Debugging Optimized Code With Expected Behavior

Lawrence W. Zurawski
Ralph E. Johnson
University of Illinois at Urbana-Champaign

April 14, 1991

Thank-you for reading this paper. I am interested in any and all comments you might have about it. In particular, I am interested in what you find that is

1. *not clear*
2. *too repetitive*
3. *too formal*

The paper has been submitted for publication, so you should give comments by first quarter 91 to ensure that they get into the final version. Please send comments to Ralph Johnson, Department of Computer Science, 1304 W. Springfield Ave, Urbana IL 61801. *Phone:* 217-244-0093. *E-mail:* johnson@cs.uiuc.edu.

Abstract

This paper shows how a compiler and debugger can work together so that the debugger can provide the behavior that the user expects but the compiler can provide highly optimized programs. It introduces the context-oriented model of debugging, which hides the effects of code optimization from the debugger by having the debugger interact with contexts, which are data abstractions representing the run-time state. This reduces the problem of debugging optimized code to that of implementing contexts in the presence of optimizations.

1 Introduction

Optimizing compilers and source-level debuggers have become essential for program development. However, their basic premises differ, so they usually do not work well together. An optimizing compiler makes a program faster by changing the sequence of states that the program passes through to compute a result. A debugger examines intermediate states of a program's execution, so it exposes the effects of the optimizations.

The conflict between debugging and optimization is usually resolved by having debuggers work only on unoptimized programs. Production programs are usually optimized, so they cannot be debugged. However, it is important to be able to debug production programs.

- It is inconvenient to recompile the program with optimizations turned off and run the program again before using the debugger.

- It can be expensive or impossible to recreate the input sequence that led to the error.
- In some languages, it is difficult to determine which procedures should be recompiled, since the notion of a program is not well defined. For example, a Lisp or Smalltalk program is not a single piece of code, but rather a collection of procedures scattered throughout the programming environment, many of which are used by many different applications.

Making the debugger compatible with the optimizer means that the programmer never needs to choose between debugging and execution speed.

This paper describes how to make an optimizing compiler cooperate with a source-level debugger. These techniques will work for any language, though they will work best for languages that are amenable to static analysis, i.e. Ada or Scheme, but not C. The techniques assume an integrated programming environment, where compiler and debugger work closely together and where there is a database with complete information on all parts of the program. The debugger will often need part of the program to be unoptimized, but normal program execution will use only the optimized program and the debugger will transparently switch back and forth between the two. The unoptimized program is only needed if the program state is changed, and only the procedures affected by the change need to be unoptimized.

The type of debugger with which we are concerned has three important characteristics:

- It allows the user to examine and modify the state of an executing program.
- It interacts with the user at the source-language level, hiding the effects of optimization.
- It is always ready; no special actions need be performed by the user, either at compile time or at run time, to enable debugging.

The conflict between debugging and optimization is resolved by sacrificing debugging capabilities, optimization, or some of each. All of the previous research in this area has emphasized optimization over debugging capabilities [Zel84, Hen82, CMR88]. Full optimization is performed, but debugging is restricted. The debugger never gives an incorrect response, but sometimes tells the user that it cannot carry out a debugging request because it is unable to completely unravel the effects of optimization. This is known as *truthful behavior* [Zel84].

We emphasize debugging over optimization; the debugger is never prevented from carrying out a request. This is known as *expected behavior* [Zel84]. The debugger presents the user with a source-level, unoptimized view of the program state. All low-level details are hidden from the user, including the optimizations that were performed and the structure of the low-level program state, i.e. the run-time system. Sometimes a particular optimization, such as eliminating a dead assignment or moving a statement out of a loop, must be prevented or changed to provide expected debugger behavior, but the resulting programs are only a little slower.

One of the main contributions of this paper is the *context-oriented model* of debugging, which is described in Section 2. This model ensures that only a few parts of the debugger depend on the underlying implementation of the run-time system. The context-oriented model isolates the parts of the debugger that are affected by optimization and makes the debugger portable.

Another key idea in this paper is that a debugger can provide expected behavior if it can recover the source-level, unoptimized program state (current execution locations, variable values, and procedure traceback) from the low-level, optimized state whenever it is invoked. The compiler computes the set of *inspection points*, which are the points in the program where the debugger might be invoked. For each inspection point, the compiler computes *recovery information*, which describes the effects of optimization at that inspection point. Recovery information lets the debugger recover the values that the variables would have had in an unoptimized program. The compiler needs dependency information to perform optimizations and compute the recovery information. We use static single assignment (SSA) form [CFR*89] to represent this dependency information.

The debugger uses recovery information to map the low-level, optimized program state to the source-level state at inspection points. Debugger commands that change the program state are implemented by converting the affected part of the low-level state (which is always the state contained in a single activation record) to unoptimized form before carrying out the command.

The techniques described here were used to modify the standard Smalltalk debugger for use with TS, an optimizing compiler for Typed Smalltalk [JGZ88] [Zur89].

1.1 Debugging vs. Optimization

Debugger commands access and change the program state. Typical debugger commands are:

- Report the source statement that corresponds to the current execution point
- Report the value of a source-level variable
- Change the value of a source-level variable
- Report the procedure traceback
- Set and remove breakpoints
- Evaluate an expression
- Change and recompile a part of the program
- Execute the next statement (single-step)
- Resume execution

A debugger that examines the optimized program state will get different results than one that examines the unoptimized state. Allowing access to the program state at intermediate points invalidates assumptions made by the optimizer, such as the assumption that certain variables are not assigned or referenced at these points. The compiler can help resolve this conflict by computing recovery information that the debugger can use to unravel the effects of optimization. This lets the debugger carry out requests even if the optimized and unoptimized program states differ.

Optimizations can remove and reorder statements and alter the program's control flow. Variables may be eliminated, overlaid, or assigned to registers. The debugger difficulties caused by optimization are classified as either *data problems*, which are related to reporting

and changing the values of variables, or *location problems*, which include difficulties with reporting the current execution point, reporting the procedure traceback, setting and fielding breakpoints, and single stepping [Zel84].

The debugger often needs mappings between the source and object code. It uses a mapping from object code locations to source statements to report the current execution location at the source level by mapping the program counter to its source statement. It uses a mapping from source statements to object code locations for single stepping and to set breakpoints. These mappings are trivial for unoptimized code, since each object code instruction is part of exactly one source statement, each source statement starts at a particular object code instruction, and the instructions generated for a source statement are contiguous in the object code. None of the above are necessarily true for optimized code.

There are several ways in which particular optimizations can cause difficulties for debugging. Many of these difficulties are described in [Zel84, Hen82, ST83]. Detailed descriptions of the various optimizations are found in [ASU86, BE76, WJW*75, LM69].

- Optimizations such as common-subexpression elimination, constant propagation, and copy propagation make it difficult to change the values of variables from the debugger. As an example, consider the following statements:

  $$\begin{aligned} 1&: \ i \leftarrow a + b \\ 2&: \ j \leftarrow a + b \end{aligned}$$

  Common-subexpression elimination replaces the expression $a+b$ in the second assignment with i, resulting in the following code:

  $$\begin{aligned} 1'&: \ i \leftarrow a + b \\ 2'&: \ j \leftarrow i \end{aligned}$$

  If the value of a or b is changed by the debugger before statement $2'$, the change will not be reflected in the assignment to j in statement $2'$. Changing the value of i immediately before statement $2'$ will cause an incorrect value to be assigned to j in $2'$.

- Eliminating a dead assignment will cause the destination of the assignment to have an unexpected value between the point of the eliminated assignment and the next assignment to the variable. Eliminating dead assignments is an important optimization, because dead assignments are often created by optimizations such as common-subexpression elimination and induction variable elimination. Code motion can also cause variables to have unexpected values at certain points in the program.

- Optimizations such as register allocation and storage overlaying can result in a variable's value not being available in certain program regions. Also, if a variable resides in different locations at different points during execution then a mechanism is needed to track the variable through the different locations.

- Optimizations such as inline procedure expansion, procedure discovery, cross jumping, tail recursion elimination and code motion alter the correspondence between source locations and executable code locations. As a result, these optimizations can prevent the debugger from correctly reporting the current source-level execution location.

Also, inline procedure expansion and tail recursion elimination remove procedure calls, so reporting the procedure traceback is complicated by these optimizations. Procedure discovery causes similar problems by introducing procedure calls that are not present in the source code.

1.2 Related Work

Zellweger [Zel84] describes many of the problems caused by optimization and shows how to solve some of them. She describes the implementation of an interactive, source-level debugger that works with programs on which inline procedure expansion and cross-jumping have been performed. Her goal is to provide expected debugger behavior whenever possible, and to never allow less than truthful behavior.

Hennessey [Hen82] is concerned with detecting and recovering *noncurrent* variables in the presence of several local and global optimizations. A noncurrent variable is one whose value at a given execution point in the optimized program does not equal its value at the same point in the unoptimized program. He presents algorithms to determine which variables are noncurrent and to determine which of the noncurrent variables are recoverable.

Coutant, Meloy, and Ruscetta [CMR88] describe techniques that can be used for reporting variable values in the presence of several optimizations including register allocation, induction variable elimination, and code reordering. The problem of tracking a variable through the different registers and memory locations that it might occupy during the execution of the program is addressed, but no attempt is made to recover noncurrent variables.

Warren and Schlaeppi [WS78] propose two modes of compiler/debugger operation. In "full optimization" mode, the debugger would explain how optimization had affected the program. In "no source change" mode, optimizations would be restricted to involve only compiler-introduced temporaries, or to occur only within a statement. This project lasted only long enough to produce the design document.

Fieler proposes a design that uses multiple levels of optimization for debugging optimized programs [Fei82]. As optimization increases, debugging capabilities decrease. The compiler would not collect any information regarding the effects of optimization.

Seidner and Tindall describe many of the problems incurred when debugging optimized code, without proposing specific solutions [ST83]. The difficulty of debugging optimized code is also noted in [BE76], [Joh79], [PL83], and [Sat75].

1.3 Contributions

The most important way in which our work differs from that done previously is that we show how to provide expected debugger behavior, while sacrificing some potential optimization. Others have chosen full optimization, while only providing truthful debugger behavior. To allow expected behavior, the compiler performs modified versions of traditional optimizations and computes recovery information. The technique of restricting optimization to provide expected behavior has been suggested previously [Zel84] as being an important open research area, but has never before been investigated thoroughly. Another important contribution is that a wide range of debugger commands and optimizations are considered. Debugger commands that change the program state, such as changing the value of a variable and single stepping, are given much more attention than in the past.[1]

[1] Neither Zellweger nor Hennessey allow debugger commands that change the program state.

Several important techniques are used to provide expected debugger behavior with little loss of optimization:

- The source-level unoptimized program state, the low-level optimized state, and the recovery information are all carefully described as part of a model of debugging. The model minimizes the work required to port a debugger to a new implementation of the run-time system. Also, the techniques provide a framework for including other optimizations. In fact, the recovery information was initially designed to unravel the effects of inline procedure expansion and global dead-store elimination. Little change was required to add other optimizations.

- The compiler identifies inspection points, which are execution points (in the compiled code) at which the debugger might be invoked, and optimizes completely between these points.

- Debugger commands that change part of the program state are implemented by recovering the source-level state and using this information to convert the affected part of the low-level state to unoptimized form before carrying out the command. Zellweger suggests this technique, but does not investigate it thoroughly.

2 The Context-Oriented Model of Debugging

This section answers the following question: "What information about the source-level state does a debugger need to provide all of its commands?" The answer is a debugger-oriented semantics called the context-oriented model of debugging. It is abstract enough not to bias the implementation of a language, but detailed enough to describe the procedure traceback and access to intermediate values of variables.

The context-oriented model of debugging separates most of a debugger from the underlying implementation of the run-time system. This independence is achieved by having a source-level abstraction of the program state that serves as an intermediary between the debugger front-end and the low-level program state. This not only makes the debugger portable, but isolates the parts of the debugger that are affected by optimization, so they can be considered separately.

2.1 Contexts

A source-level debugging session is a sequence of operations that access or modify the source-level program state. The user issues source-level requests to the debugger front-end and the debugger responds to the user in these same terms. The front-end carries out a debugging request by invoking a sequence of operations on the source-level program state.

The source-level state is represented by *contexts*. Contexts are the source-level counterparts of stack frames, i.e. a context represents the source-level state of a procedure invocation. Contexts are data abstractions with operations that the debugger front-end invokes to get information in source-level terms. Operations on contexts access the low-level program state and completely hide the conversion between the low-level and source-level states (including unraveling the effects of optimization).

Encapsulating the conversion between the low-level and source-level states in the context operations not only simplifies the debugger front-end, it also makes the front-end independent of the implementation of the run-time system. The front-end does not depend on

the structure of the run-time system, the form of the executable code, or whether the executable code is optimized. This reduces the problem of debugging optimized code to that of implementing contexts in the presence of optimization. A debugger can be ported to a new implementation of the run-time system by simply reimplementing contexts. Figure 1 illustrates the context-oriented model.

