A RETARGETABLE DEBUGGER

Norman Ramsey

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1993

Abstract

Debuggers are specific to the machines, operating systems, and languages that they support. Much of a debugger has to be re-implemented for each new machine, so debuggers that work with a variety of machines and operating systems can get unwieldy. Improvements to debuggers may be lost unless they are re-implemented as users move to new machines. If retargeting debuggers were easier, other improvements would be more valuable. This thesis describes the design and implementation of `ldb`, a prototype retargetable debugger.

Dealing with symbol-table formats is one of the most machine-dependent aspects of debuggers. `ldb` eliminates this machine dependence by using one format on all machines. The format is a language—a dialect of PostScript, which is extensible and can represent procedures. `ldb` reduces retargeting effort associated with variations in run-time support by controlling its target process using a debug nub, which is a small piece of object code linked with the target program.

Much of a debugger's job is to undo what the compiler has done or to do what the compiler could do, but at run time. For example, to print the value of a variable, the debugger must undo the compiler's mapping of source-level data to the machine level. To evaluate an expression, the debugger must check that it is syntactically and semantically correct and translate it into an executable form. `ldb` makes the compiler do as much of this work as possible. The compiler emits PostScript that `ldb` uses to print values. A variant of the compiler runs at debug time and compiles expressions into other PostScript, which `ldb` uses to evaluate the expressions.

Debugging tasks like planting breakpoints and walking the call stack have no analogs in a compiler. `ldb` reduces retargeting effort for these tasks by using layers of abstraction to minimize machine-dependent code, which is confined to the innermost layers.

These techniques produce a debugger with little machine-dependent code. `ldb`'s total code size is about 15,000 lines of Modula-3 and C, but it needs no more than 550 lines of machine-dependent code for any of its 4 targets.

Acknowledgments

It has been a pleasure and an education to work with Dave Hanson. He is a master of careful engineering, and I have learned much from his example. There are only a few like him left.

It is also a pleasure to thank my thesis readers. Anne Rogers helped me extract ideas from a mass of detail and often dropped what she was doing to help me plan the manuscript. Despite her disclaimers she gives excellent advice to grownups. John Ellis reviewed the work with astonishing thoroughness and attention to detail; his many valuable suggestions enabled me to strengthen it throughout. For example, he showed how I could remove 25 lines from the VAX nub, and the formalization of breakpoint commands mentioned in Section 7.7 is his idea. David Redell was kind enough to review several chapters under difficult circumstances; his comments were especially useful because of his substantial experience with debugging.

For many years I have been enriched by David Dobkin's perspective on research and education in computing. He suggested that I work on debugging, but it is not his fault that I took the suggestion seriously. Mary Fernandez has been a most congenial colleague, with whom I have enjoyed many discussions. She wrote the first implementation of the pattern language used in Appendix B, and she reviewed early drafts of several chapters. Mark Weiser and Mike Spreitzer invited me to Xerox PARC, where my ideas matured; I arrived with an artifact and left with an understanding. Mike Spreitzer helped build that understanding by independently moving an early version of `ldb` to the SPARC.

Many people helped make Princeton a fun place to work, and I will miss them all. Extra thanks to Jim Plank, for everything from "granularity" to `jgraph`, and to Norbert Schlenker, who made my office a fine place to be.

`ldb` has been funded by a Fannie and John Hertz Fellowship, an AT&T Bell Laboratories Fellowship, a summer research internship at the Xerox PARC Computer Science Laboratory, and NSF grant number CDA-8619811 A03.

Special thanks to my wife, Cory, without whose love and support I never would have finished. She always treated me like Dr. Jekyll even when I behaved like Mr. Hyde.

Contents

Abstract iii

Acknowledgments v

1 Introduction 1

1.1 Related work . . . 5

1.2 Modula-3 terminology and conventions . . . 7

1.3 Organization . . . 8

2 Debugging With `ldb` 11

3 Abstract Memories 21

3.1 Kinds of abstract memories . . . 23

3.2 Implementing abstract memories . . . 25

3.2.1 Trapped memories . . . 29

3.2.2 Combining abstract memories . . . 30

3.3 Discussion . . . 32

4 PostScript Symbol Tables 35

4.1 Representing debugging information in PostScript . . . 37

4.2 Using PostScript symbol tables . . . 41

4.3 Representing symbols and types in Modula-3 . . . 43

4.4 Printing values . . . 44

4.5 Performance enhancements . . . 48

4.6 Generating PostScript symbol tables . . . 51

4.7 Discussion . . . 53

5 Expression Evaluation 59

5.1 Debugger end of the expression server . . . 61

5.2 Evaluating and printing compiled expressions . . . 62

5.3 Making `lcc` act as an expression server 63
5.3.1 Reconstructing `lcc`'s symbol and type data 63
5.3.2 Delaying assignment of machine-dependent data 68
5.3.3 PostScript back end 69
5.4 Discussion 75

