ds-pair contains information about what is generated in a block. In the absence of aliasing, there is exactly one Gen ds-pair for a given variable in each block. The Gen ds-pair for a variable V and a block B is written Gen_B^V. Gen_B^V is (d, s), where d is the last definition of V on B's definition list or *null* if there is none, and s is the last store into V on B's store list or *null* if there is none.

Only Gen ds-pairs may have null entries. Ds-pairs with null entries do not appear in the In or Out sets of a block or in PRS_B^V.

The κ Operation

In a standard dataflow computation, a definition in a block kills definitions (of the same variable) that reach the entry of the block. In the currency determination dataflow computation given in Algorithm PRS, definitions kill definitions, and stores kill stores. If $\text{Gen}_B^V\text{.d} = d$, ds-pairs reaching the exit of B contain d as their definition element. If $\text{Gen}_B^V\text{.d} = \mathit{null}$, ds-pairs reaching the exit of B contain the same definition element they had at the entry of B. Stores are handled analogously. This is represented with the operator κ:

Definition 17.3: $(e, t)\,\kappa\,(\mathit{null}, \mathit{null}) = (e, t)$

$$(e, t)\,\kappa\,(d, \mathit{null}) = (d, t)$$
$$(e, t)\,\kappa\,(\mathit{null}, s) = (e, s)$$
$$(e, t)\,\kappa\,(d, s) = (d, s)$$

The $\overset{\kappa}{\cup}$ Operation

Multiple ds-pairs may reach the entry of a block. The $\overset{\kappa}{\cup}$ operation defines the ds-pairs that reach the exit of a block, given all of the pairs that reach block entry. The $\overset{\kappa}{\cup}$ operation takes a set (the In set for a block) as its left operand, whereas the κ operation takes a single ds-pair as its left operand. If the block generates both a store and a definition, its Out set will contain the ds-pair consisting of that store and definition, and if it contains a null in either position (or both), its Out set depends on the In set. Because every variable is defined

[26]There is an independent argument for representing cross-jumping this way: B might contain the sequence oq, requiring a jump from the middle of A to the middle of B. The second method of modelling cross-jumping can model this, the first cannot.

to have an initial definition and store, propagation will eventually cause any reachable block to have a nonempty In set.

If the In set is empty, the Out set is the set of complete ds-pairs (those containing no *nulls*) in the Gen set, and if the In set is not empty, the Out set is the set of ds-pairs produced by individual κ operations between each ds-pair in R and the ds-pair S:

Definition 17.4: $$Complete(S) = \begin{cases} \emptyset & \text{if } d = null \text{ or } s = null \\ (d, s) & \text{otherwise} \end{cases}$$

Definition 17.5: $$R \overset{\kappa}{\cup} S = \begin{cases} Complete(S) & \text{if } R = \emptyset \\ \{r \,\kappa\, S | r \in R\} & \text{otherwise} \end{cases}$$

17.5 Paired Reaching Sets at Block Boundaries

Algorithm PRS computes paired reaching sets at block boundaries for a DS-graph component (a subroutine). A source node, *Start*, is grafted on to the component to provide a place for initial definitions (*d-init*) and stores (*s-init*) representing the creation of variables. The algorithm consists of an initialization step and an iterative step. (If the DS-graph component is a subroutine, line 1 of Initialize should set Gen^V_{Start} to (*d-incoming*,*s-incoming*) when V is a parameter.)

Initialize
Input:
a component of the DS-graph, modified by the addition of a *Start* node;
Output:
the Gen sets of each variable for each block.
Step 1:

```
0  for each variable $V$
1      $\text{Gen}^V_{Start} = (d\text{-}init, s\text{-}init)$
2      for each node $B$ other than $Start$
3          set $\text{Gen}^V_B.d$ to the last definition of $V$ in the definition list of $B$
               or to $null$ if there is none
4          set $\text{Gen}^V_B.s$ to the last store into $V$ in the store list of $B$
               or to $null$ if there is none
```

End of Initialize

Iterate
Input:
a component of the DS-graph, modified by the addition of a *Start* node,
the Gen sets of each variable for each block;
Output:
the paired reaching sets of each variable at each block boundary.

```
0  for each variable $V$
1      for each block $B$
2          $\text{In}^V_B = \text{Out}^V_B = \emptyset$
3      iteratively compute $\text{In}^V_B$ and $\text{Out}^V_B$ until convergence, according to the following,
4          for each block $B$
5              $\text{In}^V_B = \bigcup_P \text{Out}^V_P$ for $P$ predecessors of $B$
6          for each block $B$
```

```
7           $\text{Out}_B^V = \text{In}_B^V \overset{\kappa}{\cup} \text{Gen}_B^V$
```
End of Iterate

Algorithm PRS

```
Input:
    a component of the DS-graph, modified by the addition of a *Start* node,
Output:
    the paired reaching sets of each variable at each block boundary.
0   Initialize
1   Iterate
```
End of Algorithm PRS

17.6 Paired Reaching Sets at Breakpoints

Algorithm PRS provides In and Out sets at block boundaries. Our goal is to determine a variable's currency at a breakpoint, which may be in the middle of a block rather than at a block boundary. The definitions of current, noncurrent, and endangered refer to 'a breakpoint B' that lies on a path-pair. This is well-defined for syntactic breakpoints, but not for semantic breakpoints whose representative instruction has been moved by optimization, so we compute paired reaching sets at syntactic breakpoints only. The consequences of this are that a block containing the breakpoint is present in both the source and the object graphs, and the breakpoint is on both the definition list and the store list for some block in the DS-graph. Note that for a syntactic breakpoint for a statement that has been moved or eliminated, the element on the store list representing the breakpoint is the representative instruction for a following statement.

PRS_{Bk}^V, the set of ds-pairs relevant to V that reach a breakpoint Bk, is derived from the In sets computed by Algorithm PRS.

Initialize-BK

```
Input:
    a variable $V$, a syntactic breakpoint $Bk$, and the node $B$ containing $Bk$;
Output:
    The Gen set of $V$ for $B$ at $Bk$;
0   Set $\text{Gen}_{Bk}^V.d$ to the last definition of $V$ prior to $Bk$ on $B$'s definition list,
        or to *null* if there is none.
1   Set $\text{Gen}_{Bk}^V.s$ to the last store into $V$ prior to $Bk$ on $B$'s store list,
        or to *null* if there is none.
```
End of Initialize-BK

Algorithm PRS-BK

```
Input:
    a variable $V$, a syntactic breakpoint $Bk$, the node $B$ containing $Bk$, and $\text{In}_B^V$;
Output:
    The paired reaching set of $V$ at $Bk$;
0   Initialize-BK
1   $\text{PRS}_{Bk}^V = \text{In}_B^V \overset{\kappa}{\cup} \text{Gen}_{Bk}^V$
```

End of Algorithm PRS-BK

17.7 A Variable's Currency

The contents of PRS_{Bk}^{V} tell us V's currency:

Theorem 17.6: $\mathrm{PRS}_{Bk}^{V} = \emptyset$ iff either V is not in scope at Bk or Bk is unreachable. Otherwise:

V is current at Bk iff $\forall (d,s) \in \mathrm{PRS}_{Bk}^{V}$, s was generated from d;

V is endangered at Bk iff $\exists (d,s) \in \mathrm{PRS}_{Bk}^{V}$ such that s was not generated from d;

V is noncurrent at Bk iff $\nexists (d,s) \in \mathrm{PRS}_{Bk}^{V}$ such that s was generated from d.

Theorem 17.6 is proven in Appendix C.1.

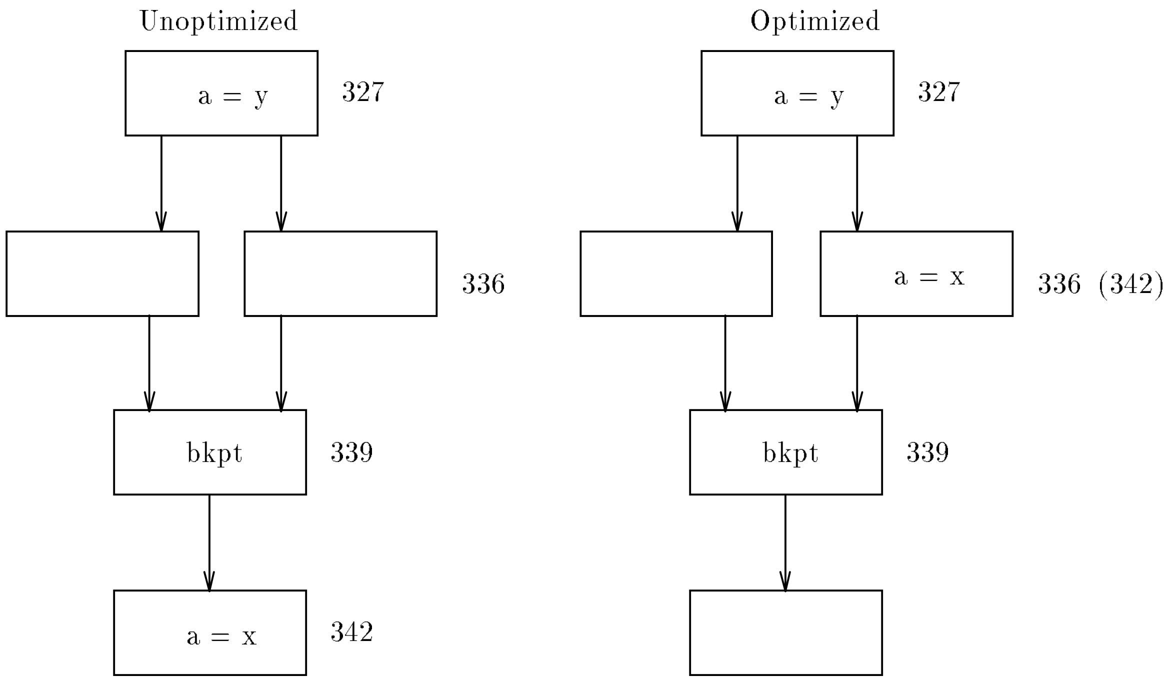
18 When a Variable is Endangered

When the debugger is asked to display a variable, it determines whether the variable is current. If the variable is current, the debugger displays its value without comment. If the variable is endangered, in addition to displaying its value, the debugger can give the user some help in understanding why the value is endangered. The general flavor of what the debugger can do is given by the following sample message that might accompany the display of variable `a` when the optimization shown in Figure 18.1 has occurred.

> "Breakpoint 1 has been reached at line 339. `a` should have been set at line 327. However, optimization has moved the assignment to `a` at line 342 to near line 336. `a` was actually set at one of lines 327 or 342."

The information contained in this message is available from the paired reaching set PRS^{a}_{339} and the unoptimized and optimized flow graphs. The description of the effects of optimization will vary in specificity as the effects of optimization vary in complexity.

Figure 18.1: The display of `a` could be accompanied by this message: "Breakpoint 1 has been reached at line 339. `a` should have been set at line 327. However, optimization has moved the assignment to `a` at line 342 to near line 336. `a` was actually set at one of lines 327 or 342."



19 Transparency

19.1 Assignments Through Aliases

Consider an assignment $*P$ through a pointer (or through an array element where the index is a variable) that could point to V. When execution is suspended at a breakpoint B, $*P$ may be an alias for V. $*P$ must be considered to be a definition of V that reaches B. If $*P$ is not an alias for V in some particular execution, the value that V contains at the breakpoint came from whatever definition would have reached if $*P$ were not present. Therefore, this definition must also be considered to reach B. For any language that allows such aliasing, the assumption of a single definition reaching along a given path-pair does not hold.

If there are multiple definitions of V that reach B along p, all of them but one (the one furthest from B on p) must be assignments through aliases, because other kinds of assignments kill prior definitions. An assignment through an alias is defined as such by its ambiguity about whether V is assigned into, because if it can be determined that an assignment through a pointer does assign into V every time, that assignment kills prior definitions, and if it can be determined that an assignment through a pointer never assigns into V, the assignment is not a definition of V.

19.2 Definitions and Stores Revisited

The computation of paired reaching sets depends on the definition of κ given in Sections 17.4, which in turn depends on the assumption that a data object assigned a value within a basic block is unconditionally affected by the assignment. This plays out as an assumption that definitions kill definitions and stores kill stores. This assumption holds for direct assignments but not for assignments through aliases. Because we have two kinds of assignments with different characteristics, we need two kinds of definitions and stores. Definitions and stores that are unambiguous as to the variable that is affected we call *opaque*. Definitions and stores that affect one of a set of variables such that it cannot be determined at compile time which variable will be affected, we call *transparent*. Thus `*p = 0;` is a transparent definition of V unless the compiler determines that `p` cannot point to V at that assignment. V is one member of the set of variables that might be affected by the definition. `p` is called a *transparent definer*.

It may be that `p` never points to V, but the compiler cannot determine that. The debugger can do no better than the compiler, and we must treat assignments through `p` as if they can affect V.

It can be shown that a compiler can make an assignment through a pointer that is not current without violating program semantics. An example is given in Appendix A. The circumstances under which a compiler can do so are sufficiently constrained that for the remainder of this work, I assume that transparent definers are current at each of their uses.

19.3 κ Revisited

The introduction of transparent definitions requires a redefinition of the κ operator so that transparent definitions and stores do not kill elements that should reach subsequent points in the program. The right operand of κ is doing the killing, so we do not need to worry about the transparency of the left operand. We could redefine κ by independently

$(e,t)\,\kappa\,(d,s)$	d is null	d is opaque	d is transparent
s is null	(e,t)	(d,t)	(e,t) (d,t)
s is opaque	(e,s)	(d,s)	(e,s) (d,s)
s is transparent	(e,t) (e,s)	(d,t) (d,s)	(e,t) (d,s) (e,s) (d,t)

Table 19.1: This redefinition of κ overgenerates ds-pairs.

letting an opaque definition of V kill other definitions of V, transparent and opaque alike, and letting an opaque store into V kill stores into V, transparent and opaque alike, while not letting a transparent definition of V kill definitions of V and not letting a transparent store into V kill stores into V. Such a definition is given in Table 19.1. However, this overgenerates ds-pairs.

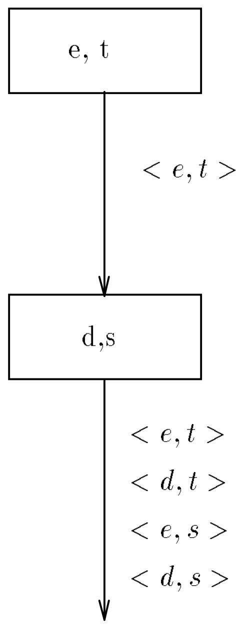
As a motivating example, suppose there is a block B containing the definition d whose source code is `*p = x`, where `p` may point to V and the store s generated from d has survived the optimizer without being moved or eliminated. Gen_B^V is (d,s), and d and s are transparent. Assume that no optimization has affected assignments into `p`. Then on a given execution, either V is affected (`p` points to V) or V isn't affected (`p` doesn't point to V). In this case, although $(d,t) \in (e,t)\kappa(d,s)$, there is no input on which d reaches the exit of B in the unoptimized version and t reaches the exit of B in the optimized version. This is illustrated in Figure 19.1.

In general, let s be a store through some pointer, d be the definition that generated s, and B be a breakpoint. If in every execution in which s can affect V at B, d reaches B, then at B, any ds-pair (d,x) where $x \neq s$ or (x,s) where $x \neq d$ is called *infeasible*. Such a ds-pair does not represent a definition that reaches B in the unoptimized code and the store that reaches B on the same input in the optimized code. The definition of κ in Table 19.1 places infeasible ds-pairs in paired reaching sets, and its use would result in conservative errors in the results.

19.4 Constrained Transparency: Transparent Assignments Without Elimination or Code Motion

Many compilers do not do sufficient pointer analysis to optimized transparent assignments. Suppose transparent assignments are present but are not optimized, that is, no elimination or motion of transparent assignments occurs. Consequently, if a block contains a transparent assignment, both the associated definition and the store it generated are in the block.

Assume $\text{Gen}_B^V = (d,s)$ and $\text{In}_B^V = \{(e,t)\}$. If d and s are transparent, neither d nor s may be null, and s was generated from d. The transparent definition may affect V, so Out_B^V must contain (d,s). The transparent definition may not affect V, so Out_B^V must contain (e,t), regardless of whether e or t are transparent. If d and s are not transparent, the original definition of κ is correct even if e or t is transparent. Table 19.2 redefines κ



```
p = &V;

...

if (cond) {

   p = &W;

   }

...

*p = x; // This is d.
```

Figure 19.1: e is a definition of V and t is a store into V. Whether d is a definition of V and s is a store into V depends on which assignment to `p` reaches the assignment through `p`. (d,t) is an infeasible ds-pair because d defines V only when s stores into V, in which case s kills t. (e,s) is an infeasible ds-pair for similar reasons.

$(e,t)\,\kappa\,(d,s)$	d is null	d is opaque	d is transparent
s is null	(e,t)	(d,t)	
s is opaque	(e,s)	(d,s)	
s is transparent			(e,t) (d,s)

Table 19.2: $(e,t)\,\kappa\,(d,s)$: This definition assumes that no elimination or motion of transparent assignments may occur. No infeasible ds-pairs are generated.

by combining these two cases. Blank entries in the table represent combinations that by assumption cannot occur. No infeasible ds-pairs are generated under this definition of κ.

$\overset{\kappa}{\cup}$ in the Presence of Constrained Transparency

The definition of $\overset{\kappa}{\cup}$ must change because a block may contain many definitions of a variable V. Only the last opaque definition of V reaches the exit of the block, but all subsequent transparent definitions of V also reach the exit of the block. All of these belong in the Gen set. If the In set is empty, the Out set is the set of complete ds-pairs in the Gen set. If the Gen set contains a complete opaque ds-pair, again the Out set is the set of complete ds-pairs in the Gen set. Otherwise, the Out set is the set of ds-pairs produced by individual κ operations between each ds-pair in the In set and each ds-pair in the Gen set. We define $\overset{\kappa}{\cup}$ accordingly:

Definition 19.1: $Complete(S) = \{(d,s) \in S | d \neq null \text{ and } s \neq null\}$

Definition 19.2: $$R \overset{\kappa}{\cup} S = \begin{cases} \mathit{Complete}(S) & \text{if } R = \emptyset \\ \mathit{Complete}(S) & \text{if } \exists e \in \mathit{Complete}(S) \text{ such that } e \text{ is opaque} \\ \{r \,\kappa\, s \mid r \in R, s \in S\} & \text{otherwise} \end{cases}$$

Currency in the Presence of Constrained Transparency

Here I present algorithms modified to handle constrained transparency. Most of the additional work has been incorporated into the definitions of κ and $\overset{\kappa}{\cup}$. The iterative step of Algorithm PRS works without modification, but the initialization step must construct Gen sets that may contain multiple ds-pairs.

```
Initialize^CT
Input:
    a component of the DS-graph, modified by the addition of a Start node;
Output:
    the Gen sets of each variable for each block.
0   for each variable $V$
1       $\mathrm{Gen}^V_{Start} = (d\text{-}init, s\text{-}init)$
2       for each node $B$ other than $Start$
3           set $d$ to the last opaque definition of $V$ in the definition list of $B$
                or to $null$ if there is none
4           set $s$ to the last opaque store into $V$ in the store list of $B$
                or to $null$ if there is none
5           add $(d, s)$ to $\mathrm{Gen}^V_B$
6           set $d$ to the next definition of $V$ in the definition list of $B$
                and set $s$ to the next store into $V$ in the store list of $B$
7           add $(d, s)$ to $\mathrm{Gen}^V_B$
8           iterate steps 6 and 7 until the end of the definition and store lists are reached
End of Initialize^CT
```

```
Algorithm PRS^CT
Input:
    a component of the DS-graph, modified by the addition of a Start node,
Output:
    the paired reaching sets of each variable at each block boundary.
0   Initialize^CT
1   Iterate
End of Algorithm PRS^CT
```

Similarly, the initialization step of Algorithm PRS-BK^{CT} must construct Gen sets that may contain multiple ds-pairs.

```
Initialize-BK^CT
Input:
    a variable $V$, a syntactic breakpoint $Bk$, and the node $B$ containing $Bk$;
Output:
    The Gen set of $V$ for $B$ at $Bk$;
0   set $d$ to the last opaque definition of $V$ prior to $Bk$ on $B$'s definition list,
        or to *null* if there is none
1   set $s$ to the last opaque store into $V$ prior to $Bk$ on $B$'s store list,
        or to *null* if there is none
2   add $(d, s)$ to $\mathrm{Gen}_B^V$
3   for each subsequent transparent definition $d$ of $V$ prior to $Bk$ on $B$'s definition list,
        and each subsequent transparent store $s$ into $V$ prior to $Bk$ on $B$'s store list,
4       add $(d, s)$ to $\mathrm{Gen}_B^V$
End of Initialize-BK$^{CT}$
```

```
Algorithm PRS-BK$^{CT}$
Input:
    a variable $V$, a syntactic breakpoint $Bk$, the node $B$ containing $Bk$, and $\mathrm{In}_B^V$;
Output:
    The paired reaching sets of $V$ at $Bk$;
0   Initialize-BK$^{CT}$
1   $\mathrm{PRS}_{Bk}^V = \mathrm{In}_B^V \overset{\kappa}{\cup} \mathrm{Gen}_{Bk}^V$
End of Algorithm PRS-BK$^{CT}$
```

As before, the contents of PRS_{Bk}^V tell us V's currency. Theorem 17.6 holds in the presence of constrained tranparency. It is proven in Appendix C.1.

19.5 Unconstrained Transparency: Elimination and Code Motion of Transparent Assignments

Suppose we allow elimination and motion of transparent assignments. Now a transparent ds-pair in a Gen set may contain nulls. Furthermore, a block may contain a definition d and a store s not generated from d. As a consequence, there are circumstances in which any of the ds-pairs produced by the definition of κ given in Table 19.1 are feasible. However, as described in Section 19.3 and shown in Figure 19.1, sometimes some of those ds-pairs are infeasible. Figure 19.2 exemplifies the impossibility of producing only feasible ds-pairs in the presence of unconstrained transparency with the mechanisms presented so far. It is possible to distinguish the circumstances in which κ will produce an infeasible ds-pair from the circumstances in which it will produce a feasible ds-pair, but only with knowledge of which definitions or stores are generated along the same path—information not available to κ.