The following domains are used to define the operations on contexts:

C	Contexts
P	Procedures
V	Variables
VAL	Values
$STMT$	Source statements
NIL	A domain containing the single element nil

A list of zero or more elements from a domain D is denoted D^*. A list of one or more elements from D is denoted D^+. Concatenation is defined such that for an element d and a list $\vec{d} = d_1 \ldots d_n$, $d \cdot \vec{d} \equiv dd_1 \ldots d_n$, and $\vec{d} \cdot d \equiv d_1 \ldots d_n d$. Also, for a list $\vec{d} = d_1 \ldots d_n$ and a list $\vec{b} = b_1 \ldots b_m$, $\vec{d} \cdot \vec{b} \equiv d_1 \ldots d_n b_1 \ldots b_m$.[2]

This section defines each context operation as an abstract operation on the source-level program state. It is obvious how these operations are implemented on unoptimized contexts, but not on optimized ones. These operations will be used in Section 2.2 to define each of the debugger commands. The operations will be made concrete in Section 3 when they are described in terms of their effect on the optimized program state. All of the context operations involve either accessing or modifying part of the source-level program state, which is not surprising, since inspecting and modifying the state of an executing program is the essence of debugging.

Every context has the following operations:

- Return the calling context. The stack grows upward; the calling context for the bottommost context in the procedure traceback is nil.

$$CALLER : C \rightarrow (C + NIL)$$

- Return the value of any local variable.

$$VAR_VALUE : (C \times V) \rightarrow VAL$$

- Change the value of any local variable. This operation has four arguments: The context in which the value is to be changed, the variable whose value is to be changed, the value to assign to this variable, and the sequence of contexts that represents the entire procedure traceback. The need for the fourth argument will become clear in Section 3. The value returned by this operation is a pair consisting of the updated context and the updated sequence of contexts representing the entire procedure traceback. The other state-changing context operations also require the entire procedure traceback as an argument and return a pair consisting of the updated context and the updated procedure traceback.

[2] We generally use $\equiv$ to mean "is defined as" and $=$ for all other uses of "equals".

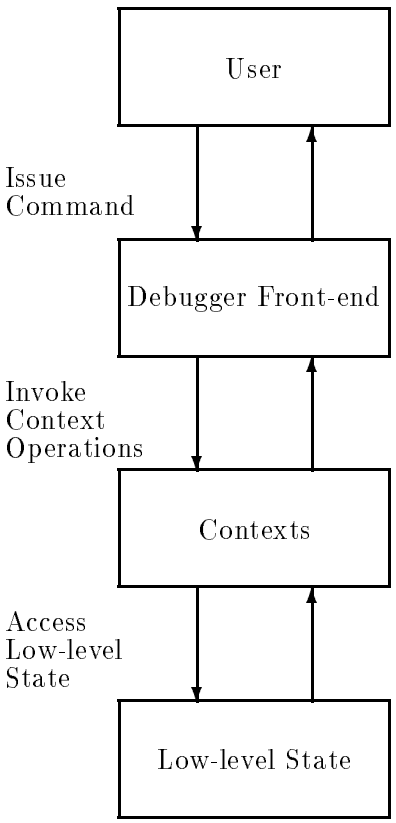

Figure 1: The context-oriented model of debugging

$$ASSIGN_VALUE : (C \times V \times VAL \times C^+) \rightarrow (C \times C^+)$$

- Return the source statement that corresponds to the current execution point.

$$CURRENT_STMT : C \rightarrow STMT$$

- Change the source statement that corresponds to the current execution point.

$$SET_STMT : (C \times STMT \times C^+) \rightarrow (C \times C^+)$$

- Return the procedure that is executing the context.

$$PROCEDURE : C \rightarrow P$$

- Execute the next source statement. Return an updated version of the context that has control after executing this statement, which is different from the original context if the statement is a procedure call or return.

$$EXECUTE_STMT : (C \times C^+) \rightarrow (C \times C^+)$$

- Resume execution from this context. The value returned by this operation is not used, since the debugger relinquishes control when execution is resumed.

$$RESUME : C \rightarrow NIL$$

NEW_CONTEXT(c,p,$\vec{c}$) is the constructor for contexts, where c is the caller and p is the procedure of the new context. The third argument is the sequence of contexts that represents the entire procedure traceback. The current execution point is the first statement in p, and all local variables have an initial value determined by the language in which the program is written.

2.2 Debugger Commands

This section shows how debugger commands can be implemented in terms of contexts. Each debugger has its own set of commands, so it is hard to enumerate all the commands any debugger could perform. The set of debugger commands presented here is that supported by the Smalltalk-80 debugger[Gol84] and is fairly typical. Functions representing debugger commands and auxiliary functions are given lower-case names to distinguish them from context operations, which have upper-case names. The commands are:

Report the Value of a Local Variable
The value of variable v in context c is given by

$$report_value(c, v) \equiv \\ VAR_VALUE(c, v)$$

Change the Value of a Local Variable
The value k is assigned to variable v in context c by:

$$change_value(c, v, k, \vec{c}) \equiv \\ ASSIGN_VALUE(c, v, k, \vec{c})$$

Report the Source Statement that Corresponds to the Current Execution Point

The statement currently executing in context c is

$$\begin{array}{l} report_execution_location(c) \equiv \\ \quad CURRENT_STMT(c) \end{array}$$

Proceed

Proceed is used to resume execution when a debugging session is complete. This command is valid only in the topmost context.[3] Execution resumes from the current execution point in the selected context.

$$\begin{array}{l} proceed(c) \equiv \\ \quad RESUME(c) \end{array}$$

Report the Procedure Traceback

A list of contexts representing the procedure traceback (ordered from bottommost to topmost) is created by starting with the topmost context and repeatedly accessing the calling context. The procedure traceback with topmost context c is computed as follows:

$$\begin{array}{l} procedure_traceback(c) \equiv \\ \quad \text{if } CALLER(c) = nil \\ \quad\quad c \\ \quad \text{else} \\ \quad\quad procedure_traceback(CALLER(c)) \cdot c \end{array}$$

Step

Single stepping executes the next instruction, while keeping control in the selected context. "Keeping control" means that procedure calls are executed as single statements, that the debugger does not step through procedures that are called.

Stepping can be performed from any context. The *step* command takes two arguments: the selected context c and the sequence of contexts representing the entire procedure traceback, $\vec{c}$. The description of the *step* command has three cases.

The first case is when the topmost context in the procedure traceback is selected and its current execution point corresponds to a procedure call.[4] The procedure is invoked and executes to completion, returning control to the selected context. After the initial call is performed, the auxiliary function *complete_callees* is used to repeatedly invoke the $EXECUTE_STMT$ operation until a return to the selected context is executed.

$$\begin{array}{l} step(c, \vec{c}) \equiv \\ \quad complete_callees(c, EXECUTE_STMT(c)) \end{array}$$

The second case is when the selected context's current execution point corresponds to some statement that is not a call. In this case, only the single statement is executed. Notice that if the current statement is a return then the calling context will have control after stepping.

[3] Error conditions are omitted from the definitions of the commands.

[4] The term "procedure call" encompasses all types of calls, including function calls.

$$step(c, \vec{c}) \equiv$$
$$\quad EXECUTE_STMT(c)$$

The third case is when a non-topmost context is selected. All contexts above the selected context execute to completion, returning control to the selected context. A step command on a non-topmost context invokes *complete_callees* to carry out the command.

$$step(c, \vec{c}) \equiv$$
$$\quad complete_callees(c, last(\vec{c}), \vec{c})$$

The auxiliary function $complete_callees(c, t, \vec{c})$ executes starting in context t until a return to context c is executed.

$$complete_callees(c, t, \vec{c}) \equiv$$
$$\quad \text{if } c = t$$
$$\quad\quad \langle c, \vec{c} \rangle$$
$$\quad \text{else}$$
$$\quad\quad complete_callees(c, EXECUTE_STMT(t, \vec{c}))$$

Call

Call is used to step deeper into a computation. A *call* command can be issued only in the topmost context, at a point corresponding to a procedure call. The call is executed, resulting in a new topmost context. The *call* command is the same as executing a call instruction.

$$call(c, \vec{c}) \equiv$$
$$\quad EXECUTE_STMT(c, \vec{c})$$

Restart

The Smalltalk-80 debugger will restart a procedure from its beginning. It does not reinitialize variables or undo side effects, but will discard all contexts above the selected context in the procedure traceback. *Restart* is generally used to resume execution after making a correction, without starting over completely. Restarting from context c is defined as follows:

$$restart(c, \vec{c}) \equiv$$
$$\quad SET_STMT(c, first_statement(PROCEDURE(c)), \vec{c})$$

Modify a Procedure

When a procedure is modified, all future calls to the procedure will invoke the new version. However, there might be one or more invocations of the old version of the procedure, each represented by a context on the procedure traceback. In general, it is impossible to convert these contexts to use the new version of the procedure, although there are many special cases where it is possible.

In Smalltalk-80, procedures can be modified inside or outside the debugger. Changes made outside the debugger do not effect existing contexts. Changes inside the debugger are made to a particular context, and do not effect any other contexts below it in the procedure traceback. Changing a procedure in a context c causes a new context to be created and

the values of all variables are copied from c to the new context. All contexts above the c are discarded. No attempt is made to resume execution where it left off after recompiling. Changing the procedure in context c to p is defined as follows:

$$\begin{aligned} &\mathit{modify_procedure}(c, p, \vec{c}) \equiv \\ &\quad \mathit{assign_values}(\text{NEW_CONTEXT}(\mathit{CALLER}(c), p, \vec{c}), c) \end{aligned}$$

Evaluate an expression
Variable values in the selected context can be accessed and modified when an expression is evaluated. The expression can be treated as a separate procedure that invokes the operations *VAR_VALUE* and *ASSIGN_VALUE* on the selected context to access or change the values of the context's variables.

Breakpoints
The Smalltalk-80 debugger does not have breakpoints. Instead, the user can edit any procedure to add or remove a call to the debugger. This is not as powerful as real breakpoints, because the debugger will always restart from the beginning of a procedure.

The context-oriented model of debugging can easily handle real breakpoints. A breakpoint is still a procedure call to the debugger. Thus, setting or removing a breakpoint in a procedure is equivalent to changing and recompiling the procedure. However, the procedure is changed in a very specific way. None of the original lines of the procedure have been deleted (except for breakpoints), so it is easy to convert contexts that use the procedure by setting the current execution point of the new context to be the same statement as the current execution point of the old context. Thus, past invocations of the procedure will also have the correct breakpoints.

Traditionally, setting or removing breakpoints requires much less work than compiling a procedure, but we will see that in the worst case they are equivalent.

Watchpoints
Another popular debugger feature not present in the Smalltalk-80 debugger is the watchpoint, which is a breakpoint that is triggered by a predicate, not a program location. Watchpoints can be implemented by evaluating the predicate any time one of the variables on which it depends is changed. Thus, setting or removing a watchpoint is equivalent to setting or removing a breakpoint on every definition of a variable that is used in its predicate.

3 Implementing Contexts in Optimized Programs

The previous section reduced the problem of debugging optimized code to implementing contexts. In fact, it is easy to implement a context if the entire source-level program state can be recovered at every execution point from which the debugger might be invoked. These execution points are called *inspection points*. A location l in a procedure is an inspection point if the debugger can be invoked while the procedure has an active invocation with l as its current execution point. Inspection points correspond to breakpoints, places where run-time errors might occur, places where a user interrupt might occur, and procedure calls where the debugger might be invoked (due to a breakpoint or run-time error) before the procedure returns. Section 4 describes how the compiler identifies inspection points.

As the compiler performs optimizations, it computes recovery information to ensure that the entire source-level program state (current execution locations, variable values, and the procedure traceback) can be recovered from the low-level state at every inspection point. Each procedure has recovery information that will be used by its contexts during debugging.

Context operations that change part of the program state can always be implemented by recovering the source-level state and using this information to convert the affected part of the low-level state to unoptimized form before making the change. If the state associated with a context is converted to unoptimized form then an unoptimized version of its procedure must be used when the program is resumed. The main advantage of this technique is that it allows more optimizations to be performed, because changes are only made to the unoptimized state. If this technique of converting part of the program state to unoptimized form were not used then debugging with expected behavior would require optimization to be restricted so that:

- Places encountered during single stepping would be inspection points and the source-level program state could be recovered at them.
- Operations would never be reordered, which would rule out optimizations like code motion out of loops.
- Optimizations such as common subexpression elimination would have to be restricted to ensure that their effects would not inhibit the debugger's ability to change the value of any variable.

Thus, converting part of the program state to unoptimized form results in fewer inspection points at which the source-level program state must be recoverable and permits more optimizations.

3.1 Context Data Structures

Section 2.1 described contexts as an abstract data type. Contexts are a source-level view of the program state that is independent of the implementation of the run-time system. An implementation of contexts must make the source-level program state correspond with the low-level program state. Just as a context represents the source-level state of a procedure call, an *activation record* represents the low-level state of a procedure call.

Inline procedure expansion will cause several contexts to correspond to a single activation record. Thus, part of the recovery information that is stored with each procedure describes the contexts to create at each inspection point. The rest of the recovery information is used to retrieve the unoptimized values of variables, to map from executable code locations to inspection points, and to map from source statements to executable code locations.

When the debugger is invoked, it uses the recovery information for each active procedure to create a sequence of contexts that represents the source-level program state. The recovery information that is associated with each inspection point is:

- A list of $\langle$*procedure, source statement*$\rangle$ pairs that is used for recovering the procedure traceback (to unravel the effects of inline procedure expansion, as discussed in section 4.3.3). When the debugger is invoked, this list is used to create one or more contexts to correspond to each activation record.

- A function for computing the *unoptimized values* of variables from their *optimized values* at inspection points. Contexts provide unoptimized values, while activation records store optimized values.

The recovery information associated with the procedure as a whole is:

- A mapping from executable code locations to inspection points.
- A mapping from source statements to executable code locations. This mapping is used for single stepping, which happens only in unoptimized code. Hence, it need not be maintained by the compiler during optimization.