6 The Debug Nub **77**
6.1 Debugger's view of the nub 79
6.2 Implementing the nub protocol 80
6.3 Illustrating the nub protocol 82
6.4 Implementing the nub 84
6.4.1 Procedure call 85
6.4.2 Process context 89
6.5 Discussion 90

7 Breakpoints and Events **95**
7.1 Example 96
7.2 Events 97
7.3 User-level breakpoints 99
7.4 Traps 102
7.5 Low-level breakpoints 104
7.6 Procedure calls 107
7.7 Discussion 108

8 Stack Walking **111**
8.1 Machine-independent layers 112
8.2 Generic layer (`CommonFrame.T`) 116
8.2.1 Preserved registers 119
8.2.2 The frame on top of the stack 123
8.3 Machine-dependent layer 126
8.4 Discussion 129

9 `ldb`'s PostScript **133**
9.1 Implementation 134
9.1.1 Safe, fast lexical analysis 136
9.1.2 Compact specifications for operator implementations 137
9.1.3 Performance tuning 139
9.2 Discussion 139

10 Retargeting `ldb` **143**
10.1 The nub . . . 145
10.2 PostScript . . . 146
10.3 Debugger code: breakpoints and stack walking . . . 147
10.4 The compiler . . . 150
10.5 Discussion . . . 150

11 Evaluation **155**
11.1 PostScript . . . 155
11.2 Strengths . . . 157
11.3 Weaknesses . . . 158
11.4 Comparison with `dbx` and `gdb` . . . 159
11.5 Compatibility and other retargeting costs . . . 161
11.6 Recommendations . . . 163
11.7 Further work . . . 164
11.8 Envoi . . . 166

References **167**

A A Formal Model of Breakpoints **175**
A.1 Modeling the program counter and execution . . . 176
A.2 Counting events . . . 179
A.3 Implementing the breakpoint . . . 180
A.4 Completing the model . . . 185

B MIPS Instruction Specification **187**
B.1 Instruction specification . . . 187
B.2 Computing follow sets . . . 193

List of Tables

1 Debugger operations that need symbol-table information. 42

2 Locations of method definitions and implementations. 113

3 Register classifications used by `ldb`. 116

List of Figures

1 Traditional debugging overview. 3
2 `ldb` overview. 3
3 Example Program 12
4 Summary of `ldb` commands. 20
5 Structure of the abstract memory used on the MIPS. 22
6 Implementation of an abstract memory for one stack frame. 24
7 68020 abstract memory. 32
8 VAX abstract memory. 32
9 Procedure `fib` with stopping points. 36
10 The tree structure of `fib`'s symbol table. 36
11 Use of anchor symbols. 39
12 Top-level dictionary for `fib.c`. 41
13 Linker table for the program `fib`. 41
14 Printing procedure for arrays. 46
15 Building a lazy version of `fib`'s top-level dictionary. 49
16 Types of values associated with key `*CACHE*` in a symbol or type dictionary. 51
17 Communication paths between `ldb` and an expression server. 60
18 Steps in evaluating an expression. 61
19 Code to evaluate `atan2(sin(1.0), cos(1.0))`. 72
20 Code to evaluate `argv != 0 && argv[0]`. 74
21 Layers implementing the debug-nub protocol. 81
22 Messages exchanged during the processing of one event. 83
23 Parts of the debug nub. 84
24 The nub's implementation of procedure call. 86
25 Source file `fib.c` 98
26 User-level breakpoint set 101
27 The layers of the stack-frame abstraction. 112
28 The generic implementation of the `abstractMemory` method. 120

29 `fib`'s stack frame on the MIPS. 121
30 Generic procedure for creating the top frame on the stack. 124
31 The hierarchy of interpreter types. 135
32 The machine-dependent parts of `ldb`. 144
33 The MIPS configuration module. 148
34 The MIPS configuration interface. 149
35 Opcode tables from MIPS architecture manual. 189

Chapter 1

Introduction

Most programmers don't use a debugger (Gramlich 1983). Perhaps they would if debuggers were more reliable, did more, and had better user interfaces. Anyone trying to improve a debugger learns that debuggers are specific to the machines, operating systems, and languages that they support. Much of the debugger has to be re-implemented for each new machine, so debuggers that work with a variety of machines, operating systems, and languages can get unwieldy. For example, GNU's `gdb` (Stallman and Pesch 1991) is 150,000 lines of C, and at least 57,000 of those lines are machine dependent. Because retargeting debuggers is hard, improvements to debuggers may be lost unless they are re-implemented as users move to new machines. If retargeting debuggers were easier, other improvements would be more valuable. This thesis describes techniques for building retargetable debuggers. These techniques are used in the design and implementation of `ldb`, a prototype retargetable debugger.