To capture this further information, ds-pairs are extended to ds-list-pairs—pairs of lists rather than pairs of elements. The definition element in a ds-list-pair for a variable V is the list of definitions of V that reach a block B along a path p, in the order that they occur on p. The store element is the list of stores into V that reach B along p, similarly ordered. It follows that the first element in each list is opaque and the rest are transparent. Figure 19.3 is the example from Figure 19.2 using ds-list-pairs instead of ds-pairs. .

The data that is the input to κ has been modified to contain the information it was lacking, and in the process, it has become necessary to redefine κ to work on the new form of the data. The list operations are the same for definitions and stores, and are performed

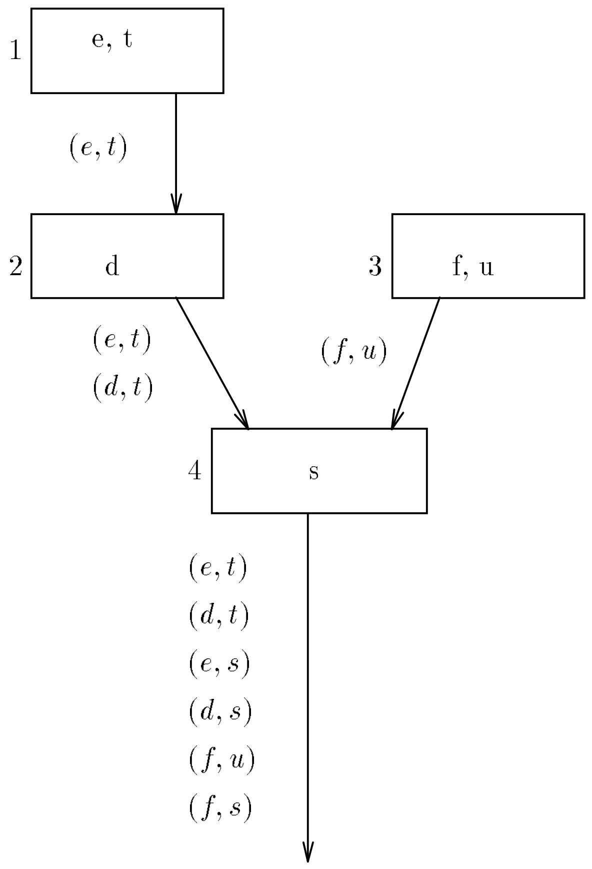


Figure 19.2: If transparent store s is not generated from transparent definition d, all ds-pairs in the Out set of node 4 are feasible. If s is generated from d, (d,t) and (e,s) are infeasible. However, (f,s) is feasible. The salient difference between (e,s) and (f,s) is that e and d reach node 4 along the same path, while f and d do not.

independently on definitions and stores, so I introduce a list operator ℓ. The empty list is expressed as *null*, and $|$ is used as a list concatenation operator. Given two lists x and y:

Definition 19.3:
$$x \,\ell\, y = \begin{cases} x & \text{if } y = null \\ y & \text{if an element of } y \text{ is opaque} \\ x|y & \text{if all elements of } y \text{ are transparent} \end{cases}$$

Then given two ds-list-pairs (e_l, t_l) and (d_l, s_l):

Definition 19.4: $(e_l, t_l)\kappa(d_l, s_l) = (e_l \,\ell\, d_l, t_l \,\ell\, s_l)$

With multiple assignments encoded in a single ds-list-pair, we return to a situation in which the Gen set for a block is a single entity.

$\overset{\kappa}{\cup}$ in the Presence of UnConstrained Transparency

Complete must be redefined to act on ds-list-pairs:

Definition 19.5:
$$Complete((d_l, s_l)) = \begin{cases} \emptyset & \text{if } d_l \text{ or } s_l \text{ is } null \\ (d_l, s_l) & \text{otherwise} \end{cases}$$

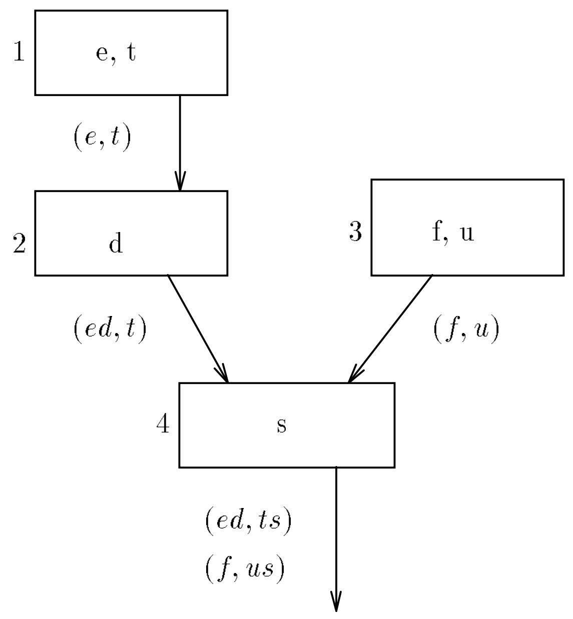


Figure 19.3: This is the example from Figure 19.2 recast to use ds-list-pairs. Encoded in the ds-list-pairs is the fact that e and d reach node 4 along the same path while f and d do not.

Definition 19.4 defines κ as an append operation on transparent definitions. Repeated κ operations on a loop containing only transparent definitions of V result in lists that grow indefinitely, as the definitions and stores within the loop get appended ad infinitum.

The purpose of ds-list-pairs is to capture for any particular path those transparent definitions (or stores) that might reach a breakpoint along that path. In any particular execution, exactly one definition (and store) actually reaches along a particular path (i.e., the last transparent or opaque definition that actually refers to V). Intuitively then, we do not need to record all transparent definitions since the last opaque definition but instead needs to record which one could be last, second to last (in case the last one doesn't actually refer to V), etc. For example, if a definition could be last and n^{th} from last and it actually points to V, then it is the last one that reaches (not the n^{th} from last). More generally, if it could be m^{th} from last and n^{th} from last, where $m < n$, then the m^{th} from last definition will reach the breakpoint provided the definitions following it on that path do not actually refer to V. This is captured formally in Lemma C.12.

It follows from this intuition that we can construct ds-list-pairs that do not grow indefinitely by including only the last occurrence of any duplicated definitions and stores. There are cases in which information about currency is lost unless the last two instances of a definition or store are included in the ds-list-pairs. In these cases, the choice is between an algorithm that has the potential for non-conservative error (in which an endangered variable may be reported as current) and a more complex algorithm that has greater potential for conservative error (a current variable may be reported as endangered). These cases are discussed in Section 20. For reasons presented in that section, I have chosen the former of these alternatives, thus the algorithm presented below has the potential for non-conservative error.

The ds-list-pair construction method given below preserves the characteristic that the i^{th} store in a ds-list-pair is generated from the i^{th} definition in that ds-list-pair for all i if and only if V is current along the path from which that ds-list-pair is derived, except in the cases discussed in Section 20 (this is shown in Appendix C).

Definition 19.6: Let $x_l =< x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_n >$ where x_i and x_j, are in the same equivalence class. Then

$last(x_l) =< x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_j, \ldots, x_n >$, and

$last^*(x_l)$ is the result of applying *last* to x_l repeatedly until each equivalence class is represented at most once.

$Last(d_l, s_l) = (last^*(d_l), last^*(s_l))$

$$R \overset{\kappa}{\cup} S = \begin{cases} Last(Complete(S)) & \text{if } R = \emptyset \\ Last(Complete(S)) & \text{if } \exists e \in Complete(S) \text{ such that } e \text{ is opaque} \\ \{Last(r \,\kappa\, s) | r \in R, s \in S\} & \text{otherwise} \end{cases}$$

19.6 Currency in the Presence of Unconstrained Transparency

Here I present algorithms modified to handle unconstrained transparency. Again, most of the additional work has been incorporated into the definition of $\overset{\kappa}{\cup}$, so only the initialization phase of each algorithm needs modification.

Initialize^T
Input:
  a component of the DS-graph, modified by the addition of a *Start* node;
Output:
  the Gen sets of each variable for each block.

```
0  for each variable $V$
1     $\text{Gen}^V_{Start} = (d\text{-}init, s\text{-}init)$
2     for each node $B$ other than $Start$
3        $d_l = null$
4        for each definition $d$ of $V$ in $B$'s definition list, in order of appearance
5           $d_l = d_l \,\ell\, d$
6        $s_l = null$
7        for each store $s$ into $V$ in $B$'s store list, in order of appearance
8           $s_l = s_l \,\ell\, s$
9        $\text{Gen}^V_B = (d_l, s_l)$
```
End of Initialize^T

Algorithm PRS^T
Input:
  a component of the DS-graph, modified by the addition of a *Start* node,
Output:
  the paired reaching sets of each variable at each block boundary.

```
0  Initialize$^T$
1  Iterate
```
End of Algorithm PRS

Initialize-BK^T
Input:

```
    a variable $V$, a syntactic breakpoint $Bk$, and the node $B$ containing $Bk$;
Output:
    The Gen set of $V$ for $B$ at $Bk$;
0   $d_l = null$
1   for each definition $d$ of $V$ prior to $Bk$ on $B$'s definition list, in order of appearance
2       $d_l = d_l \,\ell\, d$
3   $s_l = null$
4   for each store $s$ into $V$ prior to $Bk$ on $B$'s store list, in order of appearance
5       $s_l = s_l \,\ell\, s$
6   $\mathrm{Gen}^V_{Bk} = (d_l, s_l)$
```
End of Initialize-BK^T

Algorithm PRS-BK^T

```
Input:
    a variable $V$, a syntactic breakpoint $Bk$, the node $B$ containing $Bk$, and $\mathrm{In}^V_B$;
Output:
    The paired reaching sets of $V$ at $Bk$;
0   Initialize-BK$^T$
1   $\mathrm{PRS}^V_{Bk} = \mathrm{In}^V_B \overset{\kappa}{\cup} \mathrm{Gen}^V_{Bk}$
```
End of Algorithm PRS-BK^T

Once again, the contents of PRS^V_{Bk} tell us V's currency. Excepting the cases discussed in Section 20:

- The i^{th} store in a ds-list-pair is generated from the i^{th} definition in that ds-list-pair if and only if V is current along the path from which that ds-list-pair is derived.
- Each ds-list-pair e tells us whether V is current along the set of paths from which e is derived.
- All paths are represented by some ds-list-pair in PRS^V_{Bk}.

In other words, PRS^V_{Bk} contains the interesting ds-list-pairs, that is, PRS^V_{Bk} contains a set of ds-list-pairs that is (nearly) sufficient to determine V' currency.

Definition 19.7: *V is current at Bk by ds-list-pair e* iff V is current along each path p to Bk such that e is is derived from p.

The following theorem does not hold in the cases discussed in Section 20.

Theorem 19.8: $\mathrm{PRS}^V_{Bk} = \emptyset$ iff either V is not in scope at B or B is unreachable. Otherwise:

V is current at Bk iff V is current at Bk by e, $\forall e \in \mathrm{PRS}^V_{Bk}$;

V is endangered at Bk iff $\exists e \in \mathrm{PRS}^V_{Bk}$ such that V is not current at Bk by e;

V is noncurrent at Bk iff $\nexists e \in \mathrm{PRS}^V_{Bk}$ such that V is current at Bk by e.

In Appendix C.2, Theorem 19.8 is proven for most cases. It is shown how it fails for some exceptional cases, which are discussed below.

20 Cases in which the Algorithm May Err

Consider the case in which a loop-invariant transparent assignment to V is moved to a loop pre-header. Let the loop be L, the invariant definition of V in L be dx and the store generated from dx be sx. (In general, I use the naming convention that a definition dx generates the store sx.) Let p be a path to a breakpoint Bk (where Bk is after L) that contains two or more iterations of L. Although the argument made here does not depend on it, for simplicity of presentation assume that no other transparent assignments to V reach Bk. Then the sequence of definitions of V that reach Bk along p is $< d_1, d_2, d_3, \ldots d_n >$ where there are $n-1$ interations of L in p, and d_2 through d_n are instances of dx. The sequence of stores into V that reach Bk along p is $< s_1, s_2 >$ where s_2 is sx, in the pre-header. d_1 and s_1 are the opaque definition of V and store into V that reach Bk from above the loop.

Assuming that s_1 is generated from d_1, V is current at Bk: either s_1 qualified reaches Bk with d_1 or s_2 qualified reaches Bk with d_n. (s_2 cannot qualified reach with d_i, $1 < i < n$, because that would turn off d_n, which would turn off not only s_2 but also d_i.) This case is handled correctly—ds-list-pair ($< d_1, dx >, < s_1, sx >$) is derived from p. This same ds-list-pair is derived no matter how many times p traverses L.

Note that if the last two duplicate definitions in a loop were recorded in ds-list-pairs, then we would have the ds-list-pair above from a path that traverses L once, and we would have ds-list-pair ($< d_1, dx, dx >, < s_1, sx >$) from paths that that traverse L more than once, and we would report that V is endangered, because of the mismatching ds-list-pair.

Now suppose instead that the left-hand-side of dx and sx is not loop invariant. Turning off some d_i, $i > 1$, does not turn off all d_i, so s_1 qualified reaches Bk with any d_i. In this case, V is endangered, but would be reported as current. Note that if the last two duplicate definitions in a loop were recorded in ds-list-pairs, then we would report V as endangered. This is a semantically pathological case, as the compiler could only move sx to a pre-header if it determined that all data that sx can affect is dead, in which case it should eliminate sx entirely.

There are a number of cases involving two assignments in a loop in which we would report V as current when V is endangered. The proof of Lemma C.12 delimits the cases in which this kind of non-conservative error can occur. Most involve two assignments within a loop, one or (in some cases) both with non-loop-invariant left-hand-sides, affected by optimization in such a way that some variable they both can affect is current if the loop is traversed once, but endangered if the loop is traversed more than once. The compiler must have determined that all variables potentially affected by both assignments are dead. Syntactically, they can be described by the sequences of definitions and stores that reach a breakpoint along a path that traverses the loop twice. Sequences for four cases are shown in Figure 20.1. Because they are symmetric with respect to definitions and stores, four others can be derived by exchanging the roles of definitions and stores in the five shown. In the sequences, the loop iteration boundaries are marked by vertical bars ('|'). The first of these cases is the one described above.

This source of non-conservative error could perhaps be eliminated by recording in the ds-list-pairs the last two duplicate definitions or stores that occur within a loop, rather than the last one, but that would introduce conservative error in the commonly occurring case of loop-invariant code motion, as well as in the above case in the circumstance that the data sx can affect is disjoint from the data sy can affect. To eliminate both the non-conservative

1. Definitions: $< d_1, |d_2 = dx, |d_4 = dx| >$
   Stores: $< s_1, s_2 = sx, | >$
2. Definitions: $< d_1, |d_2 = dx, d_3 = dy, |d_4 = dx, d_5 = dy| >$
   Stores: $< s_1, s_2 = sx, |s_3 = sy|s_4 = sy > |$
3. Definitions: $< d_1, |d_2 = dx, d_3 = dy, |d_4 = dx| >$
   Stores: $< s_1, s_2 = sx, |s_3 = sy| >$
4. Definitions: $< d_1, |d_2 = dy, d_3 = dx, |d_4 = dx| >$
   Stores: $< s_1, s_2 = sx, |s_3 = sy, s_4 = sx| >$

Figure 20.1: Cases in which the Algorithm May Err: Along some paths that traverse a loop more than once, a variable will be reported as current when it is endangered. This can not happen unless sequences of definitions and stores similar to those shown above occur on such paths, and may not happen even if they do. The vertical bars ('|') mark loop iteration boundaries. Other cases can be derived from those shown by exchanging the roles of definitions and stores.

error and the error in the case of loop-invariant code motion, an algorithm must distinguish transparent assignments whose left-hand-sides are loop invariant from those whose left-hand-sides are not. Perhaps future research will uncover a method of distinguishing these and using the information felicitously. Until that time, I prefer the non-conservative error because the algorithm is more elegant, errors will occur less frequently, and the errors that do occur occur in possibly pathological cases involving pointers.

An example follows involving two assignment within a loop, presented in detail. This example corresponds to the second case shown in Figure 20.1. Assume that the path p^L through L contains a transparent definition dx of V but no store sx. Assume further that following dx on p^L there is a transparent definition dy of V, $dy \neq dx$, and that somewhere on p^L is a transparent store sy into V, and that the left-hand-sides of dy (and sy) are not loop invariant. (dy might be an assignment into `a[i]` where `i` is not loop invariant, or an assignment through a pointer `p` where `p` is not loop invariant.) We make no assumption about whether the left-hand-side of dx (and sx) is loop invariant. Finally, assume that there is an instance of sx immediately prior to L. This could occur through code motion (out of L) or through serendipity (perhaps the programmer wrote the same assignment above and inside the loop). Again, for simplicity of presentation assume that no other transparent assignments to V reach Bk. Let p be a path to Bk that traverses L once. Ds-list-pair $(< d_1, dx, dy >, < s_1, sx, sy >)$ derives from p. (Again, d_1 and s_1 are the opaque definition and store reaching from above, and we assume s_1 is generated from d_1). V appears to be current along p, and in fact, V is current along p. Let p' be a path to Bk that traverses L twice. The sequence of definitions of V encountered along p' are $< d_1, d_2 = dx, d_3 = dy, d_4 = dx, d_5 = dy >$, and the sequence of stores into V encountered along p' are $< s_1, s_2 = sx, s_3 = sy, s_4 = sy >$. Because the left-hand-side of the assignments to V are not loop invariant, turning off d_5 turns off s_4 but does not turn off d_3 or s_3. s_3 qualified reaches with d_4, and V is endangered along p', according to Definitions 16.7, 16.8, and 16.9. However, V appears to be current because the ds-list-pair derived from p' (and from all paths that traverse L multiple times) by Algorithm PRS-BK^T is $(< d_1, dx, dy >, < s_1, sx, sy >)$, since only the last instance of duplicated definitions and stores is recorded in the ds-list-pair.

V is in fact noncurrent along p' when d_4 and s_3 are aliases for V but d_5 and s_4 are not.

This is possible if the compiler has determined that either the data sx can affect is disjoint from the data sy can affect, or all data that both sx and sy can affect is dead. If there is no data that both sx and sy can affect, then no variables are endangered, which is what will be reported. If the compiler has determined that everything that sy can affect is dead, then the compiler should not generate sy at all. Likewise, if the compiler has determined that everything that sx can affect is dead, then the compiler should not generate sx at all. We can therefore expect that an error will occur only when the compiler has determined that sx can affect some non-empty set V_x of variables, sy can affect some non-empty set V_y of variables, some element in V_x is live, some element in V_y is live, $V_x \cap V_y$ is non-empty, and every element in $V_x \cap V_y$ is dead. This introduces the potential for misleading behavior (the currency determination technique may claim a variable in $V_x \cap V_y$ is current when it is actually endangered).

21 Non-conforming Optimizations

There are useful optimizations that cannot be modelled by a valid DS-graph. Perhaps future research will discover graph transformations to model some or all of these. In the absence of such transformations, some accommodation must be made for such optimizations, or the debugger may not provide truthful behavior in their presence. I describe below a mechanism to accommodate such optimizations.

21.1 A Mechanism for Truthful Behavior

Let R be a region in the DS-graph that correlates to a region of the object graph that is to be transformed in a manner other than one of the allowable graph transformations. It must be true of R that if node N has a predecessor in R and N has a successor in R along a forward edge, then N is in R.[27] Once R is delimited, no DS-graph transformations are performed upon it, regardless of the object graph transformations that might be performed on the part of the object graph represented (however poorly) by the region.

Each node N in the DS-graph is augmented by a set $\mathit{Affected}_N$ of variables that must be assumed to be endangered because of non-conforming optimizations. For a node M outside of R, $\mathit{Affected}_M$ is empty. Any effects of the non-conforming optimization that reach beyond R are handled by the existing data-flow mechanism using the distinguished definitions and store described below.

A variable V may be endangered if it is defined in a region of the object graph to which some non-conforming transformation has been applied. , and thus V Let d_e^V be a distinguished opaque definition whose presence in a paired reaching set will signal to the debugger that this has occurred. d_e^V is added to the bottom of the definition list of each node in R that has a successor outside of R.

The compiler may be able to ensure that although V is defined in R, V is not endangered outside R by optimization within R. Let d_{ok}^V be a distinguished opaque definition and s_{ok}^V be a distinguished opaque store whose presence will signal to the debugger that this has occurred. In determining currency, the debugger acts as though s_{ok}^V was generated from d_{ok}^V. If V is known to be current on exit from R, d_{ok}^V replaces d_e^V, and s_{ok}^V is added to the bottom of the store list of the same set of nodes. The procedure is as follows:

```
for each variable $V$ that is defined within $R$
    for each node $N$ in $R$
        add $V$ to $\mathit{Affected}_N$
        if $N$ has a successor outside $R$
            if $V$ is known to be current on exit from $R$
                add $d_{ok}^V$ to the bottom of $N$'s definition list
                add $s_{ok}^V$ to the bottom of $N$'s store list
            else
                add $d_e^V$ to the bottom of $N$'s definition list
```

[27]This is a sufficient condition for being in R, but not a necessary condition. Unless R comprises the entire DS-graph, some nodes in R will not meet this condition. R is specified by the compiler.

This mechanism will allow the debugger to find accurate currency information for variables not defined within R, and to provide truthful behavior regarding currency for variables defined within R. When a breakpoint is reached and the debugger takes control, the debugger finds the node in the DS-graph that selects the block the breakpoint is in. When the user makes a query about a variable V, if V is in the *Affected* set for that node, a warning is issued. Otherwise, the currency determination algorithms are run to determine V's currency.