3.2 A Formal Description of the Context Data Structures

This section describes the lower-level data structures that will be used to describe how contexts can be implemented in the presence of optimization. We use the following conventions:

- All functions are total functions and produce the value $\perp$ where they might otherwise be undefined.
- For any function $f : D \rightarrow B$, $f[d/b]$ denotes the function that differs from f only at $d \in D$ by taking on the value $b \in B$.
- For any function $f : D \rightarrow B$, $dom(f) = \{d \in D | f(d) \neq \perp\}$.
- The powerset of D is denoted $\mathbf{P}(D)$.

The domains C, P, V, VAL, $STMT$, and NIL were defined in Section 2.1 to describe the view of the debugger. The implementation of contexts (C) also uses the following lower-level domains:

A	Activation records
S	Stopping places
OFF	Offsets in executable code

3.2.1 Activation Records

An activation record holds the low-level state of the invocation of a procedure. The components of an activation record $a \in A$,

$$a = \langle pc_a, proc_a, caller_a, values_a \rangle$$

have the following interpretation:

$pc_a \in OFF$
: The current low-level execution point, which always corresponds to an inspection point

$proc_a \in P$
: The procedure that is executing in a

$caller_a \in (A + NIL)$
: The previous activation record (nil for the first activation record)

$values_a \in V \rightarrow VAL$
: Variable values

A *sequence of activation records* $\vec{a} = a_1 \ldots a_n$ is a sequence where, for $i = 2 \ldots n$, $caller_{a_i} = a_{i-1}$, and $caller_{a_1} = nil$.

The debugger assumes that variables are stored in activation records instead of in registers. This means that variables stored in registers must be spilled to memory before they are needed, and they must be spilled in such a way that they can be found. An easy way to ensure this is to spill all live registers at every procedure call, i.e. the caller saves the registers. A call to the debugger can be treated as a procedure call, so all live variables in the topmost activation record will be spilled to memory when the debugger is invoked. If the callee saves registers then variables allocated to registers will be stored in the "wrong" activation record, i.e. in an activation record other than the one created for the procedure in which the variables were declared. These variables have to be recovered by searching up the procedure traceback for the first procedure that saved the register. Coutant *et. al.* describe a solution to this [CMR88]. For the sake of simplicity, we assume that variables are stored in the activation record of the procedure in which they were declared.

Note that we are not assuming that a variable will always be spilled to the same location in the activation record. We have not described how an activation record maps variables to their values. It would be possible for the activation record to use a different mapping of variables to offsets within the activation record at every inspection point. We have ignored this detail of activation records, since it would add uninteresting complications to the optimizations and recovery information.

There are two auxiliary functions on activation records. The function *eval(e,ar)* provides the value of expression e in activation record ar. The function *update(ar,x,v)* returns an activation record that is just like ar except that variable x has value v.

3.2.2 Contexts

The components of a context $c \in C$,

$$c = \langle stmt_c, proc_c, caller_c, ar_c \rangle$$

have the following interpretation:

$stmt_c \in STMT$
: The source statement that is currently executing in this context

$proc_c \in P$
: The procedure that is executing in this context

$caller_c \in (C + NIL)$
: The calling context (nil for the bottommost context)

$ar_c \in A$
: The corresponding activation record, i.e. the activation record that contains the low-level program state that corresponds to c

A *sequence of contexts* $\vec{c} = c_1 \ldots c_n$ is a sequence where, for $i = 2 \ldots n$, $caller_{c_i} = c_{i-1}$, and $caller_{c_1} = nil$.

3.2.3 Procedures

The components of a procedure $p \in P$,

$$p = \langle inspect_p, exec_to_inspect_p, source_to_exec_p, vars_p \rangle$$

have the following interpretation:

$inspect_p \in \mathbf{P}(S)$
: The inspection points in the procedure

$exec_to_inspect_p \in OFF \rightarrow S$
: A mapping from executable code locations to inspection points

$source_to_exec_p \in STMT \rightarrow OFF$
: A mapping from source statements to executable code locations. This mapping is used only for unoptimized procedures.

$vars_p \in \mathbf{P}(V)$
: Local variables

3.2.4 Inspection points

Recall that inspection points correspond to points in the program at which the debugger might be invoked. Information for computing the source-level program state from the low-level state is associated with each inspection point. The components of an inspection point $s \in S$,

$$s = \langle trace_s, recover_s \rangle$$

have the following interpretation:

$trace_s \in (P \times STMT)^+$
: A list of $\langle$*procedure, source statement*$\rangle$ pairs that is used for reporting the procedure traceback at the inspection point. The first element in the list always corresponds to a procedure that is actually executing, and the remaining elements correspond to calls that were eliminated by inline procedure expansion. The ordering of the list reflects the original calling sequence. The first component of the list element $(trace_s)_i$ is denoted $proc_{(trace_s)_i}$ and the second component is denoted $stmt_{(trace_s)_i}$.[5]

$recover_s \in V \rightarrow (A \rightarrow VAL)$
: A function that maps source-level variables to *recovery functions* that can be evaluated by the debugger to recover the unoptimized values of variables. A recovery function is a function from activation records to values. It produces the unoptimized value of a single variable.

[5]Here, the list $trace_s$ is parenthesized to emphasize that the subscript s is part of the name of the list, while the subscript i is an index into the list.

3.2.5 Unoptimized Programs

It is instructive to see how these data structures are used to represent unoptimized programs. An unoptimized procedure has a direct correspondence between the source-level and low-level program states. For an unoptimized procedure p and inspection point $s \in inspect_p$,

- There is a one-to-one correspondence between contexts and activation records (because no inline substitution has been performed), so $trace_s$ is a list containing the single pair $\langle p, l \rangle$ where l is the source statement in p that corresponds to the inspection point s.
- Every variable's unoptimized value is equal to its optimized value, i.e. for every variable $x \in vars_p$, $recover_s(x) \equiv \lambda ar.eval(ar, x)$.

Applying an optimization to a procedure p results in a modified procedure p' where the difference between the recovery information for p' and p reflects the effects on the program state of the optimization that was performed. Section 4 will describe how recovery information is produced as part of optimizing a procedure.

3.3 Using the Recovery Information

Recovery information is used in two ways. First, it is used to create the initial sequence of contexts, i.e. the procedure traceback. Once this sequence of contexts has been created, many of the operations are easy to implement, since most of the hard work was done either by the compiler or at the time the procedure traceback was created. Second, recovery information is used to implement the operations on contexts. The operations that only return part of the state are easy to implement. However, the operations that change state must first convert the activation record to unoptimized form, which is more complicated.

3.3.1 Creating a Source-level Representation of the Program State

The list $trace_s$ associated with each inspection point s is used to create one or more contexts to correspond to each activation record. The contexts made from these lists are then linked together, resulting in a sequence of contexts that represents the entire source-level state, as shown in the "before conversion" part of Figure 2. Suppose the low-level state consists of the sequence of activation records $a_1, a_2, \ldots, a_n$ when the debugger is invoked, where a_n is the topmost activation record. For $i = 1 \ldots n$ let

$$s_i = exec_to_inspect_{proc_{a_i}}(pc_{a_i})$$

$$d_i = length(trace_{s_i})$$

Here, s_i is the inspection point that corresponds to a_i's current execution point, and d_i is the number of contexts that will be created to correspond to a_i, which is determined by the number of procedure calls that were expanded inline at s_i. A context is created for each procedure that is actually executing, as well as for each active "invocation" of an inline-substituted procedure.

The contexts $c_{i,1}, \ldots, c_{i,d_i}$ are created corresponding to each activation record $a_i, i = 1 \ldots n$, one context for each element in the list $trace_{s_i}$. These d_i contexts represent in source-level terms the state contained in the single activation record a_i. Context $c_{i,1}$ corresponds to the procedure that is executing in a_i, and the remaining contexts $c_{i,2}, \ldots, c_{i,d_i}$ correspond to calls that were expanded inline.

$$c_{i,j} \equiv \langle stmt_{c_{i,j}}, proc_{c_{i,j}}, caller_{c_{i,j}}, ar_{c_{i,j}} \rangle \quad i = 1 \ldots n, j = 1 \ldots d_i$$

The contexts are linked together as follows: Contexts corresponding to a single activation record are linked together in the order given by the list of ⟨*procedure, source statement*⟩ pairs. The caller for the first context corresponding to a given activation record is the last context that corresponds to the previous activation record. The caller for the first context in the procedure traceback is nil:

$$caller_{c_{i,j}} \equiv \begin{cases} c_{i,j-1} & i = 1 \ldots n, j = 2 \ldots d_i \\ c_{i-1,d_{i-1}} & i = 2 \ldots n, j = 1 \\ nil & i = 1, j = 1 \end{cases}$$

The procedure and current execution point for each context are copied from the corresponding element in the list of ⟨*procedure, source statement*⟩ pairs.

$$\left.\begin{array}{l} stmt_{c_{i,j}} \equiv stmt_{(trace_{s_i})_j} \\ proc_{c_{i,j}} \equiv proc_{(trace_{s_i})_j} \end{array}\right\} i = 1 \ldots n, j = 1 \ldots d_i$$

There can be many contexts for each activation record. Procedures that have been expanded inline will not have their own activation records. However, they will still need to be reported in the procedure traceback if they contain inspection points. Each activation record has a procedure, which might have had many other procedures substituted into it. The recovery information contained in an activation record's procedure is used to unravel the effects of optimization on *all of the state contained in the activation record*, which may correspond to several contexts.

Every context created from a single list of ⟨*procedure, source statement*⟩ pairs corresponds to the same activation record:

$$ar_{c_{i,j}} = a_i \quad i = 1 \ldots n, j = 1 \ldots d_i$$

The topmost context in the source-level, unoptimized program state is c_{n,d_n}. All other contexts in the procedure traceback are accessible from this context by following the "caller" links.

3.3.2 A Definition of the Context Operations

This section first shows how to implement each of the state-accessing operations. It then describes how part of the program state is converted to unoptimized form when a state-changing operation is invoked. Finally, it shows how to implement each of the state-changing operations.

Many of the operations require determining the "current inspection point" in a context. In a context c, the current inspection point is computed using the low-level execution point stored in the activation record and the mapping from executable code locations to inspection points. The current inspection point in a context c is $exec_to_inspect_{proc_{ar_c}}(pc_{ar_c})$.

State-accessing Operations
Section 2.1 stated that all contexts possess the following state-accessing operations:

$$CALLER : C \rightarrow (C + NIL)$$
$$VAR_VALUE : (C \times V) \rightarrow VAL$$
$$CURRENT_STMT : C \rightarrow STMT$$
$$PROCEDURE : C \rightarrow P$$

When a context c is created, its link to the calling context, $caller_c$, is initialized. The current source statement, $stmt_c$, and the procedure $proc_c$ are also set when the context is created. Thus, all the work is done when the contexts are created, as described in Section 3.3.1.

$$CALLER(c) \equiv caller_c$$
$$CURRENT_STMT(c) \equiv stmt_c$$
$$PROCEDURE(c) \equiv proc_c$$

The unoptimized value of a variable v in a context c is computed by evaluating the recovery function representing the unoptimized value of v at s, where s is the current inspection point in context c, i.e. $s = exec_to_inspect_{proc_{ar_c}}(pc_{ar_c})$.

$$VAR_VALUE(c, v) = recover_s(v)(ar_c)$$

Converting to Unoptimized Form
The first step in converting to unoptimized form is to determine the part of the program state that needs to be converted. Recall that there is a many-to-one correspondence between contexts and activation records. A single activation record may contain the state of several source-level procedure invocations. Thus, a state-changing operation on context c will convert activation record ar_c to unoptimized form. Making this change to the low-level state affects every context that shares an activation record with c, i.e. every context c' such that $ar_{c'} = ar_c$. Each of these contexts must be modified to refer to one of the new, unoptimized activation records.[6] Figure 2 illustrates this conversion.

Converting an activation record to unoptimized form requires obtaining an unoptimized version of every procedure that is executing in a context that corresponds to the activation record. The unoptimized procedure can be obtained by recompiling the source code for the procedure with optimizations turned off, or by having both optimized and unoptimized versions available at all times, or by a combination of both techniques. In the unoptimized version of the procedure, every source statement is treated as an inspection point so that the (unoptimized, trivial) recovery information is available at all locations that might be encountered during single stepping. Converting one context for an invocation for a procedure does not require converting any other contexts for that procedure. Thus, if there is an invocation of a converted procedure in a different activation record then both unoptimized and optimized versions of the procedure will be simultaneously active. This poses no problems — the optimized version of the procedure is used in all future invocations, and the unoptimized version can be discarded as soon as it has no active invocations.

The function $convert : (C \times C^+) \rightarrow (C \times C^+)$ converts the affected part of the program state to unoptimized form. The first argument is the context on which the state-changing

[6] Of course, the affected part of the program state may already be in unoptimized form due to an earlier change made in the selected context c. The affected state need be converted only once, regardless of how many changes are made to that part of the program state during a debugging session.