`ldb` is a source-level debugger like `gdb` or `dbx` (Linton 1990). It can be used with C programs compiled with `lcc` (Fraser and Hanson 1991b), a retargetable compiler that generates code for the MIPS, Motorola 68020, SPARC, and VAX architectures. Like `gdb` and `dbx`, `ldb` lets users set and remove breakpoints, start and stop programs, evaluate expressions, and make assignments to variables. Its total code size is about 15,000 lines, but it needs no more than 550 lines of machine-dependent code for each target.

Retargeting a debugger can be complex because a debugger interacts with several different aspects of its machine, operating system, and programming environment. These can be considered axes along which retargeting might be necessary.

Each compiler chooses some mapping of its source-level data structures to the machine's data structures. In C, these choices are fairly simple, usually limited to the sizes and alignments of the basic types and the layout of bit fields in structures. Other languages present more choices, for example, the representations of open arrays or objects in Modula-3 (Nelson 1991), or the representation of closures in Standard ML (Appel 1990).

A debugger relies on information from the compiler and linker, e.g., the types and locations of variables. Not just the contents but also the format of this information can vary from target to target. This information is not always provided for all procedures in a program; a debugger should function correctly even in its absence.

A debugger must choose a target process to debug. It should be able to choose any stopped or running process, not just processes started in a special "debug mode." It should be able to choose a process on another machine, even a machine of a different architecture from the one the debugger itself runs on. Such features are especially important for debugging long-running servers and for debugging programs on embedded computers, which may not have the screens and keyboards needed to run a debugger. Cross-architecture debugging also makes it possible to develop operating systems and compilers for a new machine while running the debugger on a more reliable machine.

A debugger must have some means of manipulating the target process being debugged. Some systems enable the debugger to run in the same address space as the target; others provide support for debugging from a separate process, or even from another machine. The nature of the support can vary with both the machine and the operating system. The language run-time system provides another source of variation; a debugger may want to work with user-level threads or to intercept exceptions, for example.

A debugger must work with the entire call stack. It must know enough about the calling sequence to infer what the machine state will be when control returns to a procedure in the middle of the stack; for example, it must be able to recover local variables kept in registers. To support expression evaluation, it must be able to call procedures in the target process, which may require more information about the calling sequence.

A debugger should work with programs written in more than one language. Each language has its own rules for resolving names, representing values, evaluating expressions, etc. Support for different languages may require re-implementation in the debugger (Beander 1983).

A debugger must be able to set breakpoints and to step through the execution of a program. Doing so may require information about the target's instruction set, for example to compute control-flow information.

`ldb` solves only some of these problems. It supports different representations of source-level data, different calling sequences, and breakpoints using different instruction sets. Instead of the representations of debugging information used by existing compilers and linkers, `ldb` defines a machine-independent representation, which the `lcc` compiler is modified to emit. Similarly, `ldb` does not use existing operating-system support for debugging, but implements its own, which is retargeted for each machine. `ldb`'s techniques should support more than one programming language, but the prototype works only with ANSI C (ANSI 1990).

Figures 1 and 2 illustrate the retargeting problems related to operating-system support, representation of information from the compiler, and expression evaluation in the source language. Figure 1

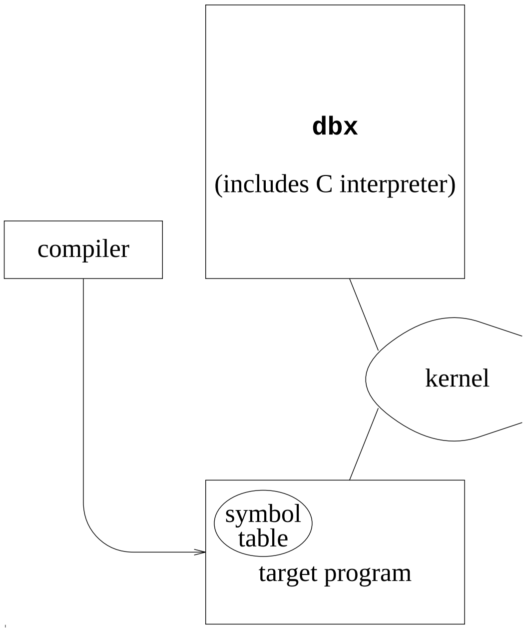


Figure 1: Traditional debugging overview.