A second order concern may require a simple modification to the DS-graph. If the DS-graph contains a node N within R that has a successor S within R and a successor outside R, then d^V_{ok} and s^V_{ok} are propagated into S (and possibly further within R). This does not affect currency determination, as the *Affected* set is checked first for nodes in R. However, it may affect the paired reaching sets of nodes within R. While these sets are admittedly suspect due to the nature of R, the debugger may use them to produce warnings. To prevent them from being tainted by d^V_{ok} and s^V_{ok}, the DS-graph can be transformed so that there is no node within R that has a successor within R and a successor outside R. A new node N' can be added in R between N and S.

There is a separate behavior issue for the debugger having to do with breakpoint locations that requires the debugger to know whether a breakpoint is within R.

21.2 Interaction with the Breakpoint Model

The method of finding syntactic breakpoints described in Section 22 is not correct when blocks have been re-ordered. A syntactic breakpoint for statement n is defined to be prior to or at the same location as the syntactic breakpoint for statement $n + 1$. In particular, the syntactic breakpoint for statement S in block B precedes the syntactic breakpoint for statement S' in block B' if B must precede B'. But the method for finding syntactic breakpoints uses no information about the relative order of B and B', and is incorrect if their relative order has changed.

A non-conforming optimization may change the relative order of blocks. To provide truthful behavior, a debugger must either use a stronger method of determining syntactic breakpoint locations, or be able to tell that a breakpoint is located within a region of the object graph where the order of blocks may have changed, so that it can warn the user when such a breakpoint is reached. A debugger can do the latter by testing $\mathit{Affected}_N$ for emptiness, where N selects the block the breakpoint is in. A user-interface issue: the debugger should give a warning when the user sets a problematic breakpoint (not just when it is reached). At that point, it would be possible for the debugger to find the nearest ancestors and descendants of N outside of R and provide a list of breakpoints where the user can be given accurate information.

Wismüller [BW93] is working on a more general breakpoint model. His work may address (or finesse) this problem.

22 Design Issues

The material in this section is an architectural design to guide in the implementation of the presented method of currency determination. This method of currency determination requires considerable support from the compiler. If the debugger is to run the algorithms presented in Sections 17 and 19, the compiler must produce the DS-graph for the debugger to run them on.

The DS-graph is not a static object. As optimization transforms the object graph, the DS-graph gets transformed. In Sections 17 and 19, we assumed that the definition and store lists were correct at each point. These also, of course, will be modified as optimization proceeds. Both the graph transformations and the definition and store list modifications have implications for the compiler's data structures. The data structures that are appropriate during compilation are different from those that are appropriate once compilation has completed. Section 22.1 describes an abstraction of a set of data structures for the DS-graph and part of the object graph that can easily be mapped to an imperative programming language.

Once compilation is complete, the DS-graph must be emitted for use by the debugger. It differs somewhat in organization from the DS-graph used during compilation, and the differences are described in Section 22.2.

22.1 Data Structures Used During Compilation

There are two differences between the structures described here and the abstractions described in Sections 17 and 19:

1. The source graph is not needed by the compiler or debugger. It is included in the discussions in Sections 17 and 19 to motivate the approach and referred to in Appendix B to demonstrate its correctness.
2. The store lists reside in the object graph, as part of the instruction lists used to generate code, rather than in the DS-graph.

A DS-graph node contains all necessary information about the source block that it selects, so the source block itself can be dispensed with. Figure 22.1 shows the abstract interconnections between the major structures used for currency determination during compilation: nodes in the DS-graph, which contain definition lists, blocks in the object graph, which contain instruction lists, and mapping information between them.

Definition and Instruction Lists and the 'Generated-from' Graph

Section 17 mentioned that currency determination needs the following:

- The assignments that constitute a definition,
- the 'generated from' relationship between definitions and stores, and
- the execution order of statements and side effects within a basic block, for blocks in both the optimized and unoptimized versions.

We assumed they would be available, and that execution order would be provided by definition lists and store lists.

Viewed abstractly, the 'generated from' relationship between definitions and stores forms a bipartite graph, which we shall henceforth call the G-graph, where definitions comprise the nodes in one partition, stores comprise the nodes in the other, and an edge is present between pairs of nodes for which the 'generated from' (or 'generates') relation holds.

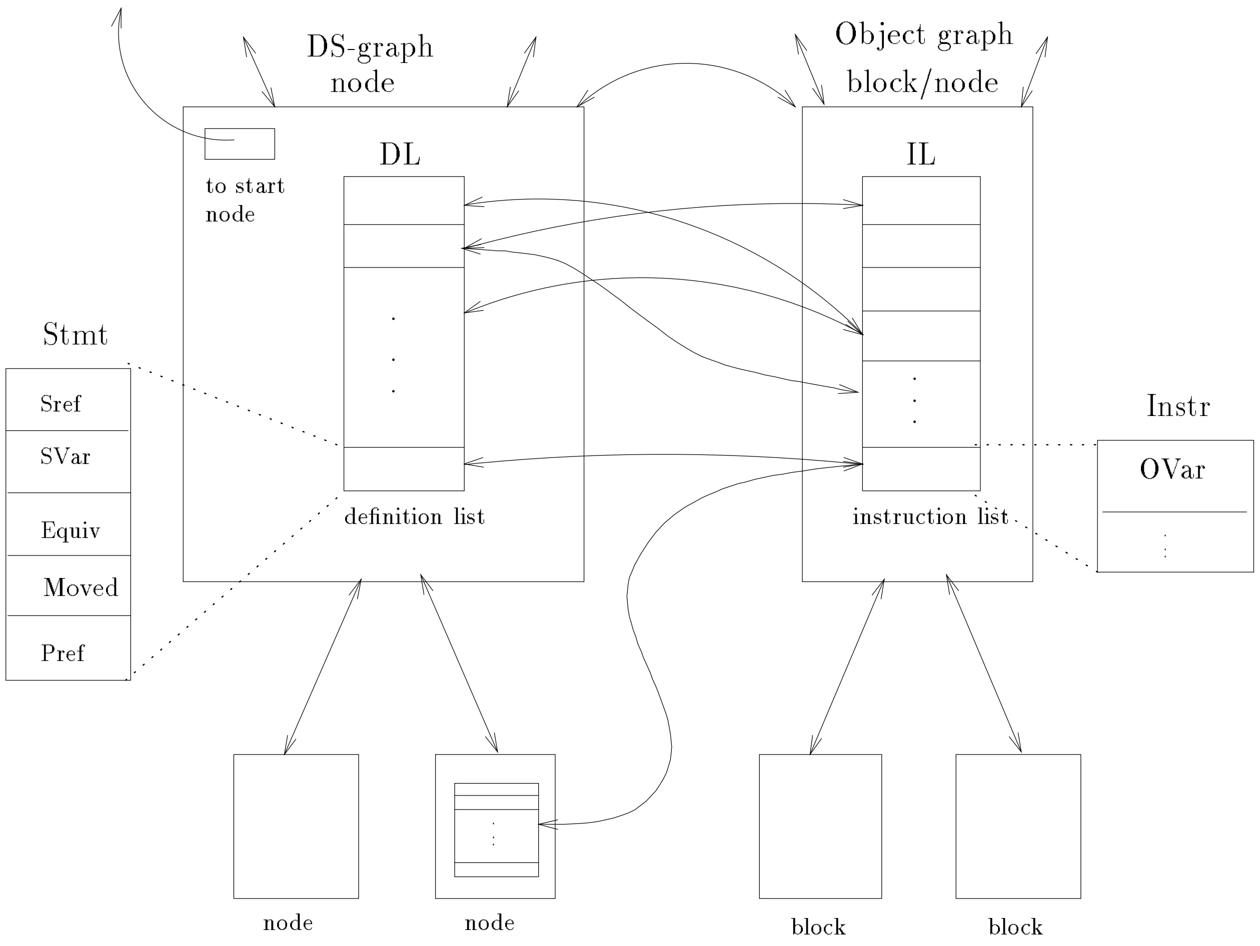


Figure 22.1: This shows the abstract interconnections between the major data structures used for currency determination during compilation: nodes in the DS-graph, which contain definition lists, blocks in the object graph, which contain instruction lists, and mapping information between them.

Definition nodes in the G-graph are represented as *Stmt* structures, each representing a statement.[28] Each DS-graph node contains a definition list DL which lists, in source order, the *Stmt*s that occur in the source code for the DS-graph node. Store nodes in the G-graph are represented as $Instr$ structures, each representing an intermediate language instruction (and eventually, a machine instruction). I describe only the part of the $Instr$ structure that is relevant to currency determination, but it is assumed to include the intermediate language expression describing the instruction, and any other information the compiler needs about the instruction. That is, the *Instr* structure is the compiler data structure that encodes intermediate language instructions, which is extended for currency determination. Each object graph block contains an instruction list IL, which lists, in execution order, the *Instr*s that occur in the intermediate language representation of the program. Every intermediate instruction in the object code has an entry in some IL, but only representative instructions

[28] A *Stmt* may represent part of a statement; see the discussion below.

are of interest for currency determination. Specifically, only representative instructions are represented as store nodes in the G-graph. Each *Instr* structure must contain information as to whether it is a representative instruction. Embedded in the *IL* for a block is the store list of the block, which consists of those *Instr*s that are representative instructions.

Note that every statement has a *Stmt* structure, whether it is an assignment or not, and every statement for which code is generated has an *Instr* structure that comprises a store node.

DL and *IL* look like arrays in Figure 22.1 because it is a convenient way to represent them visually. This should not bias an implementation—they are lists of nodes, and any convenient data structure for representing lists can be used. Elements in *IL* get moved by optimization, so an array representation of that list may not be efficient. (Because *IL* is a compiler data structure used whether currency determination is done or not, its representation is likely to be fixed before this currency determination technique is implemented in a compiler.)

The 'generated from' relationship between definitions and stores is represented in Figure 22.1 by pointers between *Stmt*s and *Instrs*. Again, this should not bias an implementation. The G-graph may be represented by any convenient data structure for representing graphs.

A definition is an equivalence class of *Stmt*s. Often a *Stmt* comprises a definition by itself, but in the case when a definition is comprised of more than one *Stmt*, some mechanism must associate them. I give *Stmt*s a field *Equiv* to represent the definition—assignments in the same equivalence class have the same value in their *Equiv* field.

Stmt and *Instr* Structures

A *Stmt* structure for a statement S has the following fields:

- *Sref*—a source reference for S (file name or id and line number, and perhaps which statement on the line, if the debugger is to handle lines with multiple statements),
- *SVar*—the variables that can be defined by S: if S is opaque, *SVar* identifies the single variable defined by S; if S is transparent, *SVar* identifies all variables that can be affected by S, if S does not assign into a variable, the *SVar* field is null,
- *Equiv*—the equivalence class that a definition falls into,
- *Moved*—a bit set if the representative instruction has been moved or eliminated, used to locate syntactic bkpts, and
- *Pref*—for definitions through pointers: a reference to the pointer.

An *Instr* structure that is a representative instruction for a statement S has the following fields:

- Any fields needed by the compiler to generate code, and
- *OVar*—the variables that can be defined by S: if S is opaque, *OVar* identifies the single variable defined by S; if S is transparent, *OVar* identifies all variables that can be affected by S, if S does not assign into a variable, the *OVar* field is null.

A compiler could recognize that part of a variable's storage is not referenced. Suppose statement S is the aggregate assignment $x = y$. If the compiler can determine that only one field of x is subsequently used, it can optimize the code to assign only into that field of x.[29] If a debugger user were to inspect some other field of x subsequent to the assignment,

[29] I know of no compiler that does this, but there is no reason that one could not.

she might be misled. The *OVar* field identifies the part of that variable assigned into by S, if such optimization has occurred. This enables the currency determination algorithm to identify other fields of x as noncurrent or endangered.

In the context of computing paired reaching sets for a variable V, the *SVar* and *OVar* fields are used to find definitions of V and stores into V to construct the Gen set for each block.

A distinct *Stmt* (and *Instr*) is produced for each modification to each program variable, so more than one *Stmt* is produced for a statement that has side effects.[30] For example, the following code causes three *Stmts* to be produced:

`a = 0;`	(Produces one *Stmt*.)
`b = c++;`	(Produces two *Stmts*: one for the assignment into b, and one for the side effect on c.)

G-graph Modification and Maintenance

During optimization, code may be deleted, moved, copied, or shared. Associating store nodes with the compiler's representation of instructions by augmenting the *Instr* structure means existing optimizer technology takes care of updating the *IL*s. However, these operations may modify the G-graph, or may necessitate updating of the data structure chosen to represent the G-graph. Let the G-graph $G = (V^D \cup V^S, E)$ where V^D are definitions (*Stmts*) and V^S are stores (*Instrs* that are representative instructions) and let I be in V^S. G is an undirected graph, thus (x, y) and (y, x) represent the same edge. For consistency, I always list the store node in an edge prior to the definition node.

In earlier sections, I assume that the generated-from relationships are known, and that the definition and store lists are correct, implying that they are correctly updated when optimization is done. The generated-from relationships is encoded in G, and the following describe how to generate a new G-graph $\overline{G}$ from G when an optimization affects those relationships. I also describe how to update fields in the definition and store lists when optimization affects them.

1. Delete I:
   $\overline{V^S} = V^S - \{I\}$
   $\overline{E} = E - \{(I, x) | x \in V^D\}$
   $\overline{G} = (V^D \cup \overline{V^S}, \overline{E})$
   $x.\mathit{Moved} = 1$ for all x such that $(I, x) \in E$.
2. Move I:
   The structure of G does not change, but the underlying data structure may change.
   $x.\mathit{Moved} = 1$ for all x such that $(I, x) \in E$.
3. Copy I:
   $\overline{V^S} = V^S \cup \{\overline{I} | \overline{I}$ is a copy of $I\}$
   $\overline{E} = E \cup \{(\overline{I}, x) | (I, x) \in E\}$
   $\overline{G} = (V^D \cup \overline{V^S}, \overline{E})$

[30] More than one *Stmt* is produced for a statement that has more than one location at which user-visible changes occur. This is true of statements with side effects. It is also true of many loop constructs. A C `for` loop may have three places of interest to the user corresponding to its three expressions, and each needs a *Stmt* if the debugger is to be able to break at each one.

4. Sharing:

   Let J be in V^S. Assume that the result of the code sharing is that I becomes the representative instruction for J's *Stmt*s as well as its own.

   $\overline{V^S} = V^S - \{J\}$

   $\overline{E} = E \cup \{(I,x)|(J,x) \in E\} - \{(J,x)|x \in V^D\}$

   $\overline{G} = (V^D \cup \overline{V^S}, \overline{E})$

   $x.Moved = 1$ for all x such that $(J,x) \in E$.

   $I.Ovar = I.Ovar \cup J.Ovar$

   For all x such that $(J,x) \in E$ and some y such that $(I,y) \in E$, $x.Equiv = y.Equiv$.

The graph transformations impose some requirements on the data structures. Many of the transformations require finding the node n such that $Oselects(n) = b$ for some block b involved in the object graph transformation. *Nselects*, the inverse mapping of *Oselects*, is needed to make this efficient. Because of block coalescing, the range of *Oselects* and the domain of *Nselects* must include block boundary markers as well as object graph blocks.

Some DS-graph and object graph transformations involve updating the definition or instruction lists. Transformations 5 and 6 share a characteristic that blocks in the object graph and nodes in the DS-graph get copied. As described informally in Section 17.3, the definitions in the definition list in a new node are in a different equivalence class from the definitions in the definition list in the node it is a copy of. Store equivalence classes are derived from definition equivalence classes. This means that when DS-graph node N is copied to $\overline{N}$, the *Stmt*s in $\overline{N}$'s DL are new nodes in the G-graph. Similarly, when object graph node B is copied to $\overline{B}$, the *Instr*s in $\overline{B}$'s IL are new nodes in the G-graph. I describe below how to generate the new G-graph nodes and edges when an inlining or unrolling transformation is performed on the DS-graph.

In the following, I use the convention that $\overline{x}_i$ denotes the i^{th} copy of *Stmt* or *Instr* x introduced in the transformation. The subscript is needed for loop unrolling, where there may be many copies of x introduced in a single transformation. The fields in $\overline{x}_i$ have the same value as the fields in x unless explicitly stated otherwise.

5. Unrolling or Inlining:

   Let *CopiedDLs* be the set of DLs in nodes that were copied in in the DS-graph transformation.

   Let *DLCopies* be the set of copies of the DLs in *CopiedDLs*.

   Let *CopiedILs* be the set of ILs in blocks that were copied in the DS-graph transformation.

   Let *ILCopies* be the set of copies of the ILs in *CopiedILs*.

   $NewStmts = \{\overline{S}|S$ is a *Stmt* in a DL in $CopiedDLs\}$

   $NewInstrs = \{\overline{I}|I$ is an *Instr* in an IL in $CopiedILs\}$

   $BothEdgesInside = \{(\overline{S},\overline{I})|\overline{S} \in NewStmts$ and $\overline{I} \in NewInstrs$ and $(S,I) \in E\}$

   $StmtEdgeInside = \{(\overline{S},I)|\overline{S} \in NewStmts$ and $(S,I) \in E$ and $\overline{I} \notin NewInstrs\}$

   $InstrEdgeInside = \{(S,\overline{I}|\overline{I}) \in NewInstrs$ and $(S,I) \in E$ and $\overline{S} \notin NewStmts\}$

   $\overline{V^D} = V^D \cup NewStmts$

   $\overline{V^S} = V^S \cup NewInstrs$

   $\overline{E} = E \cup BothEdgesInside \cup StmtEdgeInside \cup InstrEdgeInside$

   $\overline{G} = (\overline{V^D} \cup \overline{V^S}, \overline{E})$

To set the *Equiv* field of the new *Stmt*s, *NewStmts* must be partitioned according to the equivalence classes in the *Stmt*s they are copies of. Each partition is given a unique *Equiv* value:

For $\overline{S} \in NewStmts$ and $\overline{S}' \in NewStmts$, $\overline{S}.Equiv = \overline{S}'.Equiv$ if and only if $S.Equiv = S'.Equiv$. $\overline{S}.Equiv \neq X.Equiv$ for $X \notin NewStmts$.

22.2 Data Structures Used After Compilation

Once compilation has completed, the DS-graph becomes a static object. The data structures for this object should be chosen to make the object small and the debugger's uses of the object efficient, rather than making maintenance of the object efficient. Information about representative instructions must be moved from the object graph into the DS-graph, because the object graph does not survive code generation as an explicit data structure.

Figure 22.2 shows the abstract interconnections between the major data structures used for currency determination after compilation. As is clear from comparing Figure 22.1 and Figure 22.2, IL is moved from the object graph to the DS-graph. Only representative instructions are placed in the version of IL that is moved to the DS-graph. Also, the *Instr* structure changes: information used to generate code is discarded, but the address of the generated code for a representative instruction is added in the *Cref* (*C*ode *ref*erence) field, so that the DS-graph can be used for locating breakpoints.[31]

Let N be a node in the version of the DS-graph used during compilation. Let B be the block in the object graph selected by N. Let N' be the node corresponding to N in the version of the DS-graph used after compilation.

$N'.DL = N.DL$

$N'.IL =< I_1, I_2, \ldots, I_n >$, where each I_i is an *Instr* in $B.IL$ that is a representative instruction, and for I_i and I_j in $N'.IL$, $i < j$ if and only if I_i precedes I_j in $B.IL$.

Given that the DS-graph is available to the debugger, we need to ensure that the debugger has all the information it needs to determine the currency of variables. The following debugger tasks relate to currency determination:

1. When a user sets a breakpoint at a source statement, find the breakpoint location(s). A method for finding breakpoint locations is given below.This requires one additional map from source statements to *Stmt*s. When the breakpoint is set, the source statement and breakpoint locations must be saved in a debugger data structure *BrList*.
2. When the program stops, find out which breakpoint it has halted at. This can be done by comparing the program counter against breakpoint locations found in *BrList*,
3. When the user asks for the value of a variable, run the currency determination algorithms and respond to the query. The currency algorithms take as input the variable name, a syntactic breakpoint Bk, and the node B containing Bk. The variable name is supplied by the user, and the debugger has determined Bk. The DS-graph data structure must allow B to be found given Bk.

[31]At the point that the compiler emits this address, it has only relocatable addresses. At the point the debugger uses it, it needs the relocated addresses. One way of dealing with this problem is to have the compiler emit a label at each representative instruction, and have the *Cref* field contain the label name. The linker will do the necessary relocation and the debugger can refer to the addresses symbolically.