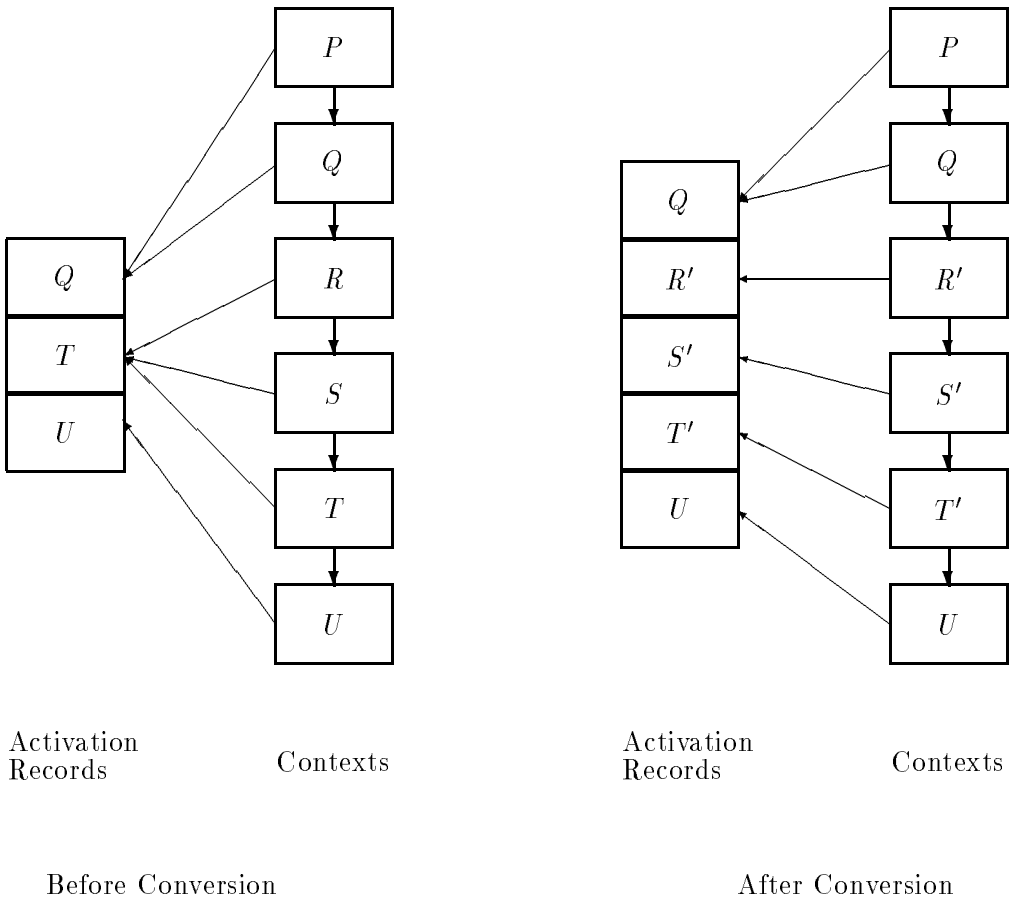

Figure 2: A state-changing operation is invoked on the context for procedure R. The state contained in the activation record corresponding to this context is converted to unoptimized form by replacing it with three unoptimized activation records and modifying all contexts that referred to the affected activation record to refer to one of the new, unoptimized activation records and to an unoptimized procedure. All contexts and activation records are labeled with the procedures they are executing. Procedures R', S', and T' are the unoptimized versions of R, S, and T, respectively (the stack grows upward).

operation is being invoked, and the second argument is the sequence of contexts that represents the entire source-level program state. The second argument is required because, as we have seen, more than one context may be affected by a change.

Given context c and procedure traceback $\vec{c}$, $convert(c, \vec{c})$ returns a pair $\langle c', \vec{c'} \rangle$, where c' is the context c modified to refer to an unoptimized activation record and to an unoptimized version of its procedure, and $\vec{c'}$ is $\vec{c}$ modified so that each context that originally shared an activation record with c now refers to one of the new, unoptimized activation records and an unoptimized procedure. There is a precise definition of *convert* in [Zur89].[7]

Contexts always provide an unoptimized view of the program state, i.e. of the activation records. As long as the contexts are only used to read the state then it is not necessary to convert the activation records, so they can remain in their optimized form. However, activation records are changed to their unoptimized form before contexts are used to write to them.

This section (and Figure 2) assume that an unoptimized activation record has a different format than an optimized one, and that each unoptimized context has its own unoptimized activation record. Neither assumption is necessary, but they fit in well with the notation used in the paper. They are bad assumptions for languages like C that allow pointers to local variables. For those languages, converting an activation record must not change the format of the activation record or the address of any variable that is the target of a pointer. This implies that optimized and unoptimized activation records will have the same size, that many variables will have the same location in both forms of the activation record, and that the activation record for a procedure p_1 in which procedure p_2 has been in-lined is the concatenation of the activation records of the unoptimized procedures.

Change the Value of a Local Variable

To assign the value b to the variable v in context c, the affected state is converted to unoptimized form and the function $values_{ar_c}$ is modified to assign the new value.

$$
\begin{array}{l}
ASSIGN_VALUE(c, v, b, \vec{c}) \equiv \\
\quad \text{let } \langle c', \vec{c'} \rangle = convert(c, \vec{c}) \text{ in} \\
\quad\quad replace_context(c', \vec{c'}, \langle stmt_{c'}, proc_{c'}, caller_{c'}, update(ar_{c'}, v, b) \rangle)
\end{array}
$$

Change the Current Statement

The only actions that need be performed after converting to unoptimized form are to set the current execution location in the selected context and in the activation record.

$$
\begin{array}{l}
SET_STMT(c, stmt\vec{c}) \equiv \\
\quad \text{let } \langle c', \vec{c'} \rangle = convert(c, \vec{c}) \text{ in} \\
\quad\quad replace_context(c', \vec{c'}, \langle stmt, proc_{c'}, caller_{c'}, a' \rangle)
\end{array}
$$

where a' is $ar_{c'}$ with its current execution location changed, i.e.

$$a' \equiv \langle source_to_exec_{proc_{ar_{c'}}}(stmt), proc_{ar_{c'}}, caller_{ar_{c'}}, values_{ar_{c'}} \rangle$$

[7] The formal definition of *convert* is not particularly complex, but is rather lengthy.

Execute the Next Source Statement
The meaning of "execute a statement" depends on the semantics of the source language. The statement is executed only after the affected part of the program state has been converted to unoptimized form, so defining what it means to execute a single statement has no direct bearing on the issue of debugging optimized code. We assume the existence of a function $m : (C \times C^+) \rightarrow (C \times C^+)$ that embodies the semantics of the various statements. Given a context c in which to execute a single statement and the procedure traceback $\vec{c}$, $m(c, \vec{c}) \equiv \langle c', \vec{c'} \rangle$, where c' is the context that has control after executing the statement and $\vec{c'}$ is the modified procedure traceback. For example, suppose the current statement in context c is an assignment $x \leftarrow b$. Then

$$m(c, \vec{c}) \equiv replace_context(c, \vec{c}, \langle stmt_c, proc_c, caller_c, update(ar_{c'}, v, b) \rangle).$$

Given this meaning function, $EXECUTE_STMT$ is defined as follows:

$$EXECUTE_STMT(c, \vec{c}) \equiv m(convert(c, \vec{c}))$$

Resume Execution Since the low-level state is updated during debugging to keep it consistent with the source-level state, no special actions need be performed to resume execution. Control is given to the selected context's activation record, and execution is resumed from this activation record. We assume the existence of the function *resume_process* which is used to resume execution of a suspended process.

$$RESUME(c) \equiv resume_process(ar_c)$$

4 Computing Recovery Information During Optimization

The previous section showed that a debugger can provide expected behavior if it can recover the source-level program state at every inspection point. Thus, the problem reduces to identifying inspection points and computing recovery information for each one. The compiler carries out an optimization only if it can produce recovery information for all affected inspection points. Optimizations modify both the code for a procedure and its recovery information. This paper does not describe how the code is modified, since its focus is not on the code transformations associated with optimizations, but instead to show how optimizations are modified to allow expected debugger behavior. The modified optimizations always apply the same code transformations as the traditional form of the optimization. They differ from the traditional optimizations by restricting the cases in which an optimization can be applied and by computing recovery information.

4.1 Identifying Inspection Points

A location l in a procedure p is an inspection point if the debugger might be invoked while p has an active invocation with l as its current execution point. There are three kinds of inspection points:

1. A breakpoint that was set by the user.

2. A point where a run-time error might occur. How the compiler identifies where an error might occur depends on the source language.

3. A procedure call where the debugger might be invoked (due to a breakpoint or run-time error) before the procedure returns.

Without loss of generality, we can view the first two kinds of inspection points as being calls to a special procedure that invokes the debugger, which will hereafter be called a *halt*. A result of this is that every inspection point corresponds to a procedure call, but the user can set a breakpoint anywhere in the program.

When a halt is executed, the current execution point in every activation record corresponds to a procedure call. This is because activation records are only created when the procedure executing the previous activation record performs a procedure call, and that procedure call will not be completed until the new activation record is taken off the procedure traceback.

A procedure is *haltable* if it is a halt, or if it contains one or more inspection points. A procedure that meets neither of the above conditions is *unhaltable*. Calls to haltable procedures are inspection points.

Making a procedure haltable by adding a breakpoint or changing a procedure that it calls can require replacing optimized procedures with unoptimized ones. This is similar to interprocedural optimizations like in-line substitution, which cause changes to one procedure to invalidate the optimizations on another. Although users might expect that it will take a relatively long time to recompile a procedure, they will expect adding or deleting a breakpoint to take little time. Thus, there should be a fast way to install unoptimized procedures when a breakpoint is added. If there is an unoptimized version of every procedure then the time to install unoptimized procedures depends primarily on the time to determine which procedures become haltable. This means that the problem of quickly adding breakpoints reduces to the problem of quickly finding which procedures become haltable.

An incremental algorithm for finding inspection points could use a call graph except for the problem of cycle in the call graph. Making a procedure haltable causes all procedures that call it to become haltable, but making a procedure unhaltable will not necessarily make the procedures that call it unhaltable. Thus, haltability is easy to propagate along a call graph with cycles but unhaltability is not.

It is easy to incrementally determine inspection points if the call graph is represented properly, i.e. if cycles in the call graph do not have to be checked. Each strongly connected component is replaced with a single node that represents all the nodes in the strongly connected component—this eliminates cycles in the call graph. A node is haltable if and only if the procedures that it represents are haltable. An edge from node A to node B means that all the procedures represented by A can call all the procedures represented by B. Since this graph has no cycles, changes in haltability can easily be propagated by examining the predecessors of the changed node.

4.2 Static Single Assignment Form

Many optimizations require flow analysis and data dependency information. The Typed Smalltalk compiler uses static single assignment (SSA) form [CFR*89] to represent this information, so this paper uses SSA representation to describe optimizations. The algorithms could be modified to use a different representation for the dependency information, such as def-use chains [ASU86], though algorithms based on SSA form tend to be simpler than those based on def-use chains.

A program in SSA form has the property that every variable is assigned exactly once. A program is converted to SSA form by replacing each variable with multiple *versions* of

that variable, where a different version corresponds to each assignment to the variable. For straight-line code it is easy to rename variables so that there is only one definition of each variable. However, this is not possible in the presence of conditionals, since multiple definitions of a variable can reach the same point in the program. In SSA form, multiple reaching definitions are resolved by inserting a *ϕ-assignment* at the beginning of a flow node. A ϕ-assignment has the form $x_{n+1} \leftarrow \phi(x_1, \ldots, x_n)$, where $x_1, \ldots, x_{n+1}$ are variables, and n is the number of predecessors possessed by the flow node in which the ϕ-assignment appears. No executable code is generated for a ϕ-assignment. Rather, ϕ-assignments are used to represent dependency information that traditionally takes the form of multiple reaching definitions. For example, suppose the assignments $x \leftarrow 7$ and $x \leftarrow 9$ reach a statement in a program before SSA conversion is performed. When the program is converted to SSA form, the assignments $x \leftarrow 7$ and $x \leftarrow 9$ are replaced by the assignments $x_1 \leftarrow 7$ and $x_2 \leftarrow 9$, and a ϕ-assignment $x_3 \leftarrow \phi(x_1, x_2)$ is inserted in the flow node where paths from the definitions of x_1 and x_2 join.

SSA form makes computing data flow information easy, since every use of a variable is reached by exactly one definition. It also makes thinking about optimization easier, since multiple reaching definitions have been replaced by ϕ-assignments. Figure 3 shows a flowgraph representation of a program prior to converting to SSA form. Figure 4 shows the same program after converting to SSA form. Converting a program to SSA form modifies the recovery information. In particular, variable references in the recovery functions are renamed in exactly the same way as uses of variables in the rest of the program.

4.3 Performing Specific Optimizations

4.3.1 Optimizations That Do Not Change the Program State

Many optimizations do not alter the correspondence between the optimized and unoptimized program states. Examples of such optimizations are constant propagation, common-subexpression elimination, copy propagation, and constant folding. All of these optimizations replace expressions with cheaper, equivalent expressions. Thus, these optimizations do not affect the values of any variables, but can make it difficult for the debugger to change the optimized program state. All of the difficulties caused by these optimizations are eliminated by converting the affected part of the low-level state to unoptimized form.

4.3.2 Elimination of Unreachable Code

Optimizations that find and eliminate unreachable code are especially important to debugging. For example, if the compiler can determine that the subscript will always be in bounds for a particular array access, then the code that performs the run-time subscript check can be eliminated. The obvious benefit of this optimization is that it removes the run-time overhead of performing the subscript check. An added benefit is that eliminating the code that performs the subscript check also eliminates the halt that signals the subscript error. Eliminating this inspection point permits more optimizations. In addition, if eliminating this inspection point makes the procedure unhaltable then all of the procedure's callers benefit, since they will now have fewer inspection points.