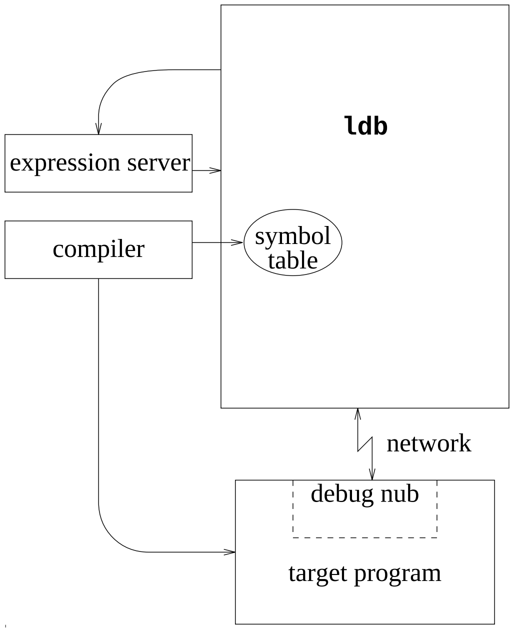


Figure 2: `ldb` overview.

shows a traditional Unix debugger, `dbx`, debugging a target process. `dbx` uses kernel support, typically the `ptrace` system call, to control its target. The meaning and usage of `ptrace` differ among different versions of Unix (Adams and Muchnick 1986). `dbx` gets its symbol-table information from the object file for the target program, where it is stored in a machine-dependent, language-dependent format. Finally, `dbx`'s organization requires two different implementations of C: one in the compiler and one in the debugger.

Figure 2 is the corresponding picture for `ldb`. The kernel interface has been replaced by a network connection to a *debug nub*, which is a small piece of object code linked with the target program. The nub manipulates the target process in response to messages sent from the debugger, as described in Chapter 6. The message protocol is the same on all machines. `ldb` uses a general-purpose programming language, PostScript (Adobe 1985), to represent symbol-table information. This representation is machine independent and language independent. Using PostScript makes it possible to put code, not just data, in the symbol table. `ldb` uses this technique to hide from the debugger the machine-dependent mapping from source-level to machine-level data, as described in Chapter 4. Finally, the debugger itself has no implementation of C; it relies on a standalone implementation that runs in its own address space as an *expression-evaluation server*. This organization makes it possible to re-use the compiler's implementation of C, as described in Chapter 5. Using an expression-evaluation server and code in the symbol table make the debugger proper almost independent of C, which should make it easier to add support for other languages.

Much of a debugger's job is either to undo what the compiler has done or to do what the compiler could do, but at run time. For example, to print the value of a variable, the debugger must map the machine-level data structures back up to the source level. To evaluate an expression, the debugger must check that it is syntactically and semantically correct and turn it into something that can be executed. `ldb` makes the compiler do as much of this kind of work as possible. PostScript provides a means by which the compiler can work for the debugger. The compiler decides how values should be printed, then tells the debugger how to do it by emitting PostScript code for the debugger to interpret. The compiler, transformed into an expression-evaluation server, also does most of the work of expression evaluation, compiling C expressions into PostScript procedures. Some changes are needed to make PostScript work as a language that supports debugging instead of imaging. These changes, and the implementation techniques used to add debugging support, are described in Chapter 9.

Although `ldb` relies on help from the compiler, it does not require major changes to the compiler. Changes are limited by using abstractions like those defined by the code-generation interface between the front and back ends of the compiler (Fraser and Hanson 1991a). `ldb` models the contents of target memory and registers as an *abstract memory*; the debugger uses abstract memories to fetch and store scalar values like those in `lcc`'s code-generation interface. PostScript is flexible enough to be easily matched to `lcc`'s execution model; as described in Chapter 5, a simple postorder traversal of `lcc`'s intermediate code is used to generate PostScript during expression evaluation.

Planting breakpoints and walking the call stack are debugging tasks that have no analogs in a compiler. `ldb` reduces the retargeting effort for these tasks by using layers of abstraction to minimize and isolate machine-dependent code. `ldb` is written in Modula-3 and uses Modula-3 subtyping to define the layers. Machine-dependent code is confined to the leaves of the type hierarchy. The bulk of the implementation is in the methods of the supertypes, which are machine-independent. A similar technique has been used in implementations of I/O streams (Nelson 1991, p. 143). Chapters 7 and 8 describe the hierarchies used to implement breakpoints and stack walking.