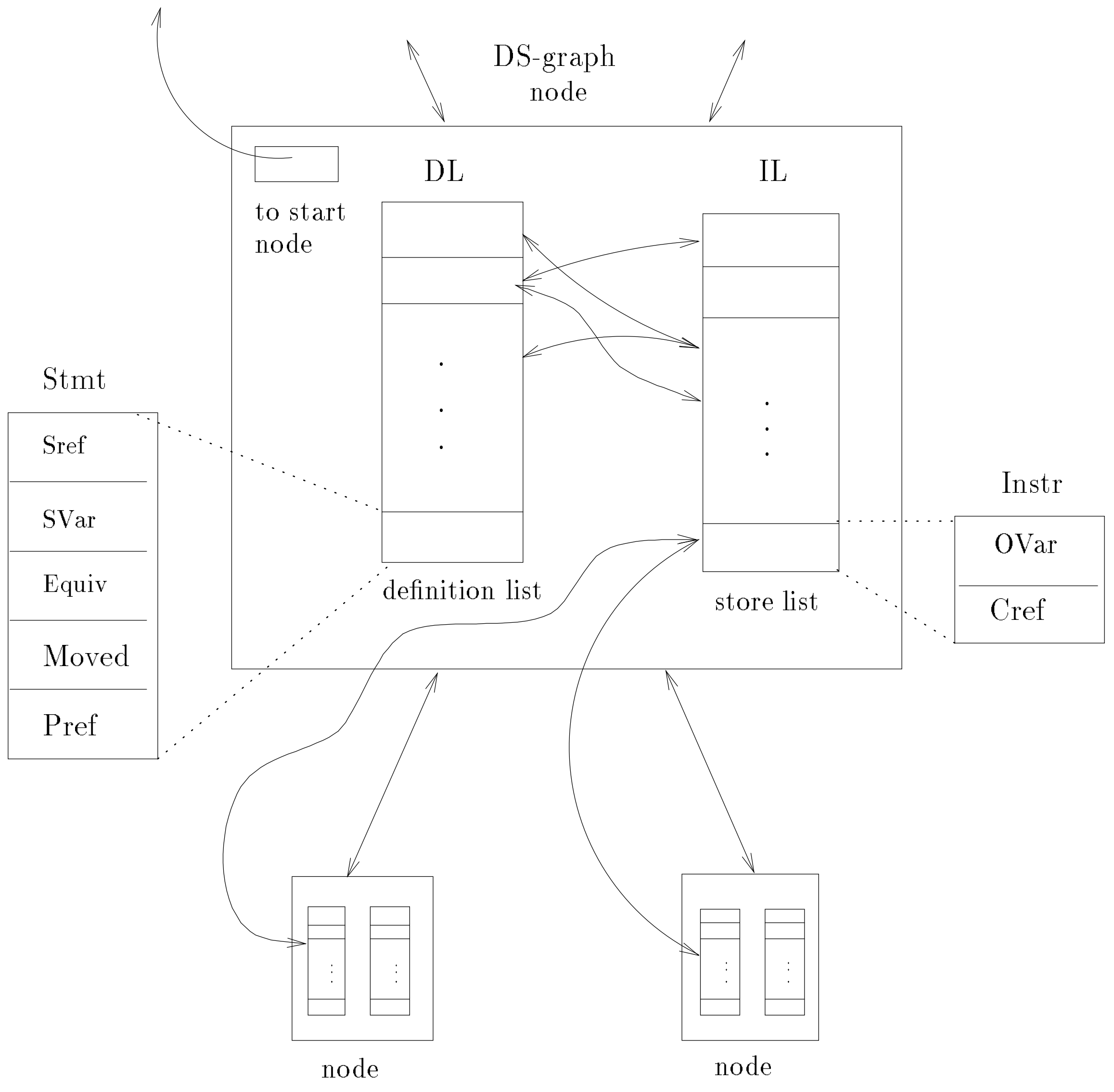


Figure 22.2: This shows the abstract interconnections between the major data structures needed by the debugger for currency determination: nodes in the DS-graph which contain definition lists and store lists and mapping information between them.

Finding Breakpoint Locations

Given a reference to a source statement, a debugger must be able to find the machine code addresses at which to set breakpoints. Existing mechanisms do not support syntactic breakpoints, so I describe one way syntactic breakpoint locations can be found.

The *Cref* field of an *Instr I* encodes a semantic breakpoint location for all statements for which *I* is the representative instruction.[32] Given the other information in the DS-graph,

[32]Currency determination using the technique presented herein cannot be performed at the semantic

Cref can be used to find the syntactic breakpoint location for a *Stmt* for which I is the representative instruction. We need some mapping M from source statements to *Stmt*s. This may be a one-to-many mapping (due to inlining or unrolling).

Syntactic Breakpoint Locations The syntactic breakpoint locations for a source statement are found by using M to find all *Stmt*s that represent it, then finding the syntactic breakpoint location L for each such *Stmt* S as follows (described in English, and again in terms of the defined structures):

```
If a representative instruction for $S$ has not been moved
    $L$ is the address of that instruction.
If there is no unmoved representative instruction for $S$
    if the block that originally contained $S$ does not appear at all in the optimized code,
        $L$ is undefined,
    else if any representative instructions for statements following $S$
    within the block containing $S$ have not been moved,
        $L$ is the location of the first of these,
    else $L$ is the location of the last representative instruction within the block containing $S$.
```

Let *Node(S)* be the node containing *Stmt* S. A syntactic breakpoint location is designated by the *Cref* field of an *Instr* that has never been moved, which can be determined by checking the *Moved* field of the *Stmt* that generates the *Instr*. A *Stmt* may generate multiple *Instr*s, through copying or sharing. The original *Instr* generated by the *Stmt*, if it has not been moved, holds the syntactic breakpoint location for the *Stmt*. Any others generated by the *Stmt* via copying or sharing do not represent syntactic breakpoint locations for the *Stmt*. The copying transformation on the G-graph does not set the *Moved* field of the *Stmt* that generates the new *Instr* because if it did, no syntactic breakpoint could be set for the *Stmt*. However, the original *Instr* must be distinguished from any copies, as it represents the only syntactic breakpoint location for the *Stmt*. In what follows, we assume the edges E in the G-graph are ordered so that the edge between a *Stmt* and the original *Instr* generated from it is the first edge out of the *Stmt*.

```
If $S.Moved$ is not set
    $L = I.Cref$ where $(I, S)$ is the first edge out of $S$ in $E$
else
    if $Oselects(Node(S)) == Null$
        $L$ is undefined,
    else
        let $S'$ be the first Stmt subsequent to $S$ in Node(S) such that $S'.Moved$ is not set,
            or if there are none, let $S'$ be the last Stmt in Node(S)
        $L = I.Cref$ where $(I, S)$ is the first edge out of $S'$ in $E$
```

breakpoint location for a statement S whose code has been moved. However, it is possible to break at a semantic breakpoint location for S, find a statement S' for which it is a syntactic breakpoint location, and determine a variable's currency relative to S' (informing the user that the variable is current or endangered at S'—this does not tell us whether it is current at S).

23 Side benefits of DS-graphs

Control flow information can be used by a debugger for purposes other than currency determination. For example, statement stepping (often called source-line stepping) is one of the more difficult debugger capabilities to implement because it is difficult to determine where the next breakpoint(s) should be set. With control flow information, this problem becomes simple. Using the program flow graphs and *Stmt* and *Instr* structures described in this section, if S is the *Stmt* for the current statement, the current breakpoint is at $I.Cref$ where (I, S) is the first edge out of S in the G-graph. For conciseness, I will call $I.Cref$ 'the *Cref* for S'.

If S is not the last *Stmt* in its node, the next breakpoint can be set at the *Cref* for the next *Stmt* in the node. If S is the last *Stmt* in the node, breakpoints can be set at the *Cref*s for the first *Stmt* of each successor node.

Data flow information can be used by a debugger for purposes other than currency determination as well. A debugger user who has just displayed the value of a variable V at a breakpoint B may well want to know where V was last set. This information is available in PRS_B^V.

24 Cost

There are four distinct costs that might be considered relevant to evaluating this method of currency determination:

1. the cost of either not debugging optimized code at the source level or being occasionally misled by the debugger,
2. the engineering cost of modifying a compiler and debugger,
3. the additional space and time the compiler takes to produce the DS-graph, and
4. the additional space and time the debugger takes to run the currency determination algorithms.

The first should be weighed against the rest to determine whether the enterprise is worthwhile. Unfortunately, they are not comparable, hard to measure, and are incurred by different groups. For the most part, academia and industry did not consider the enterprise worthwhile until the late 1980's or early 1990's. Currently there is activity in both academia and industry. If the balance of these costs has shifted, it can be seen as a continuation of the trend of the cost of programmers rising relative to the cost of computers, coupled with improved optimization technology and widespread availability of that technology.

The cost of not debugging optimized code at the source level includes programmer frustration, time spent recompiling to enable and disable optimization, disk space to store both optimized and unoptimized object modules and executables, time spent communicating with compiler vendors incorrectly claiming the compiler has a bug, time spent debugging the wrong thing because the debugger gave misleading information, and time spent finding bugs via assembly-level debugging (minus the time it would have taken via source-level debugging, were that an option). Quantitative measures of these are not available.

The engineering cost of modifying a compiler and debugger is also not available. In general, it depends on the organization of the compiler and debugger that are to be modified, and the capabilities of the engineers that do the modification. No implementation has been done to date, and any estimate I made would be subject to the same poor correlation between estimates and actual costs of software production that is endemic to the industry.

The additional space and time a compiler needs to produce the DS-graph can be estimated with greater confidence.

24.1 Cost to Produce the DS-graph (During Compilation)

The DS-graph is linear in the size of the object graph. The space used by a compiler less-than-doubles, because the compiler has data structures other than the object graph.

The time to construct the DS-graph is linear in the size of the object graph as well. The costs of Transformations 1 and 2 are linear in the size of the DS-graph for pathological programs, but will typically be constant time operations. The total cost of applications of Transformation 3 is linear in the size of the DS-graph, since each edge can be removed at most once. Transformation 4 is a constant time operation. The cost of Transformation 5 is linear in the size of the subroutine being inlined, and the cost of Transformation 6 is linear in the size of the loop being unrolled and the number of times it is unrolled.

Slists and Dlists could be as large as the number of assignments in the program, for pathological programs. Typically, their maximum length will be some small constant, so the time to update them will be linear in the number of optimizations that affect representative instructions. The time spent in optimization will less-than-double because the time spent constructing and maintaining the DS-graph will be similar to the time spent constructing and maintaining the object graph, but much of the time spent in optimization is in analysis, which has no cost counterpart in maintaining the structures needed to perform currency determination.

24.2 Cost to Use the DS-graph (After Compilation)

Algorithm PRS is a special case of Algorithm PRS^{CT} and is not separately analyzed.

Space

The DS-graph need not be made a part of the executable, so it does not affect the debugger's ability to load the program. As with symbol tables, if the debugger has to read the entire DS-graph at once, there may be an unacceptably long delay while this is done. But as with symbol tables, the DS-graph can be read in on demand, because only one component (subroutine) is needed at a time.

Currency determination increases space usage by the size of the In, Out, and PRS sets. The space usage of the In and Out sets dominates that used by the PRS sets, because there is an In and Out set per block. Let n be the number of nodes in the DS-graph and m be the number of assignments to V in the program.

For Algorithm PRS^{CT}, the In and Out sets can take $O(nm^2)$ space, because there is an In and Out set per node, and the ds-pairs are from definitions $\times$ stores, both of which are $O(m)$ in size. These sets can be constructed as needed, so the total space for In and Out sets is $O(qnm^2)$, where q is the number of variables about which the user queries.

For Algorithm PRS^T, the In and Out sets can take $O(nm(m!^2))$ space. There is an In and Out set per node ($O(n)$), the maximum size of a ds-list-pair is $O(m)$ (m definitions plus m stores), and the number of elements in In_B^V is $O(m!^2)$, giving a worst case bound on the space for In and Out sets of $O(qnm(m!^2))$. The bound on the number of elements in In_B^V is derived by taking all permutations of definitions $\times$ all permutations of stores. In the expected case, the number of definitions and stores that reach B is limited, and the order in which they reach along different paths will include few of the possible permutations. The squaring of $m!$ is from allowing definitions and stores to be ordered independently. But they are not independent—a store can only move via optimization. If many definitions or stores reach along some path, then there are many assignments through pointers, and the optimizer probably can't move many of them. I believe in practice the number of elements in any In set will be small, and would be surprised if it exceeds m.

Time

Algorithms PRS-BP^{CT} and PRS-BP^T, and the initialization phases of Algorithms PRS^{CT} and PRS^T, are inexpensive relative to the iteration phase of Algorithms PRS^{CT} and PRS^T, and will not be discussed further.

Algorithm PRS^{CT}: The worst-case asymptotic cost of Algorithm PRS^{CT} is poor, though polynomial.

The algorithm is presented as being run for all variables. However, it is reasonable for the debugger to run it on a single variable. The cost described here is for a single variable.

The worst case asymptotic cost is $O(n^3m^2)$. However, we will see that in practice two factors of n and a factor of m can be replaced with constant factors, for an $O(nm)$ running time.

Computing In and Out sets is done with an iterative algorithm that runs until it converges. The equations are

$$\mathrm{In}_B^V = \bigcup_P \mathrm{Out}_P^V \text{ for } P \text{ predecessors of } B$$

$$\mathrm{Out}_B^V = \mathrm{In}_B^V \overset{\kappa}{\cup} \mathrm{Gen}_B^V$$

and these are computed iteratively over all nodes B until no In or Out set changes.

Within each iteration the computation of Out_B is cheaper than the computation of In_B. Computing In_B involves iterating over the (up to n) predecessors of B, and In_B is computed for each of n blocks, so the union operation is performed n^2 times. The union operation is a merging of sets containing at most m elements, which can be done in time proportional to m, so each iteration has worst case cost of n^2m.

In the worst case, each iteration could add one definition to one block, so the total number of iterations could be nm, for an $O(n^3m^2)$ total worst case running time.

If a flow graph is traversed in the right order, on average 5 iterations are sufficient for convergence for a standard reaching definitions algorithm [ASU86], replacing factors of nm with a factor of 5. While this algorithm differs from a standard reaching definitions algorithm, we expect that some small constant number of iterations will be sufficient for convergence, removing a factor of nm from the running time.

Two factors of n come from iterating over n blocks with n predecessors each. A fully connected flow graph is pathological. Most blocks have one or two predecessors, though some have many (e.g., the block following a case or switch statement). Generally, the number of predecessors is some small constant, and in the exceptional cases it is a somewhat larger constant—which gives us an $O(nm)$ running time.

Algorithm PRS^T: The analysis of Algorithm PRS^T is identical except for the union operation, which is merging of sets containing at most $(m!^2)$ elements, for a worst-case asymptotic running time of $O(n^3m(m!^2))$ and an expected running time of an $O(nm)$.

In the union operation, κ and *Last* are applied once for each element in the In set, which contains $O(n)$ elements. κ and *Last* involve $O(m)$ work, so the union operation has an $O(nm)$ cost, contrasted to the $O(m)$ cost for Algorithm PRS^{CT}.

Parameters

The parameters that affect the cost are:

- n, the number of basic blocks in a routine,
- m, the number of assignments to a variable within a routine, including assignments through pointers that might point to that variable,
- the number of paths to a block, and
- the number of predecessors and successors per block.

These depend considerably on program characteristics and coding style. In particular, because each subroutine is a flow-graph component, the cost increases with the size of subroutines. The cost also increases with the use of pointers.

25 Open Problem: When a Breakpoint has Moved

Semantic breakpoints introduce additional complexity into currency determination.

Under the breakpoint model given in Section 15, there is no guarantee that a semantic breakpoint is reached in optimized code if and only if it would be reached in unoptimized code. When a semantic breakpoint is reached, the point in the optimized code at which execution is suspended (and the user examines a variable's actual value) may not correspond to the point at which the user expects execution to be suspended (the point at which the user intended to examine the value). There are four distinct situations that can arise with a semantic breakpoint for a statement S:

1. The code for S has not been moved. The semantic breakpoint is the same as the syntactic breakpoint, and no additional work is required for currency determination.
2. The code for S has been moved. In a particular execution, the semantic breakpoint location and the syntactic breakpoint location are reached along the same path.
3. The code for S has been moved. In a particular execution, the syntactic breakpoint location is reached but the semantic breakpoint location is not. This is a source of unexpected behavior, but no additional work is required for currency determination because the user never gets to ask for the value of a variable at the semantic breakpoint.
4. The code for S has been moved. In a particular execution, the semantic breakpoint location is reached but the syntactic breakpoint location is not. This is unexpected behavior already.

In situations 2 and 4 we want to determine whether the actual value of a variable at a representative instruction R (the semantic breakpoint, where the user examines the value) can differ from its expected value at a representative instruction $R' \neq R$ (the syntactic breakpoint, where the user expects to be examining the value). Note that in general a debugger cannot distinguish situation 2 from situation 4.

An approach taken by Bemmerl and Wismüller [BW92] is to use a more flexible mapping between source statements and breakpoints. They attempt to map a source statement to a breakpoint location in such a way that the breakpoint is reached if and only if it would be reached in unoptimized code.

There is a further problem. Consider Figure 25.1. For `bkpt` to be reached in the optimized code, one of the right-hand paths must be taken. If the unoptimized code is run on the same inputs, one of the right-hand paths will be taken, so optimization does not affect the value that `a` will have at the semantic breakpoint for `bkpt`: `a` is current at `bkpt`. Bemmerl and Wismüller's current approach [Wis93] would claim that `a` is endangered at `bkpt`. Suppose we allowed semantic breakpoint locations using my approach to currency determination. Let S be the semantic breakpoint location for `bkpt`. $\mathrm{PRS}_S^{\mathtt{a}} = \{(\mathtt{a=x}, \mathtt{a=x}), (\mathtt{a=y}, \mathtt{a=y})\}$. In each pair, the store is generated from the definition, suggesting that `a` is current at the final location of `bkpt`.

Now consider Figure 25.2. In this case, `bkpt` may be reached in the optimized code by any path. In particular, the left-hand path may be taken, so optimization affects the value that `a` may have at the semantic breakpoint for `bkpt`: `a` is endangered at `bkpt`. Bemmerl and Wismüller's approach would not allow a breakpoint to be set at `bkpt`. Under my approach, the wrong result is obtained: Let S be the semantic breakpoint location for `bkpt`. $\mathrm{PRS}_S^{\mathtt{a}} = \{(\mathtt{a=z}, \mathtt{a=z}), (\mathtt{a=x}, \mathtt{a=x}), (\mathtt{a=y}, \mathtt{a=y})\}$. In each pair, the store is generated from the definition, suggesting that `a` is current at the final location of `bkpt`.

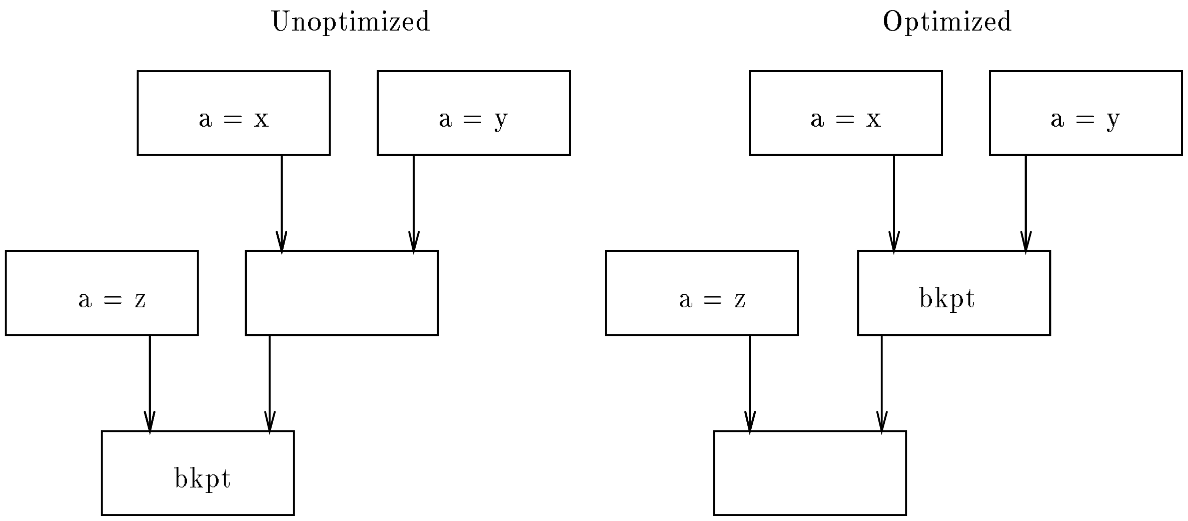


Figure 25.1: Oddly enough, `a` is current at `bkpt`

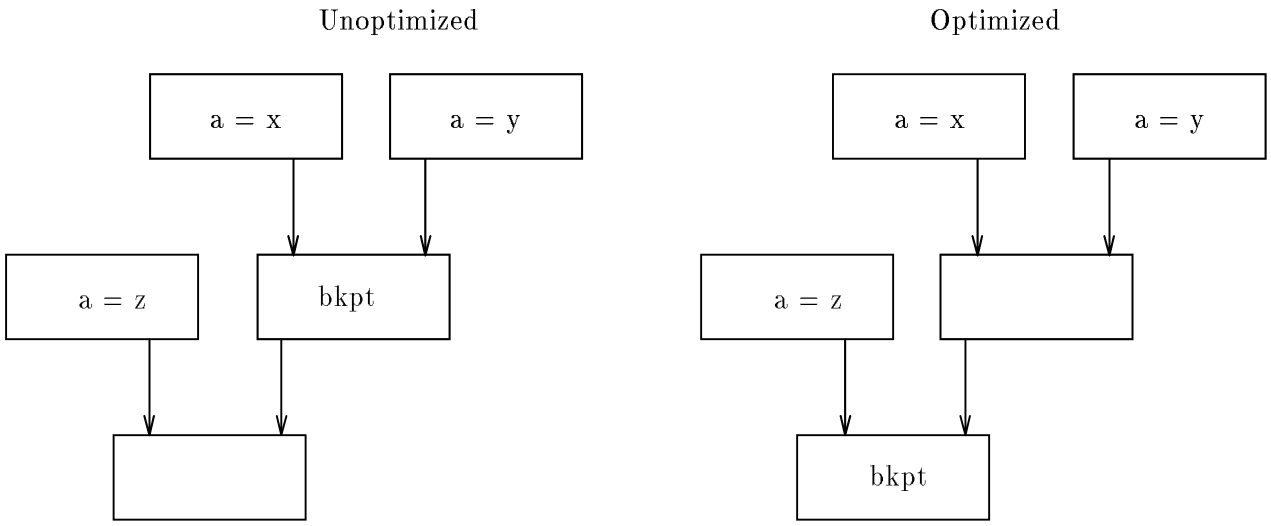


Figure 25.2: To no-one's surprise, `a` is endangered at `bkpt`

26 Currency and Residency

The work on currency determination makes a simplifying assumption that a variable resides in a single location throughout its lifetime.

V is *resident* [AG92] at some point P if and only if there is a storage location (which could be a register) that holds V at P, otherwise V is *nonresident* at P. V must be resident throughout its live ranges (else the compiler is in error). The last use in one of V's def-use chain is the end of one of V's live ranges. The point at which V's location is used to hold something else (which cannot be earlier but may be later) is the end of V's residency in that location, so V is likely to be resident outside its live ranges.

It is increasingly common for a compiler to provide live-range and variable location information to a debugger [CMR88], [Wan91], [Str91], [BHS92]. If a debugger does not supply the value of a variable outside of its live ranges, there will be points at which the variable is in some storage location but the debugger will not display it to the user.