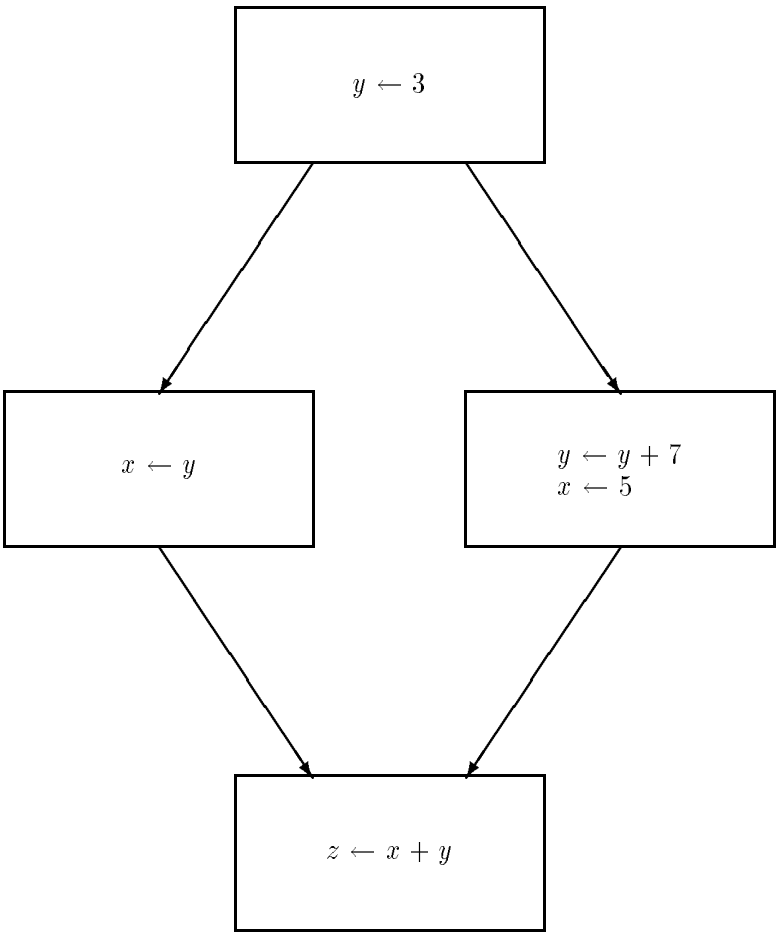

Figure 3: A program before converting to SSA form

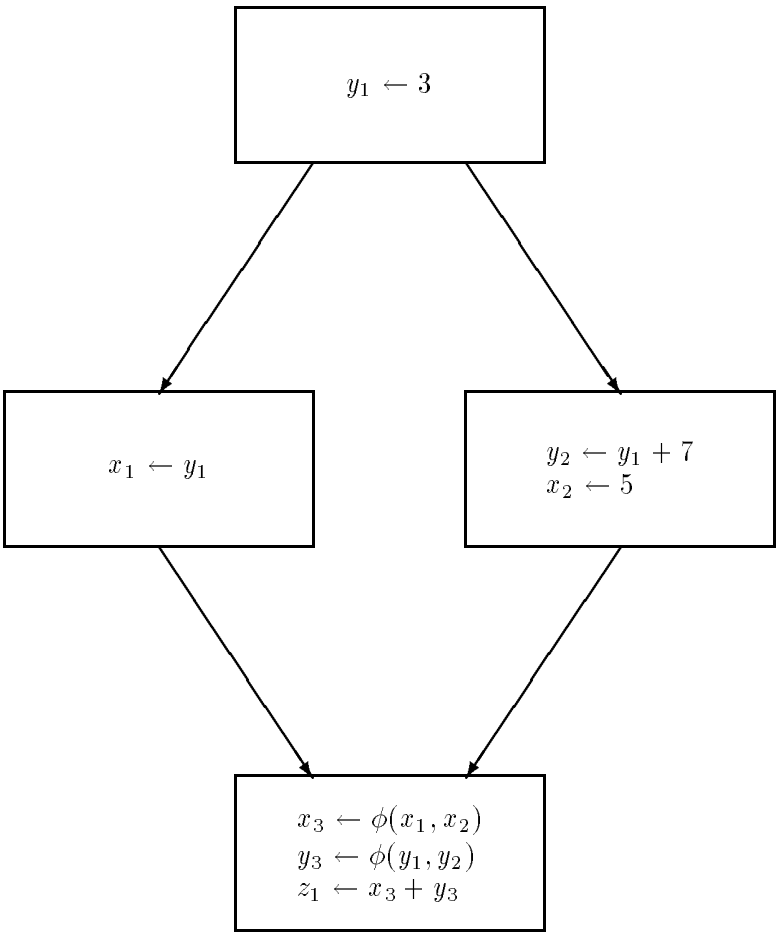

Figure 4: A program in SSA form

4.3.3 Inline Procedure Expansion

Inline procedure expansion consists of replacing a procedure call with the body of the called procedure. Its main purposes are to remove the overhead of performing a procedure call and to permit optimizations across procedure call boundaries.

Inline procedure expansion never prevents the debugger from providing expected behavior. However, the recovery information must be updated if the replacement is made at an inspection point. If the replacement is made at a non-inspection point then the procedure being substituted inline is not haltable, so debugging activity can never occur at the point of the replacement.

Expanding a call to procedure q requires updating $trace_s$ at every inspection point s in the body of q, so that a context will be created for the eliminated call. The recovery information for variables must also be updated when inline procedure expansion is performed. The new information must include variables local to the substituted procedure, since these variables will reside in an activation record for the substitutor.

Expanding a call to procedure q with inspection points $inspect_q = \{s_1, \ldots, s_n\}$ at inspection point s_0 in procedure p results in procedure p', where

$$p' \equiv \langle inspect_{p'}, exec_to_inspect_p, source_to_exec_p, vars_p \rangle.$$

Inspection point s_0 is replaced by inspection points $s'_1, \ldots, s'_n$, where $s'_1, \ldots, s'_n$ are $s_1, \ldots, s_n$ updated to include information from s_0.

$$inspect_{p'} \equiv inspect_p \cup \{s'_1, \ldots, s'_n\} - \{s_0\}$$

At any inspection point s'_i, $i = 1 \ldots n$, the procedure traceback must include contexts made from the list $traces_i$ preceded by contexts made from the list $traces_0$.

$$trace_{s'_i} \equiv trace_{s_0} \cdot trace_{s_i} \quad i = 1 \ldots n$$

This correctly handles cases where earlier substitutions were performed in either p or q.

At every inspection point s'_i, $i = 1 \ldots n$, the function for recovering unoptimized variable values, $recover_{s'_i}$, includes information from both procedures p and q, since q's local variables will now reside in p's activation record. For $i = 1 \ldots n$, the function $recover_{s'_i}$ is the union of the function $recover_{s_0}$ and the function $recover_{s_i}$. Variables with the same name that are local to different procedures are considered unique, so there are no problems with name conflicts. Since activation records are responsible for mapping variable names to values, we can ignore the fact that the offsets of some of the variables into the activation record have changed.

After the substitution, the inspection point s_0 has been eliminated, inspection points $s'_1 \ldots s'_n$ have been added to p', and all of the new inspection points include recovery information from the eliminated inspection point.

The following example shows how the list of $\langle$*procedure, source statement*$\rangle$ pairs is modified when inline procedure expansion is performed. Five procedures (A, B, C, D, and E) are shown below. Every statement shown corresponds to an inspection point, and the list of $\langle$*procedure, source statement*$\rangle$ pairs for each inspection point appears in the right column.

$$
\begin{array}{ll}
A & \\
\quad call\ \ B & (\langle A, call\ \ B\rangle) \\
 & \\
B & \\
\quad call\ \ C & (\langle B, call\ \ C\rangle) \\
\quad call\ \ D & (\langle B, call\ \ D\rangle) \\
 & \\
C & \\
\quad call\ \ E & (\langle C, call\ \ E\rangle) \\
 & \\
D & \\
\quad halt_1 & (\langle D, halt_1\rangle) \\
 & \\
E & \\
\quad halt_2 & (\langle E, halt_2\rangle)
\end{array}
$$

For an invocation of procedure A, when $halt_1$ in procedure D is executed, activation records for D, B, and A will be on the stack. At this point, every list of $\langle$ *procedure, source statement* $\rangle$ contains a single element, so the sequence of contexts created from the lists (procedures and current execution locations) will be as follows (the stack grows upward):

$$
\begin{array}{ll}
D & halt_1 \\
B & call\ \ D \\
A & call\ \ B
\end{array}
$$

The following contexts will be created for an invocation of A when $halt_2$ in procedure E is executed:

$$
\begin{array}{ll}
E & halt_2 \\
C & call\ \ E \\
B & call\ \ C \\
A & call\ \ B
\end{array}
$$

Now suppose the call to E in procedure C is expanded inline. The modified procedure and list of $\langle$*procedure, source statement*$\rangle$ pairs are shown below:

$$
\begin{array}{ll}
C & \\
\quad halt_2 & (\langle C, call\ \ E\rangle\langle E, halt_2\rangle)
\end{array}
$$

The call to B in procedure A is now expanded inline, resulting in the following:

$$
\begin{array}{ll}
A & \\
\quad call\ \ C & (\langle A, call\ \ B\rangle\langle B, call\ \ C\rangle) \\
\quad call\ \ D & (\langle A, call\ \ B\rangle\langle B, call\ \ D\rangle)
\end{array}
$$

Finally, the calls to C and D in procedure A are expanded inline, resulting in the following:

$$
\begin{array}{ll}
A & \\
\quad halt_2 & (\langle A, call\ \ B\rangle\langle B, call\ \ C\rangle\langle C, call\ \ E\rangle\langle E, halt_2\rangle) \\
\quad halt_1 & (\langle A, call\ \ B\rangle\langle B, call\ \ D\rangle\langle D, halt_1\rangle)
\end{array}
$$

In the optimized version of procedure A, when either $halt_1$ or $halt_2$ is executed, only an activation record for A will be on the stack. The lists corresponding to $halt_1$ and $halt_2$ in A contain information to create the same sequence of contexts that would have been created if no inline procedure expansion had been performed.

Zellweger [Zel84] shows how to solve the location problems caused by inline procedure expansion, but does not address any of the data problems.

4.3.4 Global Dead-Store Elimination

An assignment $l : x \leftarrow e$ can be eliminated if there are no uses of x, i.e. if no subsequent computation depends on the value assigned to x. The optimized value of x will then be unavailable. To support debugging with expected behavior, it may not be possible to eliminate $l : x \leftarrow e$ if, at some inspection point, some variable's unoptimized value depends on the optimized value of x. A simple (but far from optimal) solution is to treat all occurrences of variables in the recovery functions as uses of those variables. This strategy prevents any assignment statement that reaches an inspection point from being eliminated.

A vast improvement is achieved by distinguishing between the two types of uses: *debugging uses*, which are those that occur in the recovery functions at inspection points, and *real uses*, which are those that occur in the rest of the program. Using this strategy, an assignment $l : x \leftarrow e$ is a candidate for elimination if the only uses of x are debugging uses. In this case, the assignment is "dead" in the traditional sense, but some variable's unoptimized value at an inspection point depends on the optimized value of x. The assignment l can be eliminated only if all debugging uses of x can be eliminated by replacing the debugging uses with a recovery function whose value is equal to the value of e at l. This is vacuously true if there are no debugging uses of x. This case occurs when the assignment does not reach any inspection point. Also, most variables are temporary variables that are created by the compiler, and these variables initially do not have any debugging uses because the "recover" function only recovers the values of source-level variables.

The predicate $can_recover_p(l : x \leftarrow e\ )$ indicates whether the value of e at l could be determined if the assignment $l : x \leftarrow e$ in procedure p were eliminated, and is defined as follows:

$$can_recover_p(l : x \leftarrow e\ ) \equiv \begin{cases} false & \text{if } x \text{ has one or more debugging uses in } p \text{ and} \\ & l \text{ is a } \phi\text{-assignment or an ambiguous assignment} \\ true & \text{otherwise} \end{cases}$$

If there are no debugging uses of x then the assignment l does not reach any inspection point, so no variable's unoptimized value depends upon the optimized value of x. If l is a ϕ-assignment or an ambiguous assignment then not enough is known about the value being assigned to recover the value (recall that ϕ-assignments correspond to multiple reaching definitions). If l is neither a ϕ-assignment nor an ambiguous assignment then the right-hand-side of the assignment must be either a constant, another variable, or an arithmetic expression involving variables and constants. The value of a constant can always be recovered, since it never changes. Likewise, since every variable is assigned exactly once, a variable's value remains constant after it is assigned. The value of an expression involving variables and constants can be recovered by recovering the values of the operands and applying the operation to these values. This is a good example of how SSA form simplifies the algorithms for performing optimizations.