The debug nub eliminates the need for a special debug mode; the nub is always linked with the target program, and any target can be debugged. If a target faults unexpectedly, the nub waits for a network connection from the debugger. A user can stop a running target and connect `ldb` to it; the target is stopped by typing the Unix quit character or by using the Unix `kill` command. After the debugging session is over, the debugger can be disconnected and the target resumed as if the interruption had never occurred. If the connection or the machine running the debugger should fail while the debugger is connected, the nub recovers and waits for a connection from another instance of the debugger.

The targets of retargetable tools are often chosen when the tools are built. For example, `lcc` can cross-compile, but the target architecture is fixed when `lcc` is compiled, not when a user's program is compiled. `ldb`, by contrast, does not fix a target architecture until the user selects a target program. `ldb` stores its machine-dependent code and data in Modula-3 objects; an architecture is fixed by choosing one such object, not by conditional compilation. The choice is indifferent to the architecture that `ldb` itself runs on; `ldb` can run on one kind of machine while debugging a target program running on another kind of machine. `ldb` treats all debugging as cross-architecture debugging; it uses no extra mechanism when debugging across architectures.

1.1 Related work

Strategies for getting information from compilers and linkers vary. In some experimental systems, the compiler and debugger are tightly coupled. DEC SRC's Vulcan debugger executes in the same address space as the compiler, sharing its annotated abstract syntax trees. The DICE debugger cooperates with an incremental compiler as part of an integrated environment (Fritzson 1983). Even if the compiler and debugger are separate, the debugger may run in the same address space as the linker, sharing its data structures (Aral, Gertner, and Schaffer 1989). Other debuggers, like VAX DEBUG (Beander 1983) and `dbx` (Linton 1990), are completely separate tools that communicate with both compiler and linker only through symbol-table information. This information is placed in the object file in a machine-dependent format. `ldb` is like these debuggers, but it eliminates retargeting effort by its use of PostScript symbol tables.

`gdb` 4.0 (Stallman and Pesch 1991) supports 20 different target machines and many different versions of Unix, but of its more than 150,000 lines, over 47,000 are noted in the documentation as machine-dependent. Another 10,000 lines deal with machine-dependent object code formats like `a.out` and COFF. The long delay between the availability of MIPS machines and the availability of `gdb` for them also suggests that substantial effort is needed to retarget `gdb`.

`dbx` has been ported to seven target machines, but no single version is used everywhere; different proprietary versions with different behavior are used on different machines.

`ldb`'s abstract memories resemble hardware. By contrast, the Cedar debugger uses a language-level approach; it manipulates a "Cedar abstract machine," whose basic operations resemble those of the Cedar programming language (Swinehart *et al.* 1986, Section 6.4). The abstract machine's interface to Cedar data is used by programs other than the debugger, including a user-interface tool that lets users manipulate fields (including procedures) of records chosen dynamically. A similar abstraction could be built on top of `ldb`'s PostScript symbol tables and abstract memories, but it, like the expression server, would need detailed knowledge of the contents of the symbol tables; it would not be isolated from the details of C.

`ldb`'s nub interface is derived from the Topaz TeleDebug protocol (Redell 1989). Topaz provides completely reliable, available debugging support; Section 6.5 compares it and `ldb`. Other approaches to debugging support include system calls (Adams and Muchnick 1986), as depicted in Figure 1, controlling processes through the file system (Killian 1984), and running the target and debugger in the same address space (Aral, Gertner, and Schaffer 1989). The relative merits of these approaches are discussed in Chapter 6.

Work on the performance of debuggers for sequential programs has focussed on breakpoints. Two kinds of breakpoint implementations are in common use. One is based on traps and single stepping (Caswell and Black 1990), one on patching the code of the target program (Digital 1975). Kessler (1990) describes a fast implementation of code breakpoints; Wahbe (1992) describes simulations of four implementations of data breakpoints. Both authors suggest that the best performance is obtained by code patching. Performance is more critical in debugging parallel programs, because perturbing the execution of a single processor may change the behavior of an entire program. In a parallel-programming environment, debugging work can be offloaded onto a second processor (Aral, Gertner, and Schaffer 1989). Alternatively, monitoring and logging can be done by a special-purpose coprocessor (Gorlick 1991). All these approaches use either branch or coprocessor instructions to transfer control from target to debugging code without kernel intervention, avoiding the overhead of trap handling and context switching. The authors with working implementations describe performance improvements of three orders of magnitude.