Adl-Tabatabai and Gross [AG92], [AG93a], [AG93b] show how data-flow analysis can be used to determine the residency of variables. Their technique finds a unique location for a variable at a point or considers it nonresident at that point, but a simple modification will find all locations holding (possibly different values of) the variable at a point. Berger and Wismüller's current work ([BW93]) incorporates residency and currency determination.

Figure 26 is an example in which variable `i` is available in two locations, containing two different values, over a range of code.[33] It could be valuable to a debugger user to be able to query the debugger for all available values of a variable and the definitions associated with each value, whether the variable is current or not. This is orthogonal to the question of currency, but arises when the assumption that a variable resides in a single location is relaxed.

Another issue that arises when that assumption is relaxed is where the value of a variable is to be found. In the example in Figure 26.2, `i` is current at `bkpt1`, but a static analysis cannot tell whether `i` is in `R1` or in `R2` which without knowing which path was taken. At `bkpt2`, `i` is endangered, and a static analysis can determine that `R2` does not hold `i`.. I believe that these cases can be handled by extensions to the algorithms given earlier in which a store contains information about the stored-into location, but this falls into the category of future work.

In such cases, the debugger cannot simply display `i`, because of the ambiguity about `i`'s location. But the user may know (or be able to determine) which path was taken, and so may be able to take advantage of information the debugger has.

[33]Thanks to Dror Zernik for this example.

Figure 26.1: `i` is available in `R1` or `R2` at the same time.

```
Line #   Source         Inst#   Unoptimized      Optimized Object
=================================================================
1       i=0            I       R1=0             a(I)    R1=0
2       x=i            II      x=R1             b(III)  R2=1
3       i=1            III     R2=1             c(II)   x=R1
4       y=i            IV      y=R2             d(IV)   y=R2
```

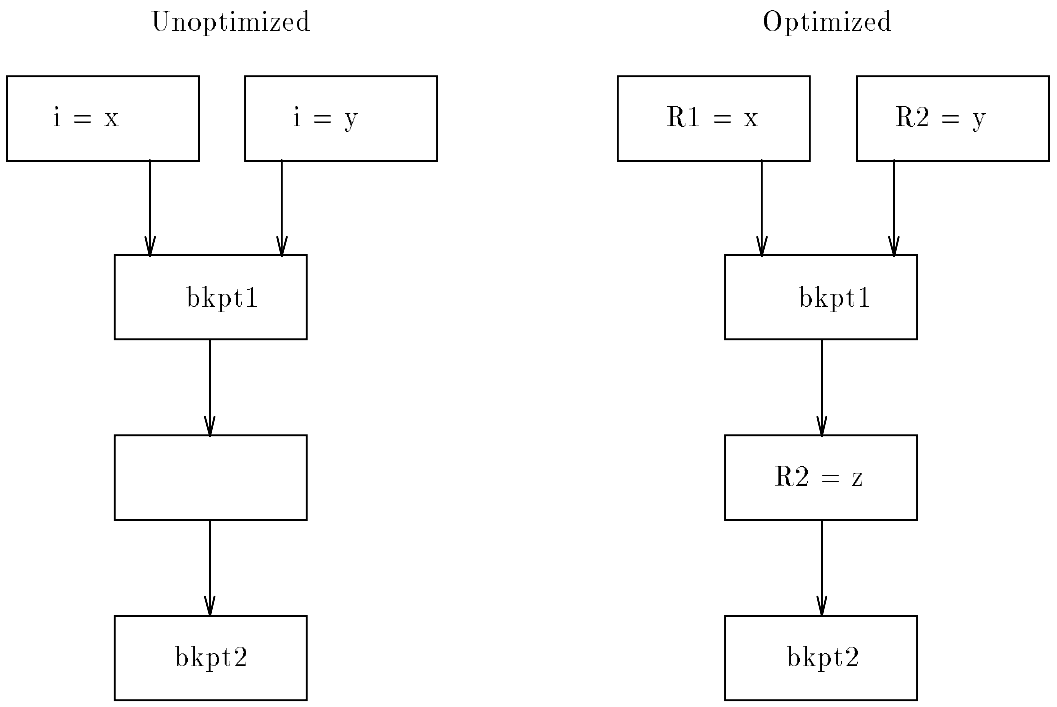


Figure 26.2: At `bkpt1`, `i` is current, but is it in `R1` or in `R2`? At `bkpt2`, `i` is endangered, and is only in `R2`.

27 Summary of Part III

The mapping between statements and breakpoints used for unoptimized code is problematic for optimized code. If such a mapping is used by a debugger on optimized code, the debugger is likely to mislead the debugger user. This work has described a mapping between statements and breakpoints that provides a reasonable approximation to what the naive user would expect when used on optimized code (and provides exactly what the naive user would expect on unoptimized code). The mapping allows the debugger user to break where a statement occurs or execute a statement at a time on a program in which statements may have been reordered and instructions generated from a statement are not necessarily contiguous. The mapping enables debugger behavior that more closely approximates the behavior provided by current debuggers on unoptimized code than other proposed mappings, and thereby neither requires debugger users to be experts on optimization nor requires users to modify their debugging strategies.

Using any such mapping, optimization can cause a debugger to provide an unexpected and potentially misleading value when asked to display an endangered variable. A debugger must be able to determine the currency of a variable if it is to provide truthful behavior on optimized code. Other researchers have given solutions to special cases of the currency determination problem. Sections 17 and 25 describe a general solution to the problem for a large class of sequential optimizations, including optimizations that modify the shape of the flow graph. These results hold in the presence of both local and global optimizations, including those listed in Table 27.1, and require no information about which optimizations have been performed.

This work has described the information a compiler must make available to the debugger for this task, as well as the nature of the information the debugger can provide to the debugger user when the user asks for the value of an endangered variable.

For most optimizations, the results described in this paper are precise (i.e., a variable claimed to be current is current, a variable claimed to be endangered is endangered, etc.). In some circumstances involving assignments through pointers, there is a trade-off between rare non-conservative results and more common non-conservative results, which is discussed

Representative Optimizations		
inlining	dead store elimination	partial redundancy elimination
code hoisting	strength reductions	local instruction scheduling
constant folding	constant propagation	global instruction scheduling
cross-jumping	copy propagation	local common subexpression elimination
loop unrolling	dead code elimination	global common subexpression elimination
	other code motion	induction-variable elimination

Table 27.1: The currency determination technique is applicable in the presence of any sequential optimizations, including all of the listed optimizations, that either do not modify the flow graph of the program or modify the flow graph in a constrained manner. Blocks may be added, deleted, coalesced, or copied; edges may be deleted, but control flow may not be radically changed. As an example of an optimization that does not observe the constraints, it does not apply to a portion of a program that contains interchanged loops.

at length in Section 20. There are two other circumstances in which the results are conservative:

- when a variable is current along all feasible paths but noncurrent along some infeasible path, in which case it will be claimed to be endangered.
- when a variable is endangered along some path due to an assignment through an alias, but there is no execution in which that path is taken and that variable is affected by that assignment.

27.1 Future Work

Currency determination at semantic breakpoints remains an open question (this topic is discussed in Section 25).

Once a debugger user has found a suspicious variable (one that due to program logic, not optimization, contains an unexpected value), the next question is 'How did it get that value?'. The sets of reaching definitions used for currency determination can be used in a straightforward manner to answer this question ('`x was set at one of lines 323 or 351`'). One direction for future research is how to efficiently be even more helpful; how to give responses such as '`x was set at line 566 to foo(y,z). At that point, z had the value 3.141 (set at line 370) and y had the value 17; y was set at line 506 to y+bar(w).`'. This was called *flowback analysis* by Balzer [Bal69], and has been investigated by others ([MC91], [Kor88]); reaching sets may be adaptable to this purpose.

Another research direction is dynamic currency determination, which is how a debugger can collect the minimal execution history information needed to determine whether an endangered variable is current or noncurrent when execution is suspended at a breakpoint. Useful in conjunction with this or as an alternative is recovery, which is to have the debugger compute and display the value that a variable would have had if optimization had not endangered the variable. Finally, an exciting possibility is extending the breakpoint model and currency determination techniques to parallel code, which is rife with noncurrent variables.

28 Conclusion

Debugging optimized code is a difficult problem, but it is worth solving for the following reasons:

- It is not always possible to completely debug an unoptimized version of a program. Examples have been given in which optimization changes the behavior of a program even when the optimizer is correct. This is not a new result, but such examples have not previously appeared in the literature.
- Source-level debuggers give misleading information in the presence of optimization, and will continue to do so without additional capabilities and support from compilers.
- Both source-level debugging based on misleading information and assembly-level debugging are more time consuming than source-level debugging based on accurate information.

These facts clearly demonstrate that additional debugger capabilities and compiler support for debugging optimized code are needed to lower the cost of software production.

I have described new debugger capabilities and the compiler support on which they are based that make it possible for debuggers to present accurate information in circumstances where formerly they would present misleading information:

- When no frame pointer is set up within the stack frame of some active subroutine—Part II describes several ways to support a expected behavior from a debugger's call-stack trace facility in this circumstance. These methods vary in their costs: there is a trade-off between run-time overhead for the debugger and required symbol table and compiler support. One method has been implemented and results are given.
- When the instructions generated from a statement are not contiguous—Section 15 describes a breakpoint model that enables truthful behavior in this case.
- When optimization has caused the storage location for a variable to contain an unexpected value—Part III describes a graph data structure to track optimizations that modify a program's flow graph, and algorithms on that data structure that determine whether a variable is current at a breakpoint. An architectural design is given from which an implementation should be straightforward, and the correctness of the algorithms is shown.

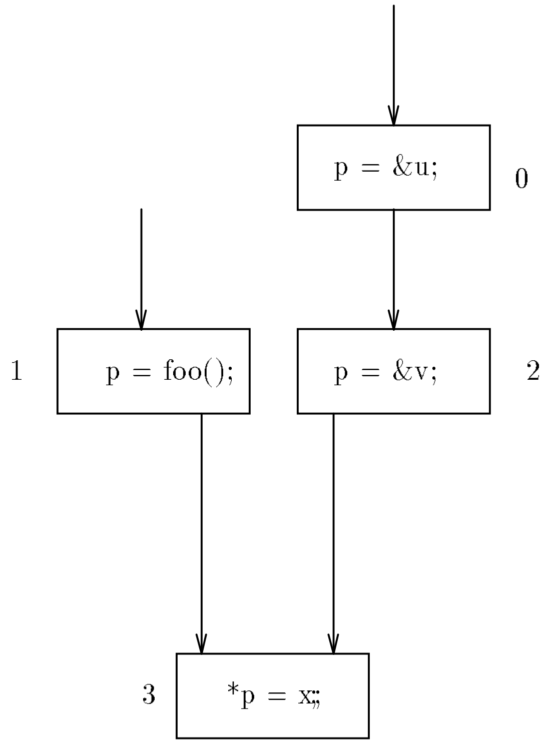


Figure A.1: If `u` and `v` are dead on exit from block 2, the assignment to `p` in block 2 can be eliminated, thus `p` is endangered at its use in block 3, without violating program semantics. Presumably, `foo()` can return the address of some variable that is live in block 3.

A Can a Noncurrent Pointer be Assigned Through?

Must it be the case that `p` is current at a trasnparent assignment through `p` (such as `*p = x`)? After all, How can a compiler make an assignment through a pointer that is not current? It would seem as though that would violate program semantics. While it may be unlikely, it is possible to arrive at an example in which such an assignment is legal. An example is given in Figure A.1.

Though perhaps unlikely, it is conceivable that V could be endangered because of such a situation. The difficulty for currency determination is that if V is endangered in this manner, it is due not to optimization affecting definitions of V, which is what we have been dealing with so far, but to optimization affecting definitions of `p`.

While in some cases it would be easy to determine that p is current, in general if in order to determine the currency of one variable we need to know the currency of another variable, we have a costly and potentially circular definition of currency determination. The circumstances under which a compiler can make a transparent definer (here `p`) noncurrent at the point of the transparent definition are sufficiently constrained that many compilers will never do it. It may be that those that do will, through the pointer analysis, determine `p`'s currency at subsequent assignments through `p`. Returning to Figure A.1, perhaps the compiler could determine for what variables `p` is noncurrent (`u` and `v`) and at what transparent definition (`*p = x;` in block 3). The compiler could tag that definition as unreliable for those variables.[34] However, the compiler's ability to produce this information is an open question. For the remainder of this work, I assume that transparent definers are current at each of their uses.

[34]If the compiler could determine a pointer's currency, why would it not determine an arbitrary variable's currency? First, for the purpose at hand, we need a pointer's currency only at subsequent uses of the pointer, whereas we need to be able to determine a variable's currency at any breakpoint. Second, if the compiler does determine a pointer's currency, it is through additional complex analysis.

B Proof of the Validity of the DS-graph

Theorem 17.6 relies on the correctness of Algorithm PRS, which in turn relies on the correctness of the DS-graph construction.

In this appendix I show that a DS-graph constructed as described in Section 17.3 is valid.

In Appendix C I show that the In sets are correctly computed: if there is a ds-pair in an In set, there is a path through the DS-graph along which that ds-pair reaches the entry of the block. Assume DS-graph node B selects source graph block B_u and object graph block B_o. I then show that $(d, s) \in \mathrm{In}_B^V$ means, loosely speaking, that there is a source graph path to B_u such that d reaches the entry of B_u along that path, also that there is a corresponding object graph path B_o such that s reaches the entry of B_o along the object graph path. This and an examination of how PRS_{Bk}^V is computed (where Bk is a breakpoint, not a node) allows us to conclude that $(d, s) \in \mathrm{PRS}_{Bk}^V$ iff d and s reach the breakpoint along corresponding paths.

Theorem 17.6 follows from this latter conclusion and the definitions of current, noncurrent, and endangered.

Let $|$ be a path concatenation operator with the natural semantics.

Object graph and DS-graph transformations are coupled, so that O_i refers to the object graph corresponding to DS_i.

Lemma B.1: If $Oselects_i(n) = Null$, then n has at most one predecessor and at most one successor.

Proof: By induction on the number of applications of DS-graph transformations.

Base Case:

A DS-graph DS_0 created according to the DS-graph Creation Rule contains no nodes n for which $Oselects(n) = Null$.

Inductive Hypothesis:

Assume that Lemma B.1 holds for any DS-graph DS_n following n graph transformations.

Inductive Step:

Show that Lemma B.1 holds for a DS-graph DS_{n+1} following $n + 1$ graph transformations.

Transformations 1, 3, and 4 do not introduce any nodes that do not select blocks in the object graph, do not increase the number of successors or predecessors of existing nodes that do not select blocks in the object graph, and do not change the object graph block selection of any node to $Null$. Thus the Lemma holds for these transformations by the inductive hypothesis.

Transformation 2:

This transformation does not increase the number of successors or predecessors of existing nodes that do not select blocks in the object graph. It does not change the object graph block selection of any node to $Null$.

It does remove a node that selects a block in the object graph and replaces it with copies that do not select any block in the object graph, but the replacements have a single predecessor and a single successor. It also replaces nodes that do not select a block in the object graph with copies that do not select any block in the object graph, but again, the replacements have a single predecessor and a single successor. This can be seen by considering the interaction between *NewEdges* and a replacement node $\overline{v}_{s,k} \in NewNodes$. $\overline{v}_{s,k}$ becomes the head of one new edge and the tail of one new edge.

Transformation 5:

This transformation does not increase the number of successors or predecessors of existing nodes that do not select blocks in the object graph. It does not change the object graph block selection of any node to $Null$.

It does remove the call node, which selects a block in the object graph, and replaces it with copies that do not select any block in the object graph, but the replacements have a single predecessor and a single successor. This can be seen by considering the interaction between *NewEdges* and a replacement node $\overline{Call}_k \in$ *CallNodes*. *Start* is $\overline{Call}_k$'s only successor, and the k^{th} predecessor of $Call$ is $\overline{Call}_k$'s only predecessor.

The transformation also copies a subgraph, but because the selection and connectivity of the copy matches the selection and connectivity of the subgraph of which it is a copy, the inductive hypothesis guarantees that the Lemma holds for nodes within the copy.

Transformation 6:

This transformation does not increase the number of successors or predecessors of existing nodes that do not select blocks in the object graph. It does not change the object graph block selection of any node to $Null$.

It may introduce nodes that do not select blocks in the object graph by making copies of the body of a loop. The selection and connectivity of each copy matches the selection and connectivity of the loop of which it is a copy, with the exception that copies of back edges and copies of edges out of the loop are replaced. They are replaced one-for-one, so the selection and number of edges in and out of each node in the copy matches the selection and number of edges in and out of the node of which it is a copy, so the inductive hypothesis guarantees that the Lemma holds for nodes within the copy.

□

We overload the function *Sselects* to take a subpath in DS_i and return a subpath in S, and we overload *Oselects* to take a subpath in DS_i and return a subpath in O_i, in the natural way.

Theorem B.2: A DS-graph created according to the DS-graph Creation Rule and modified by iterative application of DS-graph transfomations is valid.

Recall the definition of valid:

A DS-graph is valid if and only if every path p through the DS-graph denotes a path-pair, that is, the pair of paths $< p_u, p_o >$ selected by p is a path pair, and every feasible path-pair is denoted by some path through the DS-graph.

Proof: By induction on the number of applications of DS-graph transformations.

Base Case:

Let DS_0 be a DS-graph created according to the DS-graph Creation Rule. DS_0, the source graph, and object graph are isomorphic – clearly the DS-graph is valid.

Inductive Hypothesis:

Assume that a DS-graph DS_n created according to the DS-graph Creation Rule and subsequently modified by n graph transformations is valid.

Inductive Step:

Show that a DS-graph DS_{n+1} created according to the DS-graph Creation Rule and subsequently modified by $n + 1$ graph transformations is valid.

$Sselects_n = Sselects_{n+1}$ and $Oselects_n = Oselects_{n+1}$ except where explicitly mentioned. $Sselects_{n+1}$ subsumes $Sselects_n$; its domain may be larger because the $n+1^{st}$ transformation may introduce DS-graph nodes, but its value on a node present in DS_n is the same as the value of $Sselects_n$ on that node.

Let p be a path in DS_{n+1} where $Sselects_{n+1}(p) = p_u$ and $Oselects_{n+1}(p) = p_o$.

If p is present in DS_n and $Oselects_n(p) = p_o$ then by the inductive hypothesis, p denotes a path-pair (since $Sselects_{n+1}$ subsumes $Sselects_n$). Otherwise, we have a case analysis based on which transformation was last applied.

Transformation 1:

Since p was affected by the transformation, p can be expressed as:

$p = prefix| < a_s, \overline{n}_s, b_s > |suffix$

for some $s \in Subpaths$, where *prefix* does not contain any nodes in *NewNodes*.

If p is not a simple path, *suffix* may contain nodes in *NewNodes*. Induction on the number of nodes in *NewNodes* contained in p can be used to show that p denotes a path-pair. I present only the base case here, in which *suffix* contains no nodes in *NewNodes*.

$p_u = Sselects_{n+1}(prefix)| < Sselects_{n+1}(a_s), Sselects_{n+1}(b_s) > |Sselects_{n+1}(suffix)$

$p_o = Oselects_{n+1}(prefix)| < Oselects_{n+1}(a_s), \overline{b}, Oselects(b_s) > |Oselects_{n+1}(suffix)$

$\exists p' = prefix| < a_s, b_s > |suffix$, in DS_n

$p'_u = Sselects_n(prefix)| < Sselects_n(a_s), Sselects_n(b_s) > |Sselects_n(suffix)$

$p'_o = Oselects_n(prefix)| < Oselects_n(a_s), Oselects_n(b_s) > |Oselects_n(suffix)$

On input I,

1. p_o is taken iff p'_o is taken, by semantic constraint.
2. p'_u is taken iff p'_o is taken, by the inductive hypothesis.
3. p_u is taken iff p'_u is taken, because $p_u = p'_u$.
4. p_u is taken iff p_o is taken, by 1, 2, and 3.

Therefore p denotes a path-pair.

It remains to show that every feasible path-pair through S and O_{n+1} is denoted by some path through DS_{n+1}. Assume $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_{n+1} taken on input I.

If $\overline{p}_o$ does not contain $\overline{b}$, then $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_n, and is denoted by some path $\overline{p}$ through DS_n, by the inductive hypothesis. $\overline{p}$ is not modified by this transformation and thus is a path through DS_{n+1} also, and denotes $< \overline{p}_u, \overline{p}_o >$ in S and O_{n+1}.

If $\overline{p}_o$ does contain $\overline{b}$, then assume without loss of generality that $\overline{p}_o = p_o$. Then p'_o is in O_n, which by the inductive hypothesis implies p' is in DS_n denoting path-pair $< p'_u, p'_o >$ through S and O_n taken on I. The DS-graph transformation constructs p in DS_{n+1} from p', where p does indeed select p_o. By 3, p_u is taken on I and therefore $\overline{p}_u = p_u$. Thus p is a path through D_{n+1} that denotes path-pair $< \overline{p}_u, \overline{p}_o >$.

Transformation 2:

Since p was affected by the transformation, p can be expressed as:

$p = prefix|\overline{s}|suffix$

where $\overline{s}$ is composed of nodes in *NewNodes* and is a copy of some $s \in Subpaths$, and *prefix* does not contain any nodes in *NewNodes*

If p is not a simple path, *suffix* may contain copies of paths in *Subpaths*. Induction on the number of copies of paths in *Subpaths* contained in p can be used to show that p denotes a path-pair. I present only the base case here, in which *suffix* contains no nodes in *NewNodes*.

$Oselects_{n+1}(\overline{s}) =< h, t >$, for some $h \in H^O$ and some $t \in T^O$.