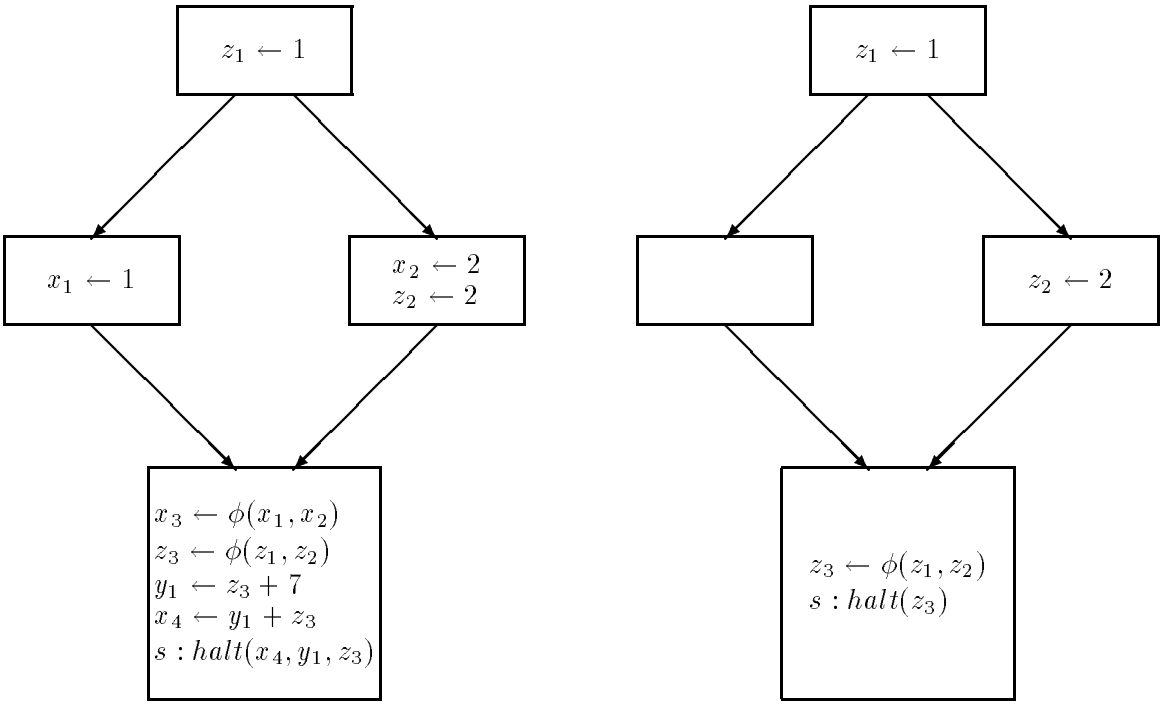

Figure 5: Eliminating dead assignments

Let $inspect_p = s_1 \ldots s_n s_{n+1} \ldots s_m$ where $s_1 \ldots s_n$ are the inspection points in p that have debugging uses of x. Eliminating $l : x \leftarrow e$ results in procedure p' where

$$p' \equiv \langle \{s'_1 \ldots s'_n s_{n+1} \ldots s_m\}, exec_to_inspect_p, source_to_exec_p, vars_p \rangle$$

Inspection points $s_1 \ldots s_n$ have been replaced by $s'_1 \ldots s'_n$, where, for $i = 1 \ldots n$, $trace_{s'_i} \equiv trace_{s_i}$ and $recover_{s'_i}$ is $recover_{s_i}$ with debugging uses of x replaced by a recovery function whose value at s'_i in p' is equal to the value of e at l in p.

$$recover_{s'_i}(v) \equiv \lambda ar.recover_{s_i}(v)(update(ar, x, eval(e, ar)))$$

where $eval(e, ar)$ provides the value of e in activation record ar. Since each variable in a program in SSA form is assigned once, eliminating an assignment eliminates a variable. Thus, ar does not contain a binding for x and $update(ar, x, eval(e, ar))$ is creating a new activation record that is just like ar except that it also has a binding for x. This has the same effect as replacing each occurrence of x in the definition of $recover_{s_i}$ with e.

The following example illustrates the algorithm for eliminating dead assignments. Consider the program shown in Figure 5(a). Initially the function $recover_s$ is:

$$\begin{aligned} recover_s(x) = &\quad \lambda ar.eval(x_4, ar) \\ recover_s(y) = &\quad \lambda ar.eval(y_1, ar) \\ recover_s(z) = &\quad \lambda ar.eval(z_3, ar) \end{aligned}$$

Dead assignments are eliminated as follows:

1. The assignment $x_4 \leftarrow y_1 + z_3$ is a candidate for elimination, since there are no real uses of x_4. The assigned value can be recovered, so the assignment is eliminated, and all occurrences of x_4 in the recovery functions are replaced by $y_1 + z_3$. The modified "recover" function is shown below:

$$\begin{aligned} recover_s(x) = &\quad \lambda ar.eval(y_1 + z_3, ar) \\ recover_s(y) = &\quad \lambda ar.eval(y_1, ar) \\ recover_s(z) = &\quad \lambda ar.eval(z_3, ar) \end{aligned}$$

2. The assignment $x_3 \leftarrow \phi(x_1, x_2)$ can be eliminated because there are no real uses or debugging uses of x_3. This in turn allows $x_1 \leftarrow 1$ and $x_2 \leftarrow 2$ to be eliminated.

3. The assignment $y_1 \leftarrow z_3 + 7$ is now a candidate for elimination since the eliminated assignment to x_4 had the only real use of y_1. The assignment is eliminated, and all occurrences of y_1 in the recovery functions are replaced by $(z_3 + 7)$.

$$\begin{aligned} recover_s(x) = &\quad \lambda ar.eval((z_3 + 7) + z_3, ar) \\ recover_s(y) = &\quad \lambda ar.eval(z_3 + 7, ar) \\ recover_s(z) = &\quad \lambda ar.eval(z_3, ar) \end{aligned}$$

4. The assignment $z_3 \leftarrow \phi(z_1, z_2)$ is now a candidate for elimination, since z_3 has no real uses. This assignment cannot be eliminated because it is a ϕ-assignment and z_3 has a debugging use at s. The final program is shown in Figure 5(b).

A more powerful recovery strategy might attempt to recover the value assigned by a ϕ-assignment by determining which conditions affect control flow and recovering the values of these conditions to determine which variable's value is assigned to the destination of the ϕ-assignment.

Hennessey [Hen82] discusses how to provide truthful debugger behavior in the presence of this optimization, using a more traditional representation for the dependency information.

4.3.5 Register Allocation

Storing a variable in a register makes the variable's (optimized) value unavailable outside the variable's live range. Debugging uses extend the live ranges of variables, so debugging uses must be considered during register allocation. This results in the live range of a variable encompassing both its real uses and its debugging uses, making the variable's optimized value available wherever it is needed either during execution or for debugging. Unfortunately, extending the live ranges of variables that are allocated to registers causes more register spills.

Competition for physical registers can be reduced by "optimizing" the recovery functions so that the unoptimized values of variables depend on the optimized values of as few variables as possible. Consider a debugging use of the variable x at inspection point s. If this debugging use were eliminated then x might still be live at s due to another (real or

debugging) use of x at some point in the procedure after s. In this case, the debugging use of x at s has no effect on the live range of x, so nothing would be gained by eliminating the debugging use. On the other hand, if there were no uses of x after s then eliminating the debugging use of x at s would "shrink" the live range of x, thereby reducing the competition for physical registers, resulting in potentially fewer register spills.

Let $\{x_1, \ldots, x_n, y_1, \ldots y_m\}$ be the variables that have debugging uses at s. Further, assume that none of $\{x_1, \ldots, x_n\}$ have uses after s and that all of $\{y_1, \ldots, y_m\}$ have uses after s. That is, each of the x_i's would be dead at s if its debugging use were eliminated, but each of the y_i's would still be live. Let $\{z_1, \ldots, z_k\}$ be the variables that are live at s (due to uses after s) but that do not have debugging uses at s. Given this, the set of live variables at s is $\{x_1, \ldots, x_n, y_1, \ldots, y_m, z_1, \ldots, z_k\}$. The debugging uses $\{x_1, \ldots, x_n\}$ are candidates for elimination, since these variables would be dead at s were it not for the debugging uses.

A debugging use of a variable is eliminated by replacing all its occurrences with a recovery function of the same value. A debugging use can be eliminated as long as a recovery function of equal value can be computed such that all the variables that it accesses are live at the inspection point. The goal is to reduce the number of live variables at a inspection point. Expression e is cheaper than variable x at inspection point s if x would be dead at s were it not for the debugging use and if all variables that occur in e are live at s. The predicate $can_eliminate_use_p(x, s)$ indicates whether the debugging use of the variable x at inspection point s in procedure p can be eliminated.

$can_eliminate_use_p(x, s) \equiv$

$true$	if $can_recover_p(l : x \leftarrow e\ ) = true$, where $l : x \leftarrow e$ is the (only) definition of x, and every variable in e whose optimized value is accessed by $recover_s$ is live at s.
$false$	otherwise

If $can_eliminate_use_p(x, s) = true$ then the debugging use of x at s in p is eliminated, resulting in procedure p', where

$$p' \equiv \langle inspect_{p'}, exec_to_inspect_p, source_to_exec_p, vars_p \rangle$$

The set $inspect_{p'}$ consists of $inspect_p$ with s replaced by s', i.e. $inspect_{p'} \equiv (inspect_p - \{s\}) \cup \{s'\}$. The inspection point s' is s with the "recover" function updated by replacing occurrences of x in the recovery functions with the recovered value, i.e. $trace_{s'} = trace_s$ and

$$recover_{s'}(v) \equiv \lambda ar.recover_s(v)(update(ar, x, eval(e, ar)))$$

To summarize, a debugging use of a variable x at an inspection point s is a candidate for elimination if eliminating the debugging use of x would make x dead at s. A debugging use can be eliminated if a recovery function of equal value can be computed and all variables that occur in this recovery function are live at s. To eliminate the debugging use of x at s, all occurrences of x in the recovery functions at s are replaced by an expression whose value is equal to the value of x. This "shrinks" the live range of x, thereby reducing competition for physical registers.

The following example illustrates the advantages of eliminating debugging uses prior to performing register allocation. Consider the program fragment shown below:

$$
\begin{array}{rl}
 & x \leftarrow y + z \\
 & \vdots \\
s: & halt \\
 & \vdots
\end{array}
$$

Suppose $recover_s(x) = \lambda ar.eval(x, ar)$, $recover_s(y) = \lambda ar.eval(y, ar)$, and $recover_s(z) = \lambda ar.eval(z, ar)$. Also, suppose that y and z have uses after s, but x would be dead at s were it not for the debugging use. The debugging use of x at s (in $recover_s(x)$) is replaced by the expression $(y + z)$. As a result, x is no longer live at s.

As was described earlier, activation records hide register allocation and whether or not registers are always spilled to the same offset within an activation record. Register allocation is addressed by Coutant, et. al. [CMR88]. They also make use of recovery functions, though not as extensively as we do. However, the flavor of their work is quite different from ours, since they have no notion of an inspection point.

4.3.6 Code Motion of Loop-Invariant Computations

Heintz gives an algorithm for performing code motion in SSA form [Hei90]. It is a variation of an algorithm that uses a precursor of SSA [CLZ86]. Heintz's algorithm assumes that there are no dead assignments in the program on which the code motion is performed. An assignment $l : x \leftarrow e$ is *loop-invariant* if every variable that occurs in e is defined outside of the loop. A loop invariant assignment $l : x \leftarrow e$ can be moved to the loop preheader if the following conditions hold:

1. The block containing l dominates all loop exit nodes. This ensures that l will be executed at least once while control is in the loop.

2. x is not used in any ϕ-assignment in the loop.

If there is an inspection point s in the loop that precedes $l : x \leftarrow e$ then the assignment cannot be moved out of the loop, because doing so would prevent the debugger from reporting the unoptimized value of x at s during the first iteration of the loop. This optimization does not need to be modified to allow debugging with expected behavior, because debugging uses can inhibit code motion. The following example illustrates this point. Consider the program shown in Figure 6(a). Converting the program to SSA form yields the program in Figure 6(b). The assignment $x_3 \leftarrow \phi(x_1, x_2)$ is dead, so it is removed. The assignment $x_2 \leftarrow 1$ can then be moved out of the loop. The resulting program is shown in Figure 6(c).

Suppose the program in Figure 6(a) had a inspection point before the assignment to x in the loop with a debugging use of x. This program is shown in Figure 7(a). Converting the program to SSA form yields the program in Figure 7(b). The assignment to x_3 cannot be removed because of the debugging use of x_3 at s. As a result, the loop-invariant assignment $x_2 \leftarrow 1$ cannot be moved out of the loop because x_2 is used in the ϕ-assignment, violating condition 2 above.

Wall, Srivastava and Templin point out that problems debugging loop-invariant code motion are eliminated by unrolling a loop once.[WST85]. Of course, loops without inspection points do not need to be unrolled to provide expected behavior.