Much current research in debugging investigates extensions beyond the kind of basic debugging features provided by `ldb`. Such extensions include better user interfaces, event-based debugging, debugging optimized code, time-travel or replay debugging, and debugging parallel or distributed programs.

Some user-interface work focuses on the use of mouse and bitmapped display, for example, to make it possible to plant and remove breakpoints by direct manipulation of the source code of the target program (Cargill 1983; Cargill 1986; Russell 1992). Other possibilities are to use the display to draw pictures of the data structures in the target program (Myers 1980) or to visualize the execution of the target program (Miller and McDowell 1991, Session 2). Work not related to displays has considered programmable user interfaces, simplifying the manipulation and exploration of target events (Olsson, Crawford, and Ho 1991) and of target data (Golan and Hanson 1993).

The problem of debugging optimized code has been addressed from several viewpoints. One holds that, when possible, the debugger should make it appear to the user of the debugger that the computer is executing the original, unoptimized program, a property called "expected behavior." When the debugger cannot provide expected behavior, it fails to carry out the user's debugging request. Hennessy (1982) discusses the problem of recovering the values of variables as they would be in the original program. He discusses several local and global optimizations, giving algorithms that can usually identify variables whose values do not correspond to those in the original program. They can sometimes recover the values of such variables. Zellweger (1984) describes the effect of several optimizations and presents an implementation that can provide expected behavior in the presence of cross-jumping and procedure inlining, optimizations that merge or duplicate object-code sequences associated with particular source-code sequences. Zurawski and Johnson (1991) combine a context-oriented debugging technique with restrictions on optimization to guarantee expected behavior throughout a debugging session; users cannot tell that they are debugging optimized programs. Brooks, Hansen, and Simmons (1992) take the opposite approach. Instead of making it appear that the computer is executing the original program, their debugger shows the actual behavior of the target program, including the effects of optimization. The debugger highlights the source being executed at several levels of granularity, including expressions, statements, blocks, loops, and functions.

Time-travel debugging enables the user to explore the state of the target program at any point in its execution. Time travel implies that users can move from later to earlier states, a feature sometimes called "replay" or "reverse execution." One implementation technique is to use source-code transformation to make the program log state changes. An early debugger based on this technique ran the program to completion, then worked with the log to give the illusion of interactive debugging with time travel (Balzer 1969). Logging cost can be reduced by taking periodic checkpoints; time travel is implemented by executing forward from a previous checkpoint. The compiler may transform the source to generate appropriate checkpointing code (Tolmach and Appel 1990), or the checkpointing may be done by the run-time system, using virtual memory operations (Feldman and Brown 1988). In addition to checkpointing internal state, the program may have to log significant external events like input.

Recent work on parallel and distributed debugging emphasizes program visualization and event matching as well as time travel (Miller and LeBlanc 1988; Miller and McDowell 1991). Event-based techniques (Bates and Wileden 1983; Bruegge 1985; Olsson, Crawford, and Ho 1991) seem particularly well suited for implementation above `ldb`. `ldb` can debug on multiple architectures simultaneously, so it can process events from pieces of client-server applications that execute on different hardware.

1.2 Modula-3 terminology and conventions

`ldb`'s implementation uses standard Modula-3 conventions, and this thesis uses Modula-3 terminology to refer to the implementation. This section explains the most important terms and conventions and describes how the code shown in the thesis has been edited for clarity.

A Modula-3 interface that provides a certain abstraction, e.g., abstract memory, "exports" that abstraction. `ldb` follows the Modula-3 convention of naming each interface after the abstraction exported by that interface. When the abstraction is a type, the name chosen for the type is `T`. Thus, the abstract memory is exported by the `Memory` interface, and the full name of its type is `Memory.T`. Other parts of the program that use `Memory.T` import the `Memory` interface; they are its clients. Clients are different from users; a client is part of a program.

Modula-3 is an object-oriented language; objects have fields and methods, and subtypes inherit those fields and methods from their supertypes. Modula-3 provides no discriminated-union type; unions are implemented using subtyping, and explicit dynamic type checking (`TYPECASE`) is used to distinguish variants. `ldb` uses this technique in its PostScript interpreter to distinguish different kinds of PostScript values.

Modula-3 provides concurrency in the form of multiple threads. Threads are synchronized through the use of mutexes and condition variables (Birrell 1991). Lampson and Redell (1980) describe a similar model of concurrency. When connected to a target program, `ldb` uses two threads; one responds to commands typed by the user, and one responds to events delivered from the debug nub.