$Oselects_n(s) =< h, b, t >$

$p_u = Sselects_{n+1}(prefix)|Sselects_{n+1}(\overline{s})|Sselects_{n+1}(suffix)$

$p_o = Oselects_{n+1}(prefix)| < h, t > |Oselects_{n+1}(suffix)$

$\exists p' = prefix|s|suffix$, in DS_n

$p'_u = Sselects_n(prefix)|Sselects_n(s)|Sselects_n(suffix)$

$p'_o = Oselects_n(prefix)| < h, b, t > |Oselects_n(suffix)$

$Sselects_{n+1}(\overline{s}) = Sselects_n(s)$

On input I,

1. p_o is taken iff p'_o is taken, by semantic constraint.
2. p'_u is taken iff p'_o is taken, by the inductive hypothesis.
3. p_u is taken iff p'_u is taken, because $p_u = p'_u$.
4. p_u is taken iff p_o is taken, by 1, 2, and 3.

Therefore p denotes a path-pair.

It remains to show that every feasible path-pair through S and O_{n+1} is denoted by some path through DS_{n+1}. Assume $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_{n+1} taken on input I.

If $\overline{p}_o$ does not contain $< h, t >$ for $h \in H^O$ and $t \in T^O$, then $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_n, and is denoted by some path $\overline{p}$ through DS_n, by the inductive hypothesis. $\overline{p}$ is not modified by this transformation and thus is a path through DS_{n+1} also, and denotes $< \overline{p}_u, \overline{p}_o >$ in S and O_{n+1}.

If $\overline{p}_o$ does contain $< h, t >$, then assume without loss of generality that $\overline{p}_o = p_o$. Then p'_o is in O_n, which by the inductive hypothesis implies p' is in DS_n denoting path-pair $< p'_u, p'_o >$ through S and O_n taken on I. The DS-graph transformation constructs p in DS_{n+1} from p', where p does indeed select p_o. By 3, p_u is taken on I and therefore $\overline{p}_u = p_u$. Thus p is a path through D_{n+1} that denotes path-pair $< \overline{p}_u, \overline{p}_o >$.

Transformation 3:

Every path in DS_{n+1} denotes a path-pair by the inductive hypothesis, since every path in DS_{n+1} is present in DS_n.

Assume $< p_u, p_o >$ is a feasible path-pair through S and O_{n+1} that is not denoted by any path through DS_{n+1}. $< p_u, p_o >$ is also a feasible path-pair through S and O_n. By the inductive hypothesis there is a path p through DS_n that denotes $< p_u, p_o >$. If p were present in DS_{n+1}, it would denote $< p_u, p_o >$, so at least one edge in p must be removed by the DS-graph transformation. Let e be the first such edge in p.

No internal node in a subpath $s \in SubPaths$ selects a block in the object graph, so by Lemma B.1, each edge on s is on no other subpath in *SubPaths*.

Suppose e is in *DelEdges*. Then e is on exactly one $s \in SubPaths$, and p contains s. $Oselects_n(s)$ is the subpath in O_n consisting of the edge that gets deleted from O_n. $Oselects_n(s)$ is in p_o, so p_o contains the deleted edge. This contradicts the assumption that $< p_u, p_o >$ is a feasible path-pair through S and O_{n+1}.

e must then be in the transitive closure of $\tau(\mathit{DelEdges})$. Then the edge prior to e on p is in the transitive closure of $\tau(\mathit{DelEdges})$, so the edge prior to e on p is removed, contradicting the assumption that e is the first edge on p that is removed.

Transformation 4:

If p is not a simple path, it may contain multiple nodes that select the block boundary marker M. Induction on the number of such nodes can be used to show that p denotes a path-pair. Again, I present only the base case, in which p contains a single such node:

p is present in DS_n. Let $p'_u = \mathit{Sselects}_n(p)$ and $p'_o = \mathit{Oselects}_n(p)$. $< p'_u, p'_o >$ is a path-pair through S and O_n by the inductive hypothesis.

$p_o = \mathit{prefix}| < h, M, x > |\mathit{suffix}$ for some $x \in E^O$

$p'_o = \mathit{prefix}| < h, t, x > |\mathit{suffix}$

It follows from the semantic constraint on the object graph transformation that if $< h, t, x >$ is taken in O_n on some input I, then $< h, x >$ is taken in O_{n+1} on I. Because M selects a position internal to h, for purposes of block entry M can be ignored, so p_o is taken on the same set of inputs as p'_o.

On input I,

1. p_o is taken iff p'_o is taken, by semantic constraint.
2. p'_u is taken iff p'_o is taken, by the inductive hypothesis.
3. p_u is taken iff p'_u is taken, because $p_u = p'_u$.
4. p_u is taken iff p_o is taken, by 1, 2, and 3.

Therefore p denotes a path-pair.

The object graph transformation introduces no new paths. The DS-graph transformation maintains the relationship between DS-graph paths and feasible paths, and removes no DS-graph paths. Therefore every feasible path-pair through S and O_{n+1} is denoted by some path through DS_{n+1}.

Transformation 5:

Since p was affected by the transformation, p can be expressed as:

$p = \mathit{prefix}| < \overline{Call} > |\overline{s}| < Succ > |\mathit{suffix}$,

where $\overline{Call}$ is the first copy of $Call$ on p,

and $\overline{s}$ is composed of nodes in $\mathit{InlinedNodes}$.

If p is not a simple path, suffix may contain nodes in $\mathit{CallNodes}$ and $\mathit{InlinedNodes}$. Induction on the number of nodes in $\mathit{CallNodes}$ contained in p can be used to show that p denotes a path-pair. I present only the base case here, in which suffix does not contain nodes in $\mathit{CallNodes}$ or $\mathit{InlinedNodes}$:

$p_u = \mathit{Sselects}_{n+1}(\mathit{prefix})| < \mathit{Sselects}_{n+1}(\overline{Call}) > |\mathit{Sselects}_{n+1}(\overline{s})| < \mathit{Sselects}_{n+1}(Succ) > |\mathit{Sselects}_{n+1}(\mathit{suffix})$

$p_o = \mathit{Oselects}_{n+1}(\mathit{prefix})|\mathit{Oselects}_{n+1}(\overline{s})| < \mathit{Oselects}_{n+1}(Succ) > |\mathit{Oselects}_{n+1}(\mathit{suffix})$

$\exists p' = \mathit{prefix}| < Call > |s| < Succ > |\mathit{suffix}$, in DS_n,

where $Call$ is the only instance of $Call$ on p, and s is composed of nodes in the to-be-inlined subroutine. The i^{th} node in $\overline{s}$ is a copy of the i^{th} node in s.

$p'_u = \mathit{Sselects}_n(\mathit{prefix})| < \mathit{Sselects}_n(Call) > |\mathit{Sselects}_n(s)| < \mathit{Sselects}_n(Succ) > |\mathit{Sselects}_n(\mathit{suffix})$

$p'_o = \mathit{Oselects}_n(\mathit{prefix})| < \mathit{Oselects}_n(Call) > |\mathit{Oselects}_n(s)| < \mathit{Oselects}_n(Succ) > |\mathit{Oselects}_n(\mathit{suffix})$

$Sselects_{n+1}(\overline{Call}) = Sselects_n(Call)$
$Sselects_{n+1}(\overline{s}) = Sselects_n(s)$
$Oselects_{n+1}(\overline{Call}) = null$

On input I,

1. p_o is taken iff p'_o is taken, because $\overline{s}$ is the inlined version of $< Call > |s$.
2. p'_u is taken iff p'_o is taken, by the inductive hypothesis.
3. p_u is taken iff p'_u is taken, because $p_u = p'_u$.
4. p_u is taken iff p_o is taken, by 1, 2, and 3.

Therefore p denotes a path-pair.

It remains to show that every feasible path-pair through S and O_{n+1} is denoted by some path through DS_{n+1}. Assume $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_{n+1} taken on input I.

If $\overline{p}_o$ does not contain any nodes in *CallNodes* or *InlinedNodes*, then $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_n, and is denoted by some path $\overline{p}$ through DS_n, by the inductive hypothesis. $\overline{p}$ is a path through DS_{n+1} also, and denotes $< \overline{p}_u, \overline{p}_o >$ in S and O_{n+1}.

Otherwise, assume without loss of generality that $\overline{p}_o = p_o$. Then p'_o is in O_n, which by the inductive hypothesis implies p' is in DS_n denoting path-pair $< p'_u, p'_o >$ through S and O_n taken on I. The DS-graph transformation constructs p in DS_{n+1} from p', where p does indeed select p_o. By 3, p_u is taken on I and therefore $\overline{p}_u = p_u$. Thus p is a path through D_{n+1} that denotes path-pair $< \overline{p}_u, \overline{p}_o >$.

Transformation 6:

Since p was affected by the transformation, p can be expressed as:

$p = prefix|\overline{s}| < Succ > |suffix$,
where $\overline{s}$ is composed of nodes in *NewNodes* and consists of the loop body unrolled i times.

If p is not a simple path, *suffix* may contain nodes in *NewNodes*. Induction on the number of sequences of nodes in *NewNodes* contained in p can be used to show that p denotes a path-pair. I present only the base case here, in which *suffix* does not contain nodes in *NewNodes*:

$p_u = Sselects_{n+1}(prefix)|Sselects_{n+1}(\overline{s})| < Sselects_{n+1}(Succ) > |Sselects_{n+1}(suffix)$
$p_o = Oselects_{n+1}(prefix)|Oselects_{n+1}(\overline{s})| < Oselects_{n+1}(Succ) > |Oselects_{n+1}(suffix)$
$\exists p' = prefix|s| < Succ > |suffix$, in DS_n,
where s consists of i traversals of the to-be-unrolled loop.
The i^{th} node in $\overline{s}$ is a copy of the i^{th} node in s.
$p'_u = Sselects_n(prefix)|Sselects_n(s)| < Sselects_n(Succ) > |Sselects_n(suffix)$
$p'_o = Oselects_n(prefix)|Oselects_n(s)| < Oselects_n(Succ) > |Oselects_n(suffix)$
$Sselects_{n+1}(\overline{s}) = Sselects_n(s)$

On input I,

1. p_o is taken iff p'_o is taken, because $\overline{s}$ is the unrolled version of s.
2. p'_u is taken iff p'_o is taken, by the inductive hypothesis.
3. p_u is taken iff p'_u is taken, because $p_u = p'_u$.
4. p_u is taken iff p_o is taken, by 1, 2, and 3.

Therefore p denotes a path-pair.

It remains to show that every feasible path-pair through S and O_{n+1} is denoted by some path through DS_{n+1}. Assume $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_{n+1} taken on input I.

If $\overline{p}_o$ does not contain any nodes in *NewNodes*, then $< \overline{p}_u, \overline{p}_o >$ is a feasible path-pair through S and O_n, and is denoted by some path $\overline{p}$ through DS_n, by the inductive hypothesis. $\overline{p}$ is a path through DS_{n+1} also, and denotes $< \overline{p}_u, \overline{p}_o >$ in S and O_{n+1}.

Otherwise, assume without loss of generality that $\overline{p}_o = p_o$. Then p'_o is in O_n, which by the inductive hypothesis implies p' is in DS_n denoting path-pair $< p'_u, p'_o >$ through S and O_n taken on I. The DS-graph transformation constructs p in DS_{n+1} from p', where p does indeed select p_o. By 3, p_u is taken on I and therefore $\overline{p}_u = p_u$. Thus p is a path through D_{n+1} that denotes path-pair $< \overline{p}_u, \overline{p}_o >$.

$\square$

C Proof of Correctness of the Data Flow Algorithms

In Section C.1 I first show that **Algorithm PRS** and **Algorithm PRS^{CT}** are correct, then that **Algorithm PRS-BK** and **Algorithm PRS-BK^{CT}** are correct. These proofs naturally fall together because **Algorithm PRS** is a special case of **Algorithm PRS^{CT}** and **Algorithm PRS-BK** is a special case of **Algorithm PRS-BK^{CT}**. I then show that Theorem 17.6 follows from this latter conclusion and the definitions of current, noncurrent, and endangered.

In Section C.2 I show that **Algorithm PRS^{T}** is correct, then than **Algorithm PRS-BK^{T}** is correct. I then show that Theorem 19.8 follows from this latter conclusion and the definitions of current, noncurrent, and endangered.

C.1 Proof of Correctness of Algorithm PRS and Algorithm PRS^{CT}

Lemma C.1: Initialize and Initialize^{CT} terminate with $(d, s) \in \text{Gen}_B^V$ iff either $d = null$ and there are no definitions of V in B's definition list or d is a definition of V in B's definition list and d reaches the exit of B, and either $s = null$ and there are no stores into V in B's definition list or s is a store into V in B's store list and s reaches the exit of B.

Proof: Inspection of Initialize and Initialize^{CT}, coupled with the assumption that the definition and store lists for each block are correct, is sufficient to show that the Lemma holds.

□

Theorem C.2:

Algorithm PRS and **Algorithm PRS^{CT}** terminate with $e \in \text{In}_B^V$ iff there is a path p to B in the DS-graph such that e reaches the entry of B along p.

Algorithm PRS is a special case of **Algorithm PRS^{CT}** where no transparent definitions are present. I prove the Theorem for the general case (**Algorithm PRS^{CT}**). To apply the proof to **Algorithm PRS**, convert the Gen elements in that algorithm to Gen sets that have a single element, and assume that all definitions and stores are opaque.

Proof: **Termination**

Initialize and Initialize^{CT} terminate because there are a finite number of variables and definitions.

Iterate terminates because

- no element is ever removed from an In or Out set,
- an element is added to at least one In or Out set on every iteration but the last, and
- elements are from definitions × stores, which is a finite set.

Only if:

If e is in In_B^V, then there is a path p to B such that e reaches the entry of B along p.

I will show the existence of the desired path using induction on the number of iterations before e is placed in In_B^V the first time.

Base Case: e is placed into In_B^V for the first time on the second iteration.

By line 2 of Iterate, In_B^V and Out_B^V are empty before the first iteration. By line 5 of Iterate and because Out_B^V is empty the first time through, In_B^V is empty immediately after the first iteration. Since e is placed into In_B^V in the second iteration, e is placed into Out_P^V in the first iteration, for some predecessor P of B, (line 5 of Iterate). Because In_P^V is empty in the first iteration, line 7 of **Iterate** reduces to $\text{Out}_P^V = \mathit{Complete}(\text{Gen}_P^V)$.

By Lemma C.1 $(d, s) \in \mathrm{Gen}_P^V$ implies d reaches the exit of P and s reaches the exit of P. Let p' be a path to P. p' exists because if there were no path to P, the edge from P to B would have been deleted (DS-graph transformation 3). Let p be $p'| < P, B >$. Then e reaches the entry of B along p.

Inductive hypothesis: Assume that if e is placed into In_B^V for the first time on the i^{th} iteration for $2 \leq i < n$, then there is a path p to B such that e reaches the entry of B along p.

Inductive step: Show that if e is placed into In_B^V for the first time on the n^{th} iteration, then there is a path p to B such that e reaches the entry of B along p.

If e is placed into In_B^V for the first time on the n^{th} iteration, then by line 5 of Iterate, $e \in \mathrm{Out}_P^V$ for some predecessor P of B for the first time in iteration $n - 1$.

e's presence in Out_P^V is due to the execution of line 7 of **Iterate**. If In_P^V were empty in iteration $n - 1$, then as in the base case, Gen_P^V would have to contain e. But then e would have been placed into In_B^V for the first time on the 2^{nd} iteration, contrary to our assumption, so e is not in Gen_P^V and In_P^V is not empty in iteration $n - 1$.

Since In_P^V is non-empty in iteration $n-1$, by definition of $\overset{\kappa}{\cup}$, $\exists f \in \mathrm{In}_P^V$ in iteration $n-1$ such that $e \in f \,\kappa\, \mathrm{Gen}_P^V$. By the inductive hypothesis, there is a path p' to P such that f reaches the entry of P along p'. Let p be $p'| < B >$. Let $e = (d, s)$. We have a case analysis based on the ways (e) can be placed into Out_P^V.

Case 1: $(null, null) \in \mathrm{Gen}_P^V$ and $f = e$. By Lemma C.1, B contains no definitions of V and no stores into V, so e reaches P's exit.

Case 2: $(null, s) \in \mathrm{Gen}_P^V$ and $f = (d, s')$. By Lemma C.1, s reaches P's exit and B contains no definitions of V. Then d also reaches P's exit, so e reaches P's exit.

Case 3: $(d, null) \in \mathrm{Gen}_P^V$ and $f = (d', s)$. By Lemma C.1, d reaches P's exit and B contains no stores into V. Then s also reaches P's exit, so e reaches P's exit.

By choice of P and p, e reaches the entry of B along p. Note that if e is transparent, Case 1 must apply.

If:

If there is a path p to B in the DS-graph such that e reaches the entry of B along p, then e is in In_B^V.

The proof is by induction on the length of p.

Base Case: $|p| = 1$. Then B's predecessor P is the start node. Initialize or Initialize^{CT} will place e into Gen_P^V (where $e = (d\text{-}init, s\text{-}init)$). In the first iteration, e is placed into Out_P^V by line 7 of Iterate. In the second and all subsequent iterations, e is placed into In_B^V by line 5 of Iterate.

Inductive hypothesis: Assume that if there is a path p to B of length i, $1 \leq i < n$, such that e reaches the entry of B along p, then $e \in \mathrm{In}_B^V$.

Inductive step: Show that if there is a path p to B of length n such that e reaches the entry of B along p, then $e \in \mathrm{In}_B^V$.

Let P be the predecessor of B on p. Because e reaches the entry of B along p, e reaches the exit of P along p. We have a case analysis of the different ways that e can reach the exit of P along p.

Case 1: $e = (d, s)$ where d and s are in P. Since e reaches the exit of P, by Lemma C.1, $e \in \mathrm{Gen}_P^V$. In every iteration, Out_P^V is placed into $\{e\}$ by line 7 of Iterate. In the second and all subsequent iterations, $\{e\}$ is placed into In_B^V by line 5 of Iterate.

Case 2: $e = (d, s)$ where d is in P and s is not. Then e is opaque. Since e reaches the exit of P, (d', s) reaches the entry of P, for some d', and by Lemma C.1, $(d, null) \in \mathrm{Gen}_P^V$. By the inductive hypothesis, $(d', s) \in \mathrm{In}_P^V$. Assume (d', s) was first placed into In_P^V in iteration j. $(d', s)\,\kappa\,(d, null) = (d, s) = e$, so line 7 of Iterate places e into Out_P^V in iteration j and all subsequent iterations. In iteration $j+1$ and all subsequent iterations, line 5 of Iterate places e into In_B^V.

Case 3: $e = (d, s)$ where s is in P and d is not. Then e is opaque. Since e reaches the exit of P, (d, s') reaches the entry of P, for some s', and by Lemma C.1, $(null, s) \in \mathrm{Gen}_P^V$. By the inductive hypothesis, $(d, s') \in \mathrm{In}_P^V$. Assume (d, s') was first placed into In_P^V in iteration j. $(d, s')\,\kappa\,(null, s) = (d, s) = e$, so line 7 of Iterate places e into Out_P^V in iteration j and all subsequent iterations. In iteration $j+1$ and all subsequent iterations, line 5 of Iterate places e into In_B^V.

Case 4: $e = (d, s)$ where neither d nor s arE in P. Then e reaches the entry of P. Since e reaches the exit of P, by Lemma C.1, $(null, null) \in \mathrm{Gen}_P^V$. By the inductive hypothesis, $e \in \mathrm{In}_P^V$. Assume e was first placed into In_P^V in iteration j. $e\,\kappa\,(null, null) = e$, so line 7 of Iterate places e into Out_P^V in iteration j and all subsequent iterations. In iteration $j+1$ and all subsequent iterations, line 5 of Iterate places e into In_B^V.

□

The following theorem relies on the assumption that a store is not generated if the compiler can determine that the store would never be executed; in particular, stores are not generated for unreachable blocks. If B is an unreachable block, then In_B^V remains empty (by DS-graph transformation 3). Gen_{Bk}^V contains a null, because B's store list is empty. [35]

Until this point, it has not been important to distinguish an entry in a definition or store list from the representative instruction in the source or object graph that it denotes. The distinction is relevant for the following lemma and theorem. Following now-established practice, for a definition list entry d, let d_u be the representative instruction in the source graph that it denotes. For a store list entry s, let s_o be the representative instruction in the object graph that it denotes. For Bk a breakpoint in the DS-graph, let Bk_u denote the breakpoint location in the source graph, and Bk_o denote the breakpoint location in the object graph.

Lemma C.3: **Initialize-BK** and **Initialize-BK^{CT}** terminate with $(d, s) \in \mathrm{Gen}_{Bk}^V$ iff either $d = null$ and there are no definitions of V in B_u prior to Bk_u or d is a definition of V prior to Bk_u in B_u and d reaches Bk_u, and either $s = null$ and there are no stores into V in B_o prior to Bk_o or s is a store into V prior to Bk_o in B_o and s reaches Bk_o.

Proof: This can be seen by Inspection of **Initialize-BK** and **Initialize-BK^{CT}**, coupled with the assumption that the definition and store lists for each block are correct.

□

Theorem C.4: For a syntactic breakpoint Bk, $(d, s) \in \mathrm{PRS}_{Bk}^V$ iff there is a path-pair p to Bk such that d_u is a definition of V that reaches Bk_u along p_u and s_o is a store into V that reaches Bk_o along p_o.

[35] If this assumption is violated, the proof does not go through, but it is not clear that there are any bad results in practice. We may have nonempty Out_B^V sets for unreachable nodes, but since the edges out of those nodes have been eliminated, no incorrect information flows from them, and no breakpoint in such a node can be reached, so no query about the currency of a variable will be asked at such a node.

Proof: Let B be the node in the DS-graph containing Bk. The fact that Bk is a syntactic breakpoint guarantees (by definition) that Bk is in B's definition list and in B's store list, and also that B selects a block B_u in the source graph and a block B_o in the object graph. The assumption that definition lists and store lists are correct guarantee that Bk_u appears in B_u and Bk_o appears in B_o.

If: Assume $(d, s) \in \text{PRS}^V_{Bk}$ where Bk is a syntactic breakpoint. I show by a case analysis of how (d, s) can be placed into PRS^V_{Bk} that there is a path-pair p to Bk such that d_u is a definition of V that reaches Bk_u along p_u and s_o is a store into V that reaches Bk_o along p_o.