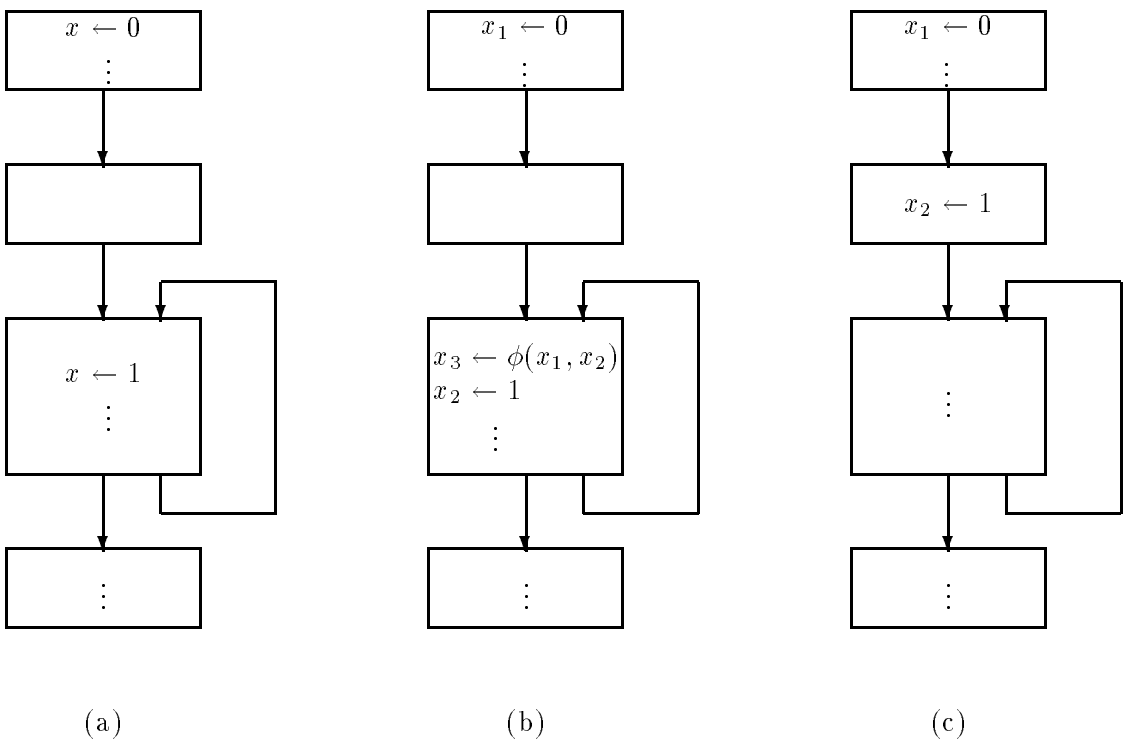

Figure 6: Performing code motion in SSA form

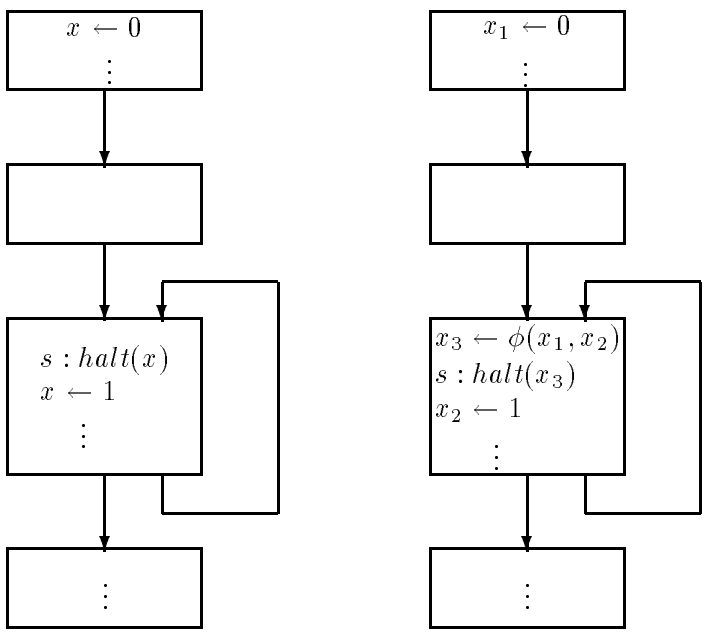

Figure 7: An inspection point inhibits code motion

4.3.7 Induction Variable Elimination and Strength Reduction

This optimization involves replacing loop variables that form arithmetic progressions with different variables and cheaper computations. New variables are added, but these have no effect on the source-level program state because they are not visible to the user. The only part of this optimization that affects the source-level state is the elimination of the dead assignments that are created as a result of replacing one variable by another. An assignment that becomes dead during induction variable elimination can be treated just like any other dead assignment. However, it is possible to eliminate more dead assignments by using a recovery strategy that makes use of the knowledge of *why* the assignment became dead, in particular, the knowledge that the assignment is dead because its destination was replaced in the loop by a different variable. Such an algorithm is presented in [CMR88]. When a multiplication is strength-reduced into successive additions, the location of the replacing temporary and the value or location of the multiplication factor are saved and can be used by the debugger to recover the unoptimized value of the eliminated induction variable within the loop.

4.3.8 Tail Recursion Elimination

Tail recursion elimination causes several problems for debugging. When a tail-recursive call is eliminated and replaced by a jump to the beginning of the procedure, an activation record will exist for only the most recent invocation of the procedure. No information is available to indicate how many recursive calls to this procedure should be reported in the procedure traceback. This problem can be solved by adding a loop counter to indicate how many times the procedure has been invoked. A more serious problem is that previous values of the procedure's local variables and parameters are lost. In general, these values cannot be recovered unless they are explicitly saved on the stack each time through the loop. This additional overhead negates much of the benefit derived from eliminating the recursive call.

There are special cases where local variables and parameter values for previous invocations can be recovered without being explicitly saved on the stack. For example, while loops in Scheme and Smalltalk are implemented by tail recursive procedures whose only argument is a closure, and the closure never changes. Thus, the values of the parameters do not change for the recursive call.

Parameters and local variables can also be recovered if they are induction variables. For example, if the value $x+c$, for some constant c is passed as the value of the formal parameter x, the value of x for a previous invocation of the procedure can easily be computed given the value of x for the most recent invocation and the number of contexts between the context for the most recent invocation and the context for the earlier invocation that is under consideration. Of course, these difficulties arise only if there are inspection points in the procedure on which the tail recursion elimination is performed, since otherwise no recovery information is needed.

4.4 Proofs of Correctness

4.4.1 Criteria for Correctness

Traditionally, an optimization performed on a procedure is considered correct if it preserves the meaning of the procedure [ASU86]. A meaning function $\mathcal{M}(p) : VAL^* \rightsquigarrow VAL^*$ that maps inputs to outputs is associated with every procedure p [LS87]. This is a partial

function because it is undefined for inputs for which execution of the procedure will not terminate. An optimization $\mathcal{T} : P \rightarrow P$ is a function, that, when applied to a procedure, returns a transformed procedure. The optimization $\mathcal{T}$ is correct if, for every procedure p,

$$\mathcal{M}(p) = \mathcal{M}(\mathcal{T}(p)).$$

An optimization must meet additional criteria to allow debugging with expected behavior. In particular, there is the additional constraint that the recovery information is computed correctly, i.e. that the source-level program state can be recovered (correctly) from the low-level state at every inspection point.

Suppose that every procedure p has a function $\mathcal{C}(p) : VAL^* \rightsquigarrow (C^+)^*$ that maps inputs to the sequence of contexts that is created using the recovery information at each point where execution halts. Suppose that when procedure p is invoked with the list of input values $\vec{b} \in VAL^*$, execution will halt k times ($k \geq 0$) before p returns. For $i = 1 \ldots k$, let $\vec{c_i}$ be the sequence of contexts that is created using the recovery information when execution halts for the ith time. The function $\mathcal{C}(p)$ returns the k sequences induced by $\vec{b}$, i.e.,

$$\mathcal{C}(p)(\vec{b}) \equiv \vec{c_1} \ldots \vec{c_k}$$

To be considered correct, an optimization must preserve both the meaning of a procedure and the representation of the source-level state that is created when execution halts. That is, an optimization $\mathcal{T}$ is correct if for every procedure p,

$$\mathcal{M}(p) = \mathcal{M}(\mathcal{T}(p)) \wedge \mathcal{C}(p) = \mathcal{C}(\mathcal{T}(p)).$$

Many optimizations change the low-level execution state at intermediate points. An optimization that alters the correspondence between the optimized and unoptimized states at an inspection point must also change the recovery information so that the source-level state that is computed using the recovery information remains the same. That is, the following invariant must be maintained by all optimizations:

> The recovery information correctly maps the low-level state to the source level whenever a halt is executed.

Every optimization discussed in this section performs the same code transformations as the traditional form of the optimization and therefore preserves the meaning of every procedure. The optimizations are applied in a restricted set of cases and are augmented to compute recovery information. The recovery information will be proven correct by first proving that the initial values given to its various components are correct. It will then be shown that each optimization that modifies the recovery information preserves the correctness of the recovery information.

4.4.2 Correctness of Initial Values

The source-level program state consists of the following components:

- Variable values
- Current execution locations
- The procedure traceback

The recovery information for an unoptimized procedure is correct, if:

1. It reflects the fact that, for an invocation of the unoptimized procedure, every variable's unoptimized value is equal to its optimized value, and

2. It reflects the one-to-one correspondence between activation records and contexts

Theorem 1 *The initial values given to the components of the recovery information are correct.*

Proof: For an unoptimized procedure p, for every inspection place $s \in inspect_p$,

- For every variable $x \in vars_p$, $recover_s(x) \equiv \lambda ar.eval(ar, x)$, which is the optimized value of x.
- $trace_s$ is a list containing the single pair $\langle p, l \rangle$ where l is the source statement in p that corresponds to s. Since one context is created for each element in the list, if execution halts at s, a single context c will be created corresponding to the invocation of p, where $proc_c = p$ and $stmt_c = l$. □

4.4.3 Correctness of Recovery Information is Preserved by Optimization

We have shown that the recovery information for an unoptimized procedure computes a correct source-level representation of the state of an invocation of the procedure whenever a halt is executed. We now show that the correctness of the recovery information is preserved by the various optimizations. This is shown by proving that each occurrence of an optimization preserves correctness. For $n \geq 0$, p_n denotes a procedure on which n occurrences of the optimization have been performed. Notice that the recovery information for p_0, which is the procedure before performing the optimization, is correct by Theorem 1. We show that if an occurrence of an optimization is performed on a procedure p_n, $n \geq 0$, that has correct recovery information, then the recovery information for the resulting procedure p_{n+1} is correct.

4.4.3.1 Inline Procedure Expansion

Performing this optimization alters the correspondence between activation records and contexts by eliminating the activation record for the call that was expanded inline. Therefore, we need to show that when a call is expanded inline, the lists of $\langle$*procedure, source statement*$\rangle$ pairs are updated so that a context will be created corresponding to the eliminated call. Let p_n be a procedure on which n occurrences of inline procedure expansion have been performed. Suppose that when execution halts for the jth time for an invocation of p_n with inputs $\vec{b}$, the low-level state consists of the sequence of activation records

$$\vec{a} = a_1 \ldots a_k$$

For $i = 1 \ldots k$, let $q_i = proc_{a_i}$ and $s_i = exec_to_inspect_{proc_{a_i}}(pc_{a_i})$, i.e. q_i is the procedure that is executing in the activation record, and s_i is the current inspection point in the procedure. We denote $\vec{a}$ by the list of pairs

$$\langle q_1, s_1 \rangle \langle q_2, s_2 \rangle \langle q_3, s_3 \rangle \ldots \langle q_k, s_k \rangle.$$

Notice that $q_1 = p_n$. If the call to q_2 at s_1 in q_1 is expanded inline, resulting in procedure $q_1' = p_{n+1}$, then since there will now be no activation record for q_2 (and no call to q_2 in q_1 and hence no inspection point s_1), the low-level state when execution of p_{n+1} with inputs $\vec{b}$ halts for the jth time is represented by

$$\langle q_1', s_2' \rangle \langle q_3, s_3 \rangle \ldots \langle q_k, s_k \rangle.$$

Theorem 2 *When inline procedure expansion is performed as described above, the sequence of contexts that is created when execution halts for the jth time is preserved, i.e.*

$$\text{trace}_{s_1} \cdot \text{trace}_{s_2} \cdot \text{trace}_{s_3} \cdot \ldots \cdot \text{trace}_{s_k} = \text{trace}_{s_2'} \cdot \text{trace}_{s_3} \cdot \ldots \cdot \text{trace}_{s_k}.$$

Proof: This is easily seen by noting that, when the call to q_2 at s_1 is substituted inline, inspection point s_2' is added to q_1', where $trace_{s_2'} = trace_{s_1} \cdot trace_{s_2}$. □

4.4.3.2 Global Dead-Store Elimination

This optimization has no effect on the procedure traceback. Hence, we only need to show that the functions for recovering unoptimized variable values are modified correctly.

Theorem 3 *Let p_n be a procedure on which n assignments have been eliminated, where $inspect_{p_n} = \{s_{n,1}, \ldots s_{n,k}\}$. If the assignment $l : x \leftarrow e$ is eliminated, resulting in procedure p_{n+1}, with $inspect_{p_{n+1}} = \{s_{n+1,1}, \ldots, s_{n+1,k}\}$, then for $i = 1 \ldots k$, for all $v \in dom(\text{recover}_{s_{n,i}})$, $\text{recover}_{s_{n,i}}(v)$ and $\text{recover}_{s_{n+1,i}}(v)$ are equivalent.*

Proof: Since the only difference between p_n and p_{n+1} is that l is eliminated, the only difference between activation records at $s_{n,k}$ and $s_{n+1,k}$ is that variable x has no value. By definition, if $l : x \leftarrow e$ is eliminated then

$$recover_{s_{n+1,i}}(v) = \lambda ar.recover_{s_{n,i}}(v)(update(ar, x, eval(e, ar)))$$

Because of SSA form, e does not contain references to x and none of the variables occurring in e changes between l and $s_{n,i}$. Thus, e has the same value as x at $s_{n,i}$ in p_n, and eliminating l has no effect on the value of e. Therefore $update(ar, x, eval(e, ar))$, where ar is the activation record at $s_{n+1,k}$, produces the same activation record as the one at $s_{n,i}$, and $recover_{s_{n,i}}$ and $recover_{s_{n+1,i}}$ are equivalent. □

4.4.3.3 Eliminating Debugging Uses

Recall that recovery functions are "optimized" by eliminating debugging uses of variables that would be dead were it not for these debugging uses. The effects of eliminating a debugging use are similar to the effects of eliminating a dead assignment — both make the optimized value of some variable unavailable, so references to this variable in the recovery functions must be replaced by an equivalent recovery function. A difference is that eliminating a debugging use affects the recovery forms at a single inspection point, whereas eliminating a dead assignment affects the recovery functions at all inspection points that contain debugging uses of the destination.