Modula-3 procedures typically indicate errors by raising exceptions, not by returning special error values. `ldb` uses exceptions to signal mistakes on the user's part, like misspelled variable names, or failures, like losing the connection to the debug nub. Exceptions are usually handled by the user interface, which prints messages describing the problem.

In Modula-3, it is common to define types with methods that are not implemented, for example, `Thread.Closure` (Birrell 1991, Section 4.3). Correct programs do not create instances of such types; they create instances of proper subtypes, which implement the methods. There is no standard term used to refer to the supertypes that are used only to define subtypes, not to create objects. When necessary, I refer to such supertypes as abstract supertypes.

Modula-3 code shown in this thesis differs from the code used in the implementation; brands and `RAISES` clauses are omitted, and qualified names are used almost exclusively. Qualified names are used even in the definitions of types and procedures, where such use is not legal Modula-3. For example, the definition of `Memory.T` is shown in the form "`TYPE Memory.T = ...`," not in the legal form "`TYPE T = ...`"

1.3 Organization

Except for the first, second, and last chapters, every chapter in this thesis begins with a few pages of introductory material and ends with a discussion. Reading only the introductory part of each chapter gives an overview of the entire work; the discussions present the results.

Chapter 2 presents a contrived example of a debugging session using `ldb`. This session illustrates both the capabilities of `ldb` and the problems that it must solve. Examples in later chapters refer both to the session and to the example program.

Abstract memories are used throughout `ldb`, and it is best to read the first part of Chapter 3 before tackling later chapters. The second part, the code, is needed only to understand some of the detailed examples that appear in later chapters.

The reader interested in `ldb`'s interaction with the compiler is directed to Chapters 4 and 5, which describe PostScript symbol tables and the expression-evaluation server. These parts of `ldb` use the compiler to do the machine-dependent work. Chapter 4 stands by itself, but Chapter 5 assumes an understanding of ideas presented in Chapter 4, especially in Section 4.4.

The reader interested in the machine-dependent parts of `ldb` is directed to Chapters 6, 7, and 8, which describe the debug nub, breakpoints and events, and stack walking. These parts of `ldb` use information from the compiler, but the machine-dependent work is done in the debugger. These chapters can be read independently of each other and of Chapters 4 and 5.

Chapter 9 explains why `ldb` uses an interpreted language and why that language is PostScript. It also presents the most interesting details of `ldb`'s implementation of PostScript.

The reader interested in the cost of retargeting `ldb` is directed to Chapter 10, which describes the sizes of each part of the system on each target and characterizes what has to be done to retarget the debugger proper, the debug nub, and the machine-dependent PostScript. Different parts of the system are more difficult to implement on some targets than on others; Chapter 10 explains the differences. Chapter 10 contains details accessible only to those who have read Chapters 6, 7, and 8.

Chapter 11 identifies good and bad aspects of `ldb`'s design, makes some suggestions about designing related tools, discusses how the techniques might be used elsewhere, and suggests some directions for future research.

The reader familiar with an earlier version of `ldb` (Ramsey and Hanson 1992) is directed especially to Sections 6.4.1 and 7.6, which describe how `ldb` calls procedures in the target, to Chapter 7, which describes more general breakpoints, and to Chapter 8, which describes a model of stack-frame layout used to eliminate some machine-dependent stack-walking code.

Chapter 2

Debugging With `ldb`

This chapter shows a user's view of `ldb`'s capabilities by means of an example debugging session. An overview of these capabilities provides an overview of the problems solved in later chapters. The example also serves as a framework in which to define commonly used terms.

`ldb` is a source-level debugger. Users refer to variables, procedures, and so on by name, and to locations in the program by source coordinate, i.e., file name, line number, and column. The debugger prints values that make sense in terms of the source programming language, C, not in terms of their representation on a particular machine. Finally, users can evaluate expressions and make assignments to variables by writing valid expressions and assignments in the source language.

`ldb`'s user interface is a simple, terminal-oriented user interface. It is not programmable. Frequently used commands have one-letter abbreviations; others must be spelled out in full. A complete list of commands appears at the end of this chapter.

The program used in this chapter computes and prints Fibonacci numbers. The source file, `fib.c`, is shown in Figure 3. I compile `fib.c` with the `-G` option, which makes `lcc` generate debugging information for `ldb` and include special startup code. This startup code gives control to the debug nub before calling `main`, giving the nub a chance to initialize its data structures and to install signal handlers, as described in Chapter 6. The nub also looks at the program's first argument, which could represent an instruction for the nub. For example, if the first argument is `--pause--`, the nub prints a message, stops the target, and waits for a debugger connection before calling `main`.

```
orchard % lcc -G -o fib fib.c
orchard % fib --pause-- 12
=> DEBUG NUB <=
to debug: ldb target fib connect orchard 2062
```

The message contains a command that, when typed in another window, starts the debugger and connects it to a target program.

```
void fib(short n) ^0{
  static int a[20];
  if (^1n > 20) ^2n = 20;
  ^3a[0] = a[1] = 1;
  { int i;
    for (^4i=2; ^7i<n; ^6i++)
      ^5a[i] = a[i-1] + a[i-2];
  }
  { int j;
    for (^8j=0; ^11j<n; ^10j++)
      ^9printf("%d ", a[j]);
  }
  ^12printf("\n");
^13}

main (int argc, char **argv) ^0{
  if (^1argc == 2)
    ^2fib(atoi(argv[1]));
  else
    ^3fib(10);
  return ^40;
^5}
```

Figure 3: Example program. Superscripts show stopping points.

`fib` is compiled and run on `orchard`, a big-endian SPARC. I run `ldb` on `dynastar`, a little-endian MIPS. The message "`ldb target fib connect orchard 2062`" holds three commands: start the debugger, choose a target program, and connect to an instance. The "`target fib`" command identifies `fib` as the target program, and it reads in PostScript describing `fib`.

```
dynastar % ldb
ldb > target fib
Loading symbols for fib... Checking linker table... Linker table OK
ldb fib (disconnected) >
```

Some PostScript comes from the compiler, some from the linker. The message "`Linker table OK`" means the two are consistent. The PostScript provides all the information that `ldb` needs to identify the names, locations, and types of procedures and variables, as described in Chapter 4. It also identifies `fib` as a SPARC program, enabling `ldb` to select its SPARC-dependent code and data.

The new prompt, "`ldb fib (disconnected) >`," indicates that a target, `fib`, has been selected, but that `ldb` is not yet connected to an instance of `fib`. The next step, `connect`, identifies a particular instance of `fib` to debug. When connecting to a target, `ldb` displays the event that made the target wait for a debugger. In this case, it is the initial pause requested by the argument `--pause--`. Other events that make targets wait include faults, e.g., invalid pointer references, and signals, e.g., from user interrupts.

```
ldb fib (disconnected) > connect orchard 2062
=> DebugNub_Pause(99) <=
* 0 ... void DebugNub_Pause(int arg = 99)
ldb fib (stopped) >
```

After displaying the event, `ldb` prints the *current focus*, initially the procedure activation that was executing when the event occurred. `DebugNub_Pause` is a special procedure used to make a target wait for a debugger.

Establishing a connection and printing the current focus involve several interactions with the debug nub. Messages are exchanged with the nub using a machine-independent protocol, so it does not matter that the host and target machines use different byte orders. The nub and its protocol are described in Chapter 6.

In typical use, the `ldb`, `target`, and `connect` commands are combined and executed with one sweep of the mouse:

```
dynastar % ldb target fib connect orchard 2062
Loading symbols for fib... Checking linker table... Linker table OK
=> DebugNub_Pause(99) <=
* 0 ... void DebugNub_Pause(int arg = 99)
ldb fib (stopped) >
```

A debugger must refer to locations in the target program at which execution is stopped or is to be stopped. `ldb` expects the compiler to define a set of *stopping points*. `lcc` places stopping points between C statements, and also at points within statements where control flow can diverge. Other compilers and debuggers use stopping points placed at coarser and finer granularities, like source-line numbers (Linton 1990) and expressions (Brooks, Hansen, and Simmons 1992). Figure 3 shows `fib.c`'s stopping points. In each procedure, stopping points are numbered in order of increasing object-code location. This order differs from source-code order; for example, when compiling `for` loops, `lcc` puts the termination test at the bottom of the loop. `ldb` refers to stopping points by procedure name and number, e.g., `fib:7`.

`ldb` offers several different kinds of breakpoints. Simple breakpoints are set at a single location, specified either as a stopping point or by source location. When a user requests a breakpoint at a source location, `ldb` finds the closest stopping point at or following the location and sets the breakpoint there. Source locations are referred to by file name, line number, and an optional column number.

```
ldb fib (stopped) > b fib:7 fib.c:13
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) >
```

Each kind of breakpoint is displayed differently. The first, set at stopping point `fib:7`, displays the stopping point and its source location: line 6, column 14 of file `fib.c`. The second, set at line 13 of file `fib.c`, displays the source location requested, the actual source location of the stopping point at which the breakpoint