Case 1: $(null, null) \in \text{Gen}^V_{Bk}$ and $(d, s) \in \text{In}^V_B$. By Theorem C.2, there is a path p to B such that (d, s) reaches the entry of B along p. By Lemma C.3, B contains no definition of V and no store into V, and thus (d, s) reaches Bk.

Case 2: $(d, null) \in \text{Gen}^V_{Bk}$ and $(x, s) \in \text{In}^V_B$ (for some x). By Lemma C.3, d reaches Bk along every path to Bk and B contains no store into V. By Theorem C.2, there is a path p to B such that (x, s) reaches the entry of B along p, and because B contains no store into V, s reaches Bk along p.

Case 3: $(null, s) \in \text{Gen}^V_{Bk}$ and $(d, x) \in \text{In}^V_B$. By Lemma C.3, s reaches Bk along every path to Bk and B contains no definition of V. By Theorem C.2, there is a path p to B such that (d, x) reaches the entry of B along p, and because B contains no definition of V, d reaches Bk along p.

Case 4: $(d, s) \in \text{Gen}^V_{Bk}$. Gen^V_{Bk} contains a store, so B is reachable. Because an initial ds-pair is placed in Gen^V_{Start}, some ds-pair is in In^V_B. Let $(x, y) \in \text{In}^V_B$. By Theorem C.2, there is a path p to B such that (x, y) reaches the entry of B along p. By Lemma C.3, d and s reach Bk along every path through the DS-graph, and in particular, along p.

By Theorem B.2, p is a path-pair. By assumption, the definition lists and store lists are correct, so d_u reaches Bk_u along p_u, and s_o reaches Bk_o along p_o.

Note that if (d, s) is transparent, Case 1 or Case 4 must apply.

Only if: Assume there is a path-pair p to a syntactic breakpoint Bk such that d_u is a definition of V that reaches Bk_u along p_u and s_o is a store into V that reaches Bk_o along p_o. I show that $(d, s) \in \text{PRS}^V_{Bk}$.

The assumption that definition lists and store lists are correct imply that d reaches Bk along p and s reaches Bk along p, which in turn implies that (d, s) reaches Bk along p. We have a case analysis of the ways that (d, s) can reach Bk along p.

Case 1: Neither d nor s are in B, B contains no opaque definitions of V or opaque stores into V, and (d, s) reaches the entry of B along p. By Lemma C.3, $(null, null) \in \text{Gen}^V_{Bk}$. By Theorem C.2, $(d, s) \in \text{In}^V_B$.

Case 2: d is in B and s is not, no opaque stores into V are in B, and s reaches the entry of B along p. By Lemma C.3, $(d, null) \in \text{Gen}^V_{Bk}$. Some definition x of V reaches the entry of B along p (the initial definition, if no other). By Theorem C.2, $(x, s) \in \text{In}^V_B$.

Case 3: s is in B and d is not, no opaque definitions of V are in B, and d reaches the entry of B along p. By Lemma C.3, $(null, s) \in \text{Gen}^V_{Bk}$. Some store x into V reaches the entry of B along p. By Theorem C.2, $(d, x) \in \text{In}^V_B$.

Case 4: d and s are both in B. By Lemma C.3, $(d, s) \in \text{Gen}^V_{Bk}$. Some definition x of V and store y into V reach the entry of B along p. By Theorem C.2, $(x, y) \in \text{In}^V_B$.

Then by line 1 of **Algorithm PRS-BK** or **Algorithm PRS-BK**^{CT}, $(d,s) \in \mathrm{PRS}^V_{Bk}$.

□

I can now prove Theorem 17.6, restated here for convenience. Note that in Theorem 17.6, B represents a breakpoint, not a node. **Theorem 17.6:**

> $\mathrm{PRS}^V_B = \emptyset$ iff B is unreachable or V is not in scope in the flow graph component containing B. Otherwise:
>
> V is current at B iff $\forall (d,s) \in \mathrm{PRS}^V_B$, s was generated from d;
>
> V is endangered at B iff $\exists (d,s) \in \mathrm{PRS}^V_B$ such that s was not generated from d;
>
> V is noncurrent at B iff $\nexists (d,s) \in \mathrm{PRS}^V_B$ such that s was generated from d.

Proof: **Algorithm PRS** and **Algorithm PRS**^{CT} terminate with $e \in \mathrm{In}^V_B$ iff there is a path p to B in the DS-graph such that e reaches the entry of B along p.

Assume $\mathrm{PRS}^V_B = \emptyset$. Then by line 1 of **Algorithm PRS-BK** or **Algorithm PRS-BK**^{CT}, In^V_B is empty and $\mathit{Complete}(\mathrm{Gen}^V_{Bk}) = \emptyset$. If V is in scope at B, we assume an initial definition of V and an initial store into V, ensuring that at some definition of V and store into V reach B along every path. But then there would be some ds-pair in In^V_B. Since there is not, either there are no definitions of V or stores into V in the flow graph component, implying that V is not in scope, or there are no paths to B.

Assume B is unreachable. Then In^V_B is empty (by lines 2 and 4 of **Iterate**. We assume that there are no stores in an unreachable block, therefore $\mathit{Complete}(\mathrm{Gen}^V_{Bk}) = \emptyset$, so by line 1 of **Algorithm PRS-BK** or **Algorithm PRS-BK**^{CT}, $\mathrm{PRS}^V_B = \emptyset$.

Assume V is not in scope. Then there are no definitions of V or stores into V in any block in the flow graph component, implying that In^V_B is empty and $\mathrm{Gen}^V_{Bk} = (\mathit{null}, \mathit{null})$, so by line 1 of **Algorithm PRS-BK** or **Algorithm PRS-BK**^{CT}, $\mathrm{PRS}^V_B = \emptyset$.

Assume $\mathrm{PRS}^V_B \neq \emptyset$, V is in scope, and B is reachable. For any path pair p, Theorem C.4 and Definitions 16.8 and 16.10 relate the pair $(d,s) \in \mathrm{PRS}^V_B$ that reaches along p, along with whether s was generated from d, to V's currency. The bidirectional nature of the implication in Theorem C.4 gives the iteration over all paths required by Definitions 16.11, 16.13 and 16.12, and the theorem follows.

□

C.2 Proof of Correctness of Algorithm PRS^T

Lemma C.5: **Initialize**^{T} terminates with $\text{Gen}_B^V = (dl, sl)$ iff either $dl = null$ and B's definition list contains no definitions of V or $dl =< d_1, d_2, \ldots, d_m >$, each d_i appears in B's definition list, the d_i are in the same order as they appear in B's definition list, and each d_i reaches the exit of B, and either $sl = null$ and B's store list contains no store into V or $sl =< s_1, s_2, \ldots, s_n >$, each s_i appears in B's store list, the s_i are in the same order as they appear in B's store list, and each s_i reaches the exit of B.

Proof: The Lemma follows from the assumption that the definition and store lists for each block are correct, inspection of **Initialize**^{T}, and the definition of ℓ.

□

Definition C.6: A *ds-list-pair for a point* P is a pair (dl, sl) of sequences such that

- $dl =< d_1, d_2, \ldots, d_m >$, where the d_i are distinct definitions of V that reach P (no two are in the same equivalence class),
- $sl =< s_1, s_2, \ldots, s_n >$, where the s_i are distinct stores into V that reach P,
- d_1 and s_1 are opaque,
- all other d_i and s_i are transparent, and
- there is a path, p, to P in the DS-graph such that for all $i < j$, d_j occurs after all instances of d_i on p (after all definitions from the equivalence class of d_i), and s_j occurs after all instances of s_i on p.

The final constraint on a ds-list-pair says that if multiple instances of the same definition reach P along p, the last of them is in the ds-list-pair for B. To illustrate this point, suppose definitions

$< d_1, d_2, \ldots, d_f \ldots, d_g, \ldots, d_h, \ldots, d_m >$ reach P along p, where d_f and d_h are the same definition and all other d_i are distinct.

$< d_1, d_2, \ldots, d_f \ldots, d_g, \ldots, d_h, \ldots, d_m >$ can not be a component of a ds-list-pair for P because d_f and d_h are not distinct.

$< d_1, d_2, \ldots, d_f \ldots, d_g, \ldots, d_{h-1}, d_{h+1}, \ldots, d_m >$ can not be a component of a ds-list-pair for P because d_h occurs after d_g on p.

$< d_1, d_2, \ldots, d_{f-1}, d_{f+1}, \ldots, d_g, \ldots, d_h, \ldots, d_m >$ can not be a component of a ds-list-pair for P.

Theorem C.7: **Algorithm PRS**^{T} terminates with $e \in \text{In}_B^V$ iff e is a ds-list-pair for B.

Proof: **Termination**

Initialize^T terminates because there are a finite number of variables and definitions.

Iterate terminates because

- no element is ever removed from an In or Out set,
- an element is added to at least one In or Out set or an element already present in an In or Out set grows on every iteration but the last,
- elements have a finite size (by the definition of $\overset{\kappa}{\cup}$, an instance of a definition or store appears at most once in a ds-list-pair, and there are a finite number of instances of definitions and stores), and

- elements are from lists-of-definitions $\times$ lists-of-stores, which is a finite set.

If:

If e is a ds-list-pair for the entry of B then e is in In_B^V.

That e is a ds-list-pair for the entry of B guarantees that there is a path p to B along which the definitions and stores in e occur.

The proof is by induction on the length of p.

Base Case: $|p| = 1$. Then B's predecessor P is the start node. **Initialize**^{T} will place e into Gen_P^V (where $e = (<\textit{d-init}>, <\textit{s-init}>)$). In the first iteration, e is placed into Out_P^V by line 7 of **Iterate**. In the second and all subsequent iterations, e is placed into In_B^V by line 5 of **Iterate**.

Inductive hypothesis: Assume that if there is a path p to B of length i, $1 \leq i < n$, such that e is a ds-list-pair for the entry of B along p, then e is in In_B^V.

Inductive step: Show that if there is a path p to B of length n such that e is a ds-list-pair for the entry of B along p, then e is in In_B^V.

Let P be the predecessor of B on p. Because e reaches the entry of B along p, e reaches the exit of P along p. We have a case analysis of the different ways that e can reach the exit of P along p.

Case 1: e consists entirely of definitions and stores in P. By Lemma C.5 and Definition 19.6, Out_P^V is set to $\{e\}$ by line 7 of **Iterate** in the first iteration. In the second and all subsequent iterations $\{e\}$ is placed into In_B^V by line 5 of **Iterate**.

Case 2: No definitions or stores in e are in P. P cannot contain any definitions of V or stores into V, or they would either kill or be appended to the sequences in e. By Lemma C.5, $\mathrm{Gen}_P^V = (\mathit{null}, \mathit{null})$. Then e is a ds-list-pair for P, so by the inductive hypothesis, $e \in \mathrm{In}_P^V$. Assume e was first placed into In_P^V in iteration i. $\mathit{Last}(e\,\kappa\,(\mathit{null}, \mathit{null})) = e$, so line 7 of **Iterate** places e into Out_P^V in iteration i and all subsequent iterations. In iteration $i + 1$ and all subsequent iterations, line 5 of **Iterate** places e into In_B^V.

Case 3: Some, but not all, definitions or stores in e are in P. Let these be e_P. By Lemma C.5 and Definition 19.6, $\mathit{Last}(\mathrm{Gen}_P^V) = e_P$. Then the rest of the definitions and stores in e reach the entry of P. By the inductive hypothesis, there is some ds-list-pair $e' \in \mathrm{In}_P^V$ that includes these. Any other element in e' is in the same equivalence class as some element in e_P, else it would be in e.

Assume e' was first placed into In_P^V in iteration i. $\mathit{Last}(e'\,\kappa\,e_P) = e$, so line 7 of **Iterate** places e into Out_P^V in iteration i and all subsequent iterations. In iteration $i + 1$ and all subsequent iterations, line 5 of **Iterate** places e into In_B^V.

Only If:

If e is in In_B^V, then e is a ds-list-pair for the entry of B.

By the definition of κ, all elements in In_B^V are pairs of sequences of definitions of V and stores into V that have an opaque first element in each sequence and transparent subsequent elements.

I will show the existence of the required path using induction on the number of iterations before e is placed in In_B^V the first time; the distinctness of definitions and stores in e and the ordering constraints on them are by the definitions of κ and *Last*.

Base Case: e is placed into In_B^V for the first time on the second iteration.

By line 2 of **Iterate**, In_B^V and Out_B^V are empty before the first iteration. By line 5 of **Iterate** and because Out_B^V is empty the first time through, In_B^V is empty immediately after the first iteration. Since e is placed into In_B^V in the second iteration, e is placed into Out_P^V in the first iteration, for some predecessor P of B, (line 5 of **Iterate**). A path p' to P exists because otherwise the edge from P to B would have been deleted (DS-graph transformation 3). Let p be $p'| < B >$. By Lemma C.5, *Complete*(Gen_P^V) reaches the exit of P.

Because In_P^V is empty in the first iteration, line 7 of **Iterate** reduces to $\text{Out}_P^V =$ *Last(Complete*(Gen_P^V)), so $e =$ *Last(Complete*(Gen_P^V)) and the d_i and s_i in e occur along p. The definitions of κ and *Last* guarantee the distinctness and ordering of definitions and stores in e.

Inductive hypothesis: Assume that if e is placed into In_B^V for the first time on the i^{th} iteration for $2 \leq i < n$, then e is a ds-list-pair for the entry of B.

Inductive step: Show that if e is placed into In_B^V for the first time on the n^{th} iteration, then e is a ds-list-pair for the entry of B.

If e is placed into In_B^V for the first time on the n^{th} iteration, then by line 5 of **Iterate**, $e \in \text{Out}_P^V$ for some predecessor P of B for the first time in iteration $n - 1$.

The presence of e in Out_P^V is due to the execution of line 7 of **Iterate**. If In_P^V were empty in iteration $n - 1$, then as in the base case, $e =$ *Last*(*Complete*(Gen_P^V)). But then e would have been placed into In_B^V for the first time on the 2^{nd} iteration, contrary to our assumption, so $e \neq$ *Last*(*Complete*(Gen_P^V)) and In_P^V is not empty in iteration $n - 1$.

It follows, by the definition of $\overset{\kappa}{\cup}$, that there is some $(dl, sl) \in \text{In}_P^V$ in iteration $n - 1$ such that $e =$ *Last*($(dl, sl)\, \kappa\, \text{Gen}_P^V$). By the inductive hypothesis, (dl, sl) is a ds-list-pair for the entry of P. Let p' be the path to P guaranteed by the inductive hypothesis. Let p be path $p'| < B >$ to B. Lemma C.5 guarantees the correctness of Gen_P^V. The definitions of κ and *Last* guarantee the distinctness and ordering of definitions and stores in e.

□

Lemma C.8: Initialize-BK^T terminates with $\text{Gen}_{Bk}^V = (dl, sl)$ iff either $dl =$ *null* and B's definition list contains no definitions of V prior to Bk or $dl =< d_1, d_2, \ldots, d_m >$, each d_i appears in B's definition list prior to Bk, the d_i are in the same order as they appear in B's definition list, and each d_i reaches Bk, and either $sl =$ *null* and B's store list contains no store into V prior to Bk or $sl =< s_1, s_2, \ldots, s_n >$, each s_i appears in B's store list prior to Bk, the s_i are in the same order as they appear in B's store list, and each s_i reaches Bk.

Proof: The Lemma follows from inspection of Initialize-BK^T, coupled with the assumption that the definition and store lists for each block are correct.

□

Theorem C.9: Algorithm PRS-BK^T terminates with $e \in \text{PRS}_{Bk}^V$ iff e is a ds-list-pair for Bk.

Proof:

Initialize-BK^T terminates because there is a finite number of definitions and stores in a node.

Algorithm PRS-BK^T terminates because In_B^V is a finite set.

If:

If e is a ds-list-pair for Bk then e is in PRS_{Bk}^V.

Let B be the node containing Bk.

Case 1: e consists entirely of definitions and stores prior to Bk in B. By Lemma C.8 and Definition 19.6, PRS^V_{BK} is set to $\{e\}$ by line 1 of Algorithm PRS-BK^T.

Case 2: No definitions or stores in e are prior to Bk in B. B cannot contain any definitions of V or stores into V, or they would either kill or be appended to the sequences in e. By Lemma C.8, $\text{Gen}^V_{Bk} = (null, null)$. Then e is a ds-list-pair for the entry of B, and by Theorem C.7, $e \in \text{In}^V_P$. e is placed in PRS^V_{BK} by line 1 of Algorithm PRS-BK^T.

Case 3: Some, but not all, definitions or stores in e are prior to Bk in B. Let these be e_{Bk}. By Lemma C.5 and Definition 19.6, $Last(\text{Gen}^V_{Bk}) = e_{Bk}$. Then the rest of the definitions and stores in e reach the entry of B. By Theorem C.7, there is some ds-list-pair $e' \in \text{In}^V_P$ that includes these. Any other element in e' is in the same equivalence class as some element in e_P, else it would be in e. $e = Last(e' \, \kappa \, \text{Gen}^V_{Bk})$, so e is placed in PRS^V_{BK} by line 1 of Algorithm PRS-BK^T.

Only If:

If e is in PRS^V_{Bk}, then e is a ds-list-pair for Bk.

By line 1 of Algorithm PRS-BP^T, $e \in \text{In}^V_B \overset{\kappa}{\cup} \text{Gen}^V_{Bk}$. By Theorem C.7, all elements in In^V_B are ds-list-pairs for the entry of B. By Lemma C.8 Gen^V_{Bk} is correct. It follows that all definitions and stores in e occur along the same path. $\overset{\kappa}{\cup}$ maintains the required distinctness and ordering conditions on definitions and stores (by Definition 19.6). Thus all elements of $\text{In}^V_B \overset{\kappa}{\cup} \text{Gen}^V_{Bk}$ are ds-list-pairs for Bk.

□

Definition C.10: A *p-list-pair for a point P* is a pair (dl, sl) of sequences such that

- $dl =< d_1, d_2, \ldots, d_m >$, where the d_i are distinct definitions of V that reach P,
- $sl =< s_1, s_2, \ldots, s_n >$, where the s_i are distinct stores into V that reach P,
- d_1 and s_1 are opaque,
- all other d_i and s_i are transparent, and
- there is a path-pair, $< p_u, p_o >$, to P such that for all $i < j$, d_j occurs after all instances of d_i on p_u, and s_j occurs after all instances of s_i on p_o.

Theorem C.11: Algorithm PRS-BP^T terminates with $e \in \text{PRS}^V_{Bk}$ iff e is a p-list-pair for Bk.

Proof: Theorem C.11 follows from Theorems C.7, C.9, B.2, and the assumption that the definition and store lists are correct.

□

When some instance of a definition d is not an alias for V, other instances of d outside of any loop or in the same iteration of a loop are not aliases for V, else the compiler could not have put them in the same equivalence class. However, the same instance of d in a different iteration of a loop may alias V, because code in the loop can change what variable d aliases (suppose the source for the definition is `A[i] = x`).

When ds-list-pairs are constructed along a path p, all but the last definition from each equivalence class that occurs along p are eliminated—similarly for stores. Suppose a transparent definition that occurs within a loop aliases V on one iteration and another variable on a later iteration, and V is endangered by virtue of the earlier definition. We must ensure that we do not lose this information when we eliminate the earlier definition from the ds-list-pair. We do not, as we can find a shorter path along which that instance of that definition occurs at most once and along which V is endangered. There is once circumstance in which we would need two instances of a definition (or store) to ensure that we have correct information, and I will argue that this circumstance is semantically pathological and unlikely. In constructing a shorter path, we may eliminate loop iterations (termed *short-circuiting*), add iterations, or re-order iterations.

The following Lemma assumes that there is at most one instance of any definition of V or store into V on any path through a loop. If there is more than one such instance on some path through a loop, either all of them are aliases for V or none of them are aliases for V, so we need only record one of them in a ds-list-pair.

Notational note: d_i is the i^{th} definition of V that occurs along p_u and s_i is the i^{th} store into V that occurs along p_o. Again, I use the naming convention that a definition dx generates the store sx.

Lemma C.12: If there is a path-pair p to a point Bk such that s_i qualified-reaches Bk with d_j along p where d_j is the k^{th} occurrence of some definition dx in a loop L, and s_i is an instance of some store sy where sy was not generated from dx, then there is a path-pair $p' =< p'_u, p'_o >$ to Bk such that there are at most k occurrences of dx in L on p_u and either

- some s_l qualified-reaches Bk with d_m along p', or
- some d_m qualified-reaches Bk with s_l along p',

where s_l was not generated from d_m.

Proof: Let p' be derived by short-circuiting to Bk directly after the loop iteration containing d_j. If s_i qualified-reaches Bk with d_j along p', the Lemma is satisfied. If not, there is some s_q that qualified-reaches Bk with d_j along p'. If s_q was not generated from d_j, the Lemma is satisfied. Assume s_q was generated from d_j. Let d_p be the last definition of V prior to entering L on p (and thus on all constructed paths). Let s_p be the last store into V prior to entering L on p (and thus on all constructed paths).

Case 1—$q > i$: Then s_q was turned off on p by an instance of dx in a later iteration of L. Note that the left-hand-side of dx and sx cannot be loop invariant, because then program semantics would require that if s_q was turned off, all instances of dx and sx would be turned off, including d_j—but d_j reaches Bk by assumption. Therefore there is some path $p^L_{\neg sx}$ through L that contains an instance of dx but does not contain an instance of sx. This implies that V is dead on $p^L_{\neg sx}$ and at Bk: because the left-hand-side of sx is not loop-invariant, sx cannot be eliminated or moved out of the loop without possibly eliminating an assignment into V. However, V may be live on some other path through L (V must be defined before its use on that path). Let p'' be the path resulting from replacing all iterations of L on p with a single iteration of $p^L_{\neg sx}$.

Case 1.a $p^L_{\neg sx}$ contains a store sz.

Case 1.a.1 $p^L_{\neg sx}$ does not dz following dx. Then p'' satisfies the Lemma with s_l being s_z and d_m being the instance of dx in $p^L_{\neg sx}$.