Theorem 4 *Let p_n be a procedure on which n debugging uses have been eliminated, where $inspect_{p_n} = \{s_{n,1}, \ldots s_{n,k}\}$. If a debugging use of the variable x is eliminated at $s_{n,j}$, $1 \leq j \leq k$, resulting in procedure p_{n+1}, with $inspect_{p_{n+1}} = \{s_{n+1,1}, \ldots, s_{n+1,k}\}$, then for all $v \in dom(\text{recover}_{s_{n,j}})$, $\text{recover}_{s_{n,j}}(v)$ and $\text{recover}_{s_{n+1,j}}(v)$ are equivalent.*

Proof: Let $l : x \leftarrow e$ be the (only) assignment to the variable x. By definition, if $recover_{s_{n,j}}(v) = \alpha$ and the debugging use of x at $s_{n,j}$ is eliminated, then

$$recover_{s_{n+1,j}}(v) = \lambda ar.\alpha(update(ar, x, eval(e, ar)))$$

Using the same argument as in the proof of Theorem 3, the recovery function

$$recover_{s_{n+1,j}}(v) = \lambda ar.\alpha(update(ar, x, eval(e, ar)))$$

evaluates to the same value at $s_{n+1,j}$ in p_{n+1} as the recovery function α at $s_{n,j}$ in p_n. □

We have shown that the recovery information is computed correctly for inline procedure expansion, global dead-store elimination, and register allocation. None of the other optimizations that we have discussed have any effect on the recovery information.

5 Evaluation

The techniques that we have presented were used to modify the standard Smalltalk debugger to build a prototype debugger for TS, an optimizing compiler for Typed Smalltalk [JGZ88]. The techniques are general enough to apply to most high-level programming languages, and they will work for debugging commands and optimizations that we have not addressed. However, the techniques require close cooperation between the debugger and the compiler, and therefore are hard to implement in many programming environments. This shows the advantages of an integrated programming environment like that provided by Smalltalk. An integrated programming environment makes it easier to keep track of recovery information and to recompile procedures.

The rest of this section looks at how well our techniques work in practice, presents some extensions to the techniques, and suggests some areas that merit future research.

5.1 Performance

The most important performance criterion is the loss of optimization in production programs caused by providing expected behavior. Production programs have no breakpoints, so the inspection points are only those caused by run-time errors and user interrupts. Most potential run-time errors in TS are array subscript errors. TS is able to infer that no subscript errors will occur in procedures that enumerate over all of the elements of an array. Thus, most of the programs that we compiled with TS were just as fast with debugging enabled as they were with it turned off.

Unfortunately, TS is not very reliable yet, so it can successfully compile only small programs. The program with the largest slowdown is a bubblesort, which has a slowdown of 17 percent. On the other hand, the optimized version of the program is 15 times faster than the only other version of Smalltalk on the same hardware, which is an interpreter. Since the loss of optimization gives the debugger for TS the same behavior as the Smalltalk debugger, the loss of optimization in the debuggable code is not excessive.

Although our primary emphasis is execution speed rather than code size, the two are interrelated. The debuggable code generated by TS for bubblesort is 12 percent larger than the fully optimized code. (One small example was 35 percent larger). Most of this difference is caused by the additional register spills that occur in the debuggable code. A register spill consists of one instruction to write the register to memory and one instruction to restore

the register. Since registers are saved and restored around procedure calls by spilling, and there are generally more live registers in the debuggable code, the additional spills increase the code size. This overhead could be reduced by treating spills around calls as a special case and using the 68020's movem instruction to save or restore any number of registers in a single instruction. However, we haven't added any optimizations to the compiler that are specifically designed to make debuggable code faster or smaller.

A secondary performance issue is the speed of the compiler. Computing the recovery information in TS does not slow down the compiler significantly. In fact, the increase in compilation time is hard to measure, since the time required for garbage collection, which can occur at unpredicatable times, is greater than the extra time required to compute the recovery information. As a result, compiling a program with debugging disabled, i.e. performing full optimization and not computing any recovery information, often takes longer than compiling to support debugging.

Thus, the debuggable code is only a little slower than the fully optimized code and the compiler is not significantly slower when modified to support debugging. Although our implementation of debugging in TS is only a prototype, it indicates that the ideas presented in this paper can work well in practice.

5.2 Extensions and Future Work

5.2.1 Optimizations

The techniques described in this paper are powerful enough to allow a large amount of optimization to be performed on a program. We have considered many different optimizations, and our techniques are amenable to adding other optimizations. The various components of the recovery information can deal with all common location and data problems that are caused by optimization. New optimizations must modify the data structures representing recovery information and ensure that optimization will never prohibit recovering the source-level state at any inspection point. We have not looked at every class of optimization. For example, we have not looked at any optimizations that restructure data to see how these optimizations fit into the framework that we have presented.

There are two ways to increase the amount of optimization further: Develop more powerful recovery strategies, or develop techniques that eliminate more inspection points. Optimizations that eliminate unreachable code have increased importance for debugging. If the compiler can prove that certain run-time errors will never occur then the error checking code and the associated inspection points can be eliminated. Traditional optimizations that find and eliminate unreachable code are useful for eliminating inspection points. TS uses only these optimizations to eliminate inspection points, since we have not yet looked at optimizations whose sole purpose is to eliminate inspection points. The importance of eliminating inspection points is illustrated by our benchmarks. When all subscript checks and the associated inspection points are eliminated by the compiler, the debuggable code is fully optimized. Similar benchmarks for which the compiler cannot eliminate all inspection points are up to 17 percent slower. An area that merits future research is to improve techniques for detecting unreachable code so that even more inspection points can be eliminated.

5.2.2 Converting to Unoptimized Form

State-changing context operations are implemented by first converting the affected part of the program state to unoptimized form. An improvement to this technique is to convert the

state only if the effects of optimization prevent carrying out the state-changing operations directly on the optimized state. Of course, this requires that the compiler compute additional recovery information that would indicate when the effects of optimization prevent carrying out the operations. We now show how this might be done for changing the value of a variable; different techniques are needed for the other state-changing operations.

At every inspection point s, the compiler maintains a set of variables $unchangeable_s$. This set is initially empty, and a variable is added when the compiler determines that the effects of optimization prohibit changing the value of the variable directly in the optimized state at s. We show how this is done for copy propagation; similar techniques could be used for other optimizations. Consider the program fragment shown below:

$$\begin{array}{rl} & x \leftarrow y \\ s: & halt \\ & z \leftarrow x \end{array}$$

Copy propagation can be performed by replacing all uses of x by y. The resulting code is as follows:

$$\begin{array}{rl} & x \leftarrow y \\ s: & halt \\ & z \leftarrow y \end{array}$$

Now, neither x nor y can be changed in the optimized program state at s. A change made to x would not be reflected in the assignment to z. Changing the value of y at s would result in an incorrect value being assigned to z. The cause of these problems is that changing the value of a variable at s invalidates an assumption that was made by the optimizer, namely that the variable was not assigned at s. Assigning to a variable at s changes the definition of the variable that reaches its uses after s. Thus, both x and y must be added to $unchangeable_s$ to indicate that, to change the value of either of these variables at s, the affected part of the program state must first be converted to unoptimized form.

Zelwegger suggested that if an context operation cannot be carried out on the optimized state, it might be possible to determine which optimizations are causing difficulties and to turn off only the offending optimizations when converting the affected state [Zel84]. Adding this capability requires that the compiler compute two additonal kinds of information. First, it needs to tell which optimizations are inhibiting each of the state-changing operations. Second, if only some optimizations are suppressed then the fully optimized state must be mapped to the different partially optimized states that can result from suppressing various combinations of optimizations. This is a very complex problem, and it is not clear that the benefits outweigh the costs, especially considering the fact that the gains in execution speed of the partially optimized code over the unoptimized code last only as long as a converted procedure has an active invocation.

5.2.3 Full Optimization and Truthful Behavior

We have emphasized debugging over optimization—optimization is restricted to allow expected debugger behavior. The techniques that have been presented can also be used to allow full optimization and truthful debugger behavior. Expected behavior is made possible by only applying an optimization in cases where the effects of the optimization do not prevent recovering the program state at some inspection point. If only truthful debugger

behavior is desired, optimizations can be unrestricted, and the compiler can indicate which parts of the program state are made unrecoverable due to optimization. For example, eliminating an assignment to the variable x makes the optimized value of x unavailable at all inspection points that are reached by the eliminated assignment. If the assigned value cannot be recovered, for every inspection point s reached by the eliminated assignment, and for every variable v whose unoptimized value depends on the optimized value of x, $recover_s(v)$ can be marked "unknown" to indicate that the value of v cannot be recovered at s.

Allowing full optimization severly restricts the debugger's ability to carry out commands that change the program state. This is because state-changing context operations are implemented by first converting the affected part of the state to unoptimized form, and converting to unoptimized form requires recovering all of the state contained in the affected activation record. Thus, if any part of the state cannot be recovered, the state-changing context operation cannot be carried out.

The improvement suggested above of converting to unoptimized form only when necessary could be used to increase the places where state-changing context operations could be carried out. Another improvement can be made by noting that dead variables have no effect on subsequent computations. Thus, dead variables can be ignored when converting to unoptimized form. If the part of the affected state that cannot be recovered consists of only dead variables, the state-changing operations can still be carried out.

5.2.4 Reflection

Contexts are part of the Smalltalk-80 programming environment. This helped lead us to the context-oriented model of debugging. The Smalltalk debugger uses contexts heavily, though the Smalltalk-80 contexts are not as abstract and implementation-independent as those described in this paper.

Smalltalk-80 contexts have other uses besides debugging, such as implementing backtracking and exception handling. The techniques presented in this paper are more than just techniques for debugging optimized code, they are techniques for implementing contexts in conjunction with an optimizing compiler. Thus, they could be used for implementing exception handling in the presence of an optimizing compiler.

There are some practical differences between debugging and other uses of contexts. The main difference is that debugger uses are rarely executed and so do not need to be fast. In particular, a debugger does not need recovery functions to be fast. However, a backtracking algorithm may need to be fast, so it might be important to speed up the recovery functions by optimizing the program less.

Contexts are an example of computational reflection; the ability of a program to access and manipulate its runtime representation. Reflection is usually defined from the point of view of an interpreter. This paper shows that certain reflective features are quite compatible with optimizing compilers.

5.3 Conclusion

We have shown that an advanced compiler can minimize the conflict between debugging and optimization. A significant amount of optimization can be performed on a program, while still allowing the program to be debugged at the source level. The debugger for TS contributes greatly to retaining the desirable features of the Smalltalk programming

environment without making large sacrifices in execution speed. This is another step toward raising the level at which the programmer works while retaining execution efficiency.

Acknowledgements

This research was supported by NSF contract CCR-8715752, by the AT&T ISEP grant, and by financial support from Tektronix.

Many thanks to Ken Albin, Max Copperman, Bjorn Freeman-Benson, Sam Kamin and Phil Pfeiffer, who commented on earlier drafts of this paper.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BE76] F.L. Bauer and J. Eickel, editors. *Compiler Construction: an Advanced Course*, pages 549–600. Springer-Verlag, 1976.

[CFR*89] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[CLZ86] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–84, 1986.

[CMR88] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. DOC: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 37–44, June 1988.

[Fei82] Peter H. Feiler. *A Language-Oriented Interactive Programming Environment Based on Compilation Technology*. PhD thesis, Carnegie-Mellon University, 1982.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[Hei90] Rick Heintz. *Low-level Optimizations for an Object-Oriented Programming Language*. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[Hen82] John Hennessey. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88, Object-Oriented Programming, Systems, Languages, and Applications*, pages 18–26, 1988.

[Joh79] Mark Scott Johnson. Translator design to support run-time debugging. *Software-Practice and Experience*, 9:1035–1041, 1979.

[LM69] Edward S. Lowry and C.W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* John Wiley & Sons, 1987.

[PL83] Michael L. Powell and Mark A. Linton. A database model of debugging (preliminary draft). *ACM SIGPLAN Notices*, 18(8):67–70, August 1983.

[Sat75] Edwin H. Satterthwaite. *Source Language Debugging Tools.* Technical Report STAN-CS-75-494, Stanford University Computer Science Department, May 1975.

[ST83] Rich Seidner and Nick Tindall. Interactive debug requirements. *ACM SIGPLAN Notices*, 18(8):9–22, August 1983.

[WJW*75] William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler.* Elsevier North-Holland, Inc., 1975.

[WS78] Henry S. Warren, Jr. and Hans P. Schlaeppi. *Design of the FDS interactive debugging system.* Technical Report RC7214, IBM, July 1978.

[WST85] David Wall, Amitabh Srivastava, and Fred Templin. A note on Hennessy's "symbolic debugging of optimized code". *ACM Transactions on Programming Languages and Systems*, 7(1):176–181, January 1985.

[Zel84] Polle Trescott Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Technical Report CSL-84-5,[P84-00047], Xerox PARC, Palo Alto, California, May 1984.

[Zur89] Lawrence W. Zurawski. *Source-level Debugging of Globally Optimized Code with Expected Behavior.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.