Case 1.a.2 dz follows dx on $p^L_{\neg sx}$. The Lemma does not hold for this case for $j = 1$. Assume $j > 1$. Let q be the path resulting from adding a second iteration of $p^L_{\neg sx}$. Then q satisfies the Lemma with d_m being the instance of dx in the second iteration of $p^L_{\neg sx}$ and s_l being the instance of sz in the first iteration of $p^L_{\neg sx}$. If $j = 1$, the Lemma is not satisfied because although s_l is not generated from d_m, two instances of dx are in L on q. This case can be broken down into a good many subcases, a number of which satisfy the Lemma for $j = 1$. Some of the subcases cannot be satisfied for $j = 1$. Any reader preparing to read further deserves to be spared further subcases given the many cases that follow.

Case 1.b $p^L_{\neg sx}$ does not contain a store.

Case 1.b.1 s_p is not an instance of sx. Then p'' satisfies the Lemma with $l = p$ and d_m being the instance of dx in $p^L_{\neg sx}$.

Case 1.b.2 s_p is an instance of sx.

Case 1.b.2.a There is a path p^L_{sz} through L containing an instance of sz, $sz \neq sx$, but not containing dx. Let p''' be the path resulting from inserting p^L_{sz} into p'' prior to $p^L_{\neg sx}$.

Case 1.b.2.a.1 p^L_{sz} does not contain sx following sz.

Case 1.b.2.a.1.a $p^L_{\neg sx}$ contains an instance of dz that turns off sz. Reverting to p'', the Lemma is satisfied with $l = p$ and with d_m being dz.

Case 1.b.2.a.1.b $p^L_{\neg sx}$ does not contain a definition dz that turns off sz. Then p''' satisfies the Lemma with s_l being sz and d_m being the instance of dx in $p^L_{\neg sx}$.

Case 1.b.2.a.2 p^L_{sz} contains sx following sz.

Case 1.b.2.a.2.a An instance of dz follows dx on $p^L_{\neg sx}$. Then p''' satisfies the Lemma with d_m being the instance of dz on p^L_{negsx} and s_l being the instance of sx on p^L_{sz}

Case 1.b.2.a.2.b d_p is an instance of dx and no instance of dz follows dx on $p^L_{\neg sx}$.

%item **Case 1.b.2.a.2.b.1** Neither p^L_{sz} nor $p^L_{\neg sx}$ contains an instance of dz. Then p''' satisfies the Lemma with $m = p$ and s_l being sz.

Case 1.b.2.a.2.b.2 p^L_{sz} contains an instance of dz. Then let q be the path resulting from removing $p^L_{\neg sx}$ from p'''. q satisfies the Lemma with d_m being the dz, and s_l being the sx, from p^L_{sz}.

Case 1.b.2.a.2.b.3 p^L_{sz} does not contain an instance of dz and an instance of dz precedes dx on $p^L_{\neg sx}$. The Lemma does not hold for this case for $j = 1$. Assume $j > 1$. Let q be the path resulting from replacing all iterations of L on p with two iterations of $p^L_{\neg sx}$. q satisfies the Lemma with d_m being the instance of dx from the second iteration of $p^L_{\neg sx}$ and s_l being the last store prior to s_p that is not an instance of sx. (Any instances of sx immediately preceding s_p must be turned off if s_p is turned off, since outside of a loop, two instances of a store must target the same location.) If $j = 1$, the Lemma is not satisfied because although s_l is not generated from d_m, two instances of dx are in L on q. There are subcases for which the Lemma is satisfied, but as the Lemma cannot be satisfied for all of them, any reader that has gotten this far is to be congratulated and spared further subcases.

Case 1.b.2.a.2.c d_p is not an instance of dx. Then let q be the path resulting from removing $p^L_{\neg sx}$ from p'''. q satisfies the Lemma with $m = p$ and s_l being sx (from p^L_{sz}).

Case 1.b.2.b Every path p^L_{sz} through L containing an instance of sz, $sz \neq sx$, contains dx. The Lemma does not hold for this case for $j = 1$. Assume $j > 1$. Let q be the path resulting from replacing all iterations of L on p with a single iteration of such a p^L_{sz} followed by a single iteration of $p^L_{\neg sx}$.

Case 1.b.2.b.1 Some p^L_{sz} does not contain sx following sz. q satisfies the Lemma with d_m being the instance of dx from $p^L_{\neg sx}$ and s_l being the instance of sz from p^L_{sz}.

Case 1.b.2.b.2 Every p^L_{sz} contains sx following sz. q satisfies the Lemma with d_m being the instance of dx from p^L_{sz} and s_l being the instance of sz from p^L_{sz}, because the instance of dx from $p^L_{\neg sx}$ turns off the instance of sx from from p^L_{sz}.

In both subcases, if $j = 1$, the Lemma is not satisfied because although s_l is not generated from d_m, two instances of dx are in L on q. This case can be broken down into a good many subcases, a number of which satisfy the Lemma for $j = 1$, based on the existence and placement of dz and whether d_p is an instance of dx. Some of the subcases cannot be satisfied for $j = 1$. Any reader that has gotten this far is to be congratulated and spared further subcases.

Case 1.b.2.c There is no path p^L_{sz} through L containing sz, $sz \neq sx$. Then s_i must be before L.

Case 1.b.2.c.1 There is a path p^L_{dz} through L containing dz, $dz \neq dx$. Let q be the path resulting from replacing all iterations of L on p with a single iteration of p^L_{dz}. q satisfies the Lemma with $s_l = s_p$ and d_m being dz.

Case 1.b.2.c.2 There is no path p^L_{dz} through L containing dz, $dz \neq dx$. The Lemma does not hold for this case for $j = 1$. Suppose $j > 1$. Let q be the path resulting from replacing all iterations of L on p with two iterations of $p^L_{\neg sx}$. q satisfies the Lemma with d_m being the instance of dx from the second iteration of $p^L_{\neg sx}$ and s_l being the last store prior to s_p that is not an instance of sx. (Any instances of sx immediately preceding s_p must be turned off if s_p is turned off, since outside of a loop, two instances of a store must target the same location.) If $j = 1$, the Lemma is not satisfied because although s_l is not generated from d_m, two instances of dx are in L on q.

Case 2—$q < i$: Then s_i was in a later iteration of L and killed s_q. Let the subpath of p taken through L on the iteration containing d_j be p^L_j. There is some subpath p^L_y of p taken through L that contains sy but does not contain dy, or s_i would not qualified reach with d_j along p.

Case 2.a: p^L_y does not contain an instance of dx. Let p'' be the path resulting from inserting p^L_y into p' immediately following p^L_j. p'' satisfies the Lemma with s_l being the instance of sy on p^L_y and $m = j$.

Case 2.b: p^L_y contains an instance of dx.

Case 2.b.1: sx does not follow sy on p^L_y. Let p'' be the path resulting from replacing p^L_j with p^L_y on p'. Again, p'' satisfies the Lemma with s_l being the instance of sy on p^L_y and $m = j$.

Case 2.b.2: sx follows sy on p_y^L. Let p' be constructed from p by replacing all iterations of L on p with p_y^L.

Case 2.b.2.a: d_p is not an instance of dy. p' satisfies the Lemma with s_l being the instance of sy on p_y^L and $m = p$.

Case 2.b.2.a: d_p is an instance of dy. (While in this case, a store (sy) has been moved into a loop, in the corresponding case in the corollary, the store has been moved out of the loop.) The Lemma does not hold for this case for $j = 1$. Suppose $j > 1$. Let q be the path resulting from replacing all iterations of L on p with two iterations of p_y^L. q satisfies the Lemma with d_m being the instance of dx from the first iteration of p_y^L and s_l being the instance of sy from the second iteration of p_y^L.

If $j = 1$, the Lemma is not satisfied because although s_l is not generated from d_m, two instances of dx are in L on q.

□

Corollary C.13: If there is a path-pair p to a point Bk such that d_i qualified-reaches Bk with s_j along p where s_j is the k^{th} occurrence of some store sx into V in a loop L, $k >= 2$, and d_i is an instance of some definition dy of V where dy was not generated from sx, then there is a path-pair $p' =< p'_u, p'_o >$ to Bk such that there are at most k occurrences of sx in L on p_u and either

- some d_l qualified-reaches Bk with s_m along p', or
- some s_m qualified-reaches Bk with d_l along p',

where s_m was not generated from d_l.

Proof: The proof follows the same arguments as the previous proof, with the roles of definitions and stores reversed.

□

Notational note: In the following theorems, d_x^i is the i^{th} instance of the definition at position x in some specified p-list-pair, so d_x^i and d_x^j are instances of the same definition. A p-list-pair $(< d_1, d_2, \ldots, d_n >, < s_1, s_2, \ldots, s_m >)$ is said to *mismatch* if for $i \leq max(n, m)$, s_i was not generated from d_i. If a p-list-pair does not mismatch, then $n = m$ and it does not matter which end we start from in an attempt to match the d_i and s_i. We are interested in the first mismatch encountered when the matching on the p-list-pair is done from back to front: a p-list-pair $(< d_1, d_2, \ldots, d_n >, < s_1, s_2, \ldots, s_m >)$ *mismatches at* d_i, s_j if $n - i = m - j$ and i and j are the largest such indices such that s_j was not generated from d_i. Given a path-pair p, *Defs(p)* is the sequence of definitions of V that occur along p_u, and *Stores(p)* is the sequence of stores into V that occur along p_o,

Theorem C.14: If a p-list-pair for a point Bk mismatches then there is a path-pair p to Bk such that

s_y^i qualified-reaches Bk with d_x^j, or

d_x^j qualified-reaches Bk with s_y^i

along p, and s_y^i is not generated from d_x^j.

Proof: Let $(< d_1, d_2, \ldots, d_n >, < s_1, s_2, \ldots, s_m >)$ be a p-list-pair for Bk that mismatches at d_a, s_b. Let d_a be an instance of da and s_b be an instance of sb. From Definition C.10, there is a path $p =< p_u, p_o >$ such that

Defs$(p_u) = \alpha| < d_a^i > |\beta$, where β contains no instances of da, and

Stores$(p_u) = \gamma| < s_b^j > |\delta$, where δ contains only instances of sb.

Because d_a, s_b is the first mismatch (starting from the end), each store in δ is generated from some definition in β and each definition in β generates some store in δ. There may be multiple instances of some of these definitions and stores in *beta* and *delta*. If these are not in the right order or numeric correspondence, some d_q, s_q in β and δ satisfies the Theorem, or one of d_a^i or s_b^j satisfies the Theorem with some s_q in δ or d_q in β. Otherwise, d_a^i, s_b^j satisfies the Theorem. .

$\square$

Theorem C.15: If there is a path-pair p to Bk such that

s_y^i qualified-reaches Bk with d_x^j, or

d_x^j qualified-reaches Bk with s_y^i

along p, and s_y^i is not generated from d_x^j, then there is a p-list-pair for Bk that mismatches.

Proof: Let

$Defs(p_u) = \alpha| < d_x^j > |\beta$, and

$Stores(p_u) = \gamma| < s_y^i > |\delta$.

Assume s_y^i qualified-reaches Bk with d_x^j (a symmetric argument applies if instead d_x^j qualified-reaches Bk with s_y^i). Note that s_x does not qualified-reach Bk with d_x^j, where s_x is generated from d_x^j. If there is some d_x^k in β, we can apply Lemma C.12 or Corollary C.13 to find a shorter path to B that has the conditions required by the Theorem, except in the cases where we have noted that Lemma C.12 and Corollary C.13 do not hold. This Theorem therefore does not hold in those cases either. (Note that if the left-hand-side of d_x^j is loop invariant, there is no d_x^k in β, because if there were, s_y^i could not qualified reach bk with d_x^j.)

We can therefore assume there are no d_x^k in β. Then there are no s_x^k in δ else(s_y^i would not qualified-reach Bk with d_x^j) and there is a p-list-pair $pl = (< \ldots, d_x > |A, < \ldots, s_y > |B)$ where every store in B is generated from some definition in A but s_y is not generated from a definition in A. Therefore PL either mismatches in A, B (due to the ordering of the definitions and stores), mismatches at d_A, s_y where d_A is in A, mismatches at d_x, s_B where s_B is in B, or mismatches at d_x, s_y.

$\square$

I can now prove Theorem 19.8, restated here for convenience.

Theorem 19.8:

$\text{PRS}_B^V = \emptyset$ iff either V is not in scope at B or B is unreachable. Otherwise:

V is current at Bk iff V is current at Bk by e, $\forall e \in \text{PRS}_{Bk}^V$;

V is endangered at Bk iff $\exists e \in \text{PRS}_{Bk}^V$ such that V is not current at Bk by e;

V is noncurrent at Bk iff $\nexists e \in \text{PRS}_{Bk}^V$ such that V is current at Bk by e.

Proof: Assume $\text{PRS}_B^V \neq \emptyset$, V is in scope, and B is reachable.

By Theorem C.11 and Definition C.10, PRS_{Bk}^V contains p-list-pairs for all paths to Bk. Theorem 19.8 follows from that fact, Theorems C.14 and C.15, and Definitions 16.11, 16.12, and 16.13. The Theorem does not hold for the cases for which Theorem C.15 does not hold.

Assume $\text{PRS}_B^V = \emptyset$. Then by line 1 of Algorithm PRS-BK^T, In_B^V is empty and $Complete(\text{Gen}_{Bk}^V) = \emptyset$. If V is in scope at B, we assume an initial definition of V and an initial store into V, ensuring that at some definition of V and store into V reach B along every path. But then there would be some ds-pair in In_B^V. Since there is not, either there are no definitions of V or stores into V in the flow graph component, implying that V is not in scope, or there are no paths to B.

Assume B is unreachable. Then In_B^V is empty (by lines 2 and 4 of **Iterate**. We assume that there are no stores in an unreachable block, therefore $\mathit{Complete}(\mathrm{Gen}_{Bk}^V) = \emptyset$, so by line 1 of **Algorithm PRS-BK**^{T}, $\mathrm{PRS}_B^V = \emptyset$.

Assume V is not in scope. Then there are no definitions of V or stores into V in any block in the flow graph component, implying that In_B^V is empty and $\mathrm{Gen}_{Bk}^V = (\mathit{null}, \mathit{null})$, so by line 1 of **Algorithm PRS-BK**^{T}, $\mathrm{PRS}_B^V = \emptyset$.

□

References

[AG93a] A. Adl-Tabatabai, T. Gross, "Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging," *Proceedings of the POPL'93, The Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[AG93b] A. Adl-Tabatabai, T. Gross, "Detection and Recovery of Endangered Variables Caused by Instruction Scheduling," To appear in the *Proceedings of the PLDI'93, ACM SIGPLAN/93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

[AG92] A. Adl-Tabatabai, T. Gross, "The Effects of Register Allocation and Instruction Scheduling of Symbolic Debugging," *Proceedings of the Supercomputer Debugging Workshop* , Dallas, Texas, October 1992.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques, and Tools," Addison-Wesley, Menlo Park, CA, 1986.

[AU77] A. V. Aho, J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, Menlo Park, CA, 1977.

[Bal69] R. M. Balzer, "EXDAMS - EXtendable Debugging and Monitoring System," *Proceedings of AFIPS Spring Joint Computer Conference*, Vol 34 pp. 125-134, 1969.

[BHS92] G. Brooks, G. J. Hansen, and S. Simmons, "A New Approach to Debugging Optimized Code," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 27, No. 7, pp. 1-11, San Francisco, California, June 1992.

[BK92] J. Brown, R. Klamann, "The Application of Code Instrumentation Technology in the Los Alamos Debugger", Proceedings of the Supercomputer Debugging Workshop '92, Dallas, Texas, October 1992. October 1992.

[Bro91] J. S. Brown, "The Los Alamos Debugger ldb", Proceedings of the Supercomputer Debugging Workshop '91, Albuquerque, New Mexico, November 1991.

[BW92] T. Bemmerl, R. Wismüller, "Quellcode-Debugging von global optimierten Programmen", GI-ITG Workshop "Parallelrechner und Programmiersprachen", Schloss Dagstuhl, Germany, Feb. 26-28, 1992 (in German)

[BW93] L. Berger, R. Wismüller, "Source-Level Debugging of Optimized Programs Using Data Flow Analysis", unpublished draft from the Department of Computer Science, Munich Institute of Technology, Germany, 1993.

[CH91] B. Chase, R. Hood, "Debugging with Lightweight Instrumentation", Proceedings of the Supercomputer Debugging Workshop '91, Albuquerque, New Mexico, November 1991.

[Coh91] R. Cohn, "Source Level Debugging of Automatically Parallelized Code," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, *SIGPLAN Notices*, Vol. 26, No. 12, pp. 132-143 December 1991.

[CM93] M. Copperman, C. E. McDowell, "A Further Note on Hennessy's "Symbolic Debugging of Optimized Code", *ACM Transactions on Programming Languages and Systems* Vol. 15, No. 2, pp. 357-365, April 1993.

[CM91b] M. Copperman, C. E. McDowell, "Debugging Optimized Code Without Surprises," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, New Mexico, November 1991.

[Cop92] M. Copperman, "Debugging Optimized Code: Currency Determination with DataFlow," *Proceedings of the Supercomputer Debugging Workshop* , Dallas, Texas, October 1992.

[Cop92] M. Copperman, "Debugging Optimized Code Without Being Misled," UCSC Technical Report UCSC-CRL-92-01, January 1992. Submitted for publication to *ACM Transactions on Programming Languages and Systems.*

[Cop90] M. Copperman, "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values," University of California, Santa Cruz technical report UCSC-CRL-90-23, May 1990.

[CT93] M. Copperman, J. Thomas "Poor Man's Watchpoints," University of California, Santa Cruz technical report UCSC-CRL-93-12, March 1993.

[Cor91] Steve Correll, personal communication, Borland International, Scotts Valley, CA, April 1991

[CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[FM80] P. H. Feiler, R. Medina-Mora, "An Incremental Programming Environment," Carnegie Mellon University Computer Science Department Report, April 1980.

[Fri83] P. Fritzon, "A Systematic Approach to Advanced Debugging through Incremental Compilation", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, California, March 1983, also published as *SIGPLAN Notices*, Vol. 18, No. 8, pp. 130-139 Aug. 1983.

[Gup90] R. Gupta, "Debugging Code Reorganized by a Trace Scheduling Compiler," Structured Programming, Vol. 11, No. 3, pp.1-10, July 1990.

[Hen82] J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982.

[Hen90] Hennessy, J., Center for Integrated Systems, Stanford University, Stanford, CA, personal communication regarding "Symbolic Debugging of Optimized Code," 1991.

[Mel90] Meloy, S., Hewlett-Packard, 3345 Mount Pleasant Rd., Lincoln, CA, personal communication regarding DOC, 1990.

[Kes90] P. Kessler, "Fast Breakpoints: Design and Implementation", *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990.

[Kor88] B. Korel, "PELAS Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260, September 1988.

[MC88] B. Miller, J. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 125-134, Madison, Wisconsin, 1988.

[MC91] B. Miller, J. Choi, "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 491-530, 1991.

[Pic90] D. Pickens, MetaWare Incorporated, Santa Cruz, CA, personal communication regarding the MetaWare High C compiler.

[PS91] P. P. Pineo, M. L. Soffa, "Debugging Parallelized Code Using Code Liberation Techniques," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, *SIGPLAN Notices*, Vol. 26, No. 12, pp. 108-119 December 1991.

[PS88] L. L. Pollock, M. L. Soffa, "High Level Debugging with the Aid of an Incremental Optimizer," *Hawaii International Conference on System Sciences*, January 1988.

[PS92] L. L. Pollock, M. L. Soffa, "Incremental Global Reoptimization of Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 2, pp. 173-200, 1992.

[ST83] R. Seidner, N. Tindall, "Interactive Debug Requirements", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, California, March 1983, also published as *SIGPLAN Notices*, Vol. 18, No. 8, pp. 130-139 Aug. 1983.

[Shu89] W. S. Shu, "A Unified Approach to the Debugging of Optimized Programs", Ph.D. Disssertation, Department of Computer Science, University of Nottingham, England, UK, 1989.

[Shu91] W. S. Shu, "A Formal Characterisation of the Effects of Optimization on Debugging", Technical Report TR003-WSS-91, Department of Computer Science, University of Yaoundé, Cameroon, 1991.

[Shu92] W. S. Shu, "A New Basis for Debugging ...from Optimized Programs", First International Conference on Research in Computer Science, Yaoundé, Cameroon (published by INRIA, France) 1992.

[ST83] W. S. Shu, "Adapting A Debugger for Optimized Programs", *SIGPLAN Notices*, Vol. 28, No. 4, pp. 39-44 April 1993.

[Sil92] J. Silverstein, ed., "DWARF Debugging Information Format," Proposed Standard, UNIX International Programming Languages Special Interest Group, April 1992, personal communication regarding DWARF.

[Str91] L. Streepy, "CXdb A New View On Optimization," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[WST85] D. Wall, A. Srivastava, R. Templin, "A note on Hennessy's *Symbolic Debugging of Optimized Code*," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 176-181, Jan. 1985.

[Wan91] Wang, F., Microtec Research, Inc., Santa Clara, CA, January 1992, personal communication regarding Microtec Research, Inc.'s Xray Debugger.

[Wis93] Wismueller, R., Institut fur Informatik, Technische Universitat Munchen, Munich, Germany, personal communication regarding current research, February 1993.

[WS78] H. S. Warren, Jr., H. P. Schlaeppi, "Design of the FDS interactive debugging system," IBM Research Report RC7214, IBM Yorktown Heights, July 1978.

[Ze83a] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Research Report CSL-83-1, Xerox Palo Alto Research Center, Palo Alto, CA, Jan. 1983.

[Ze83b] P. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs," *SIGPLAN Notices*, Vol. 18, No. 8, pp. 159-172 Aug. 1983.

[Zel84] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Research Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

[ZJ90] L. W. Zurawski, R. E. Johnson, "Debugging Optimized Code With Expected Behavior," Unpublished draft from University of Illinois at Urbana-Champaign Department of Computer Science, August 1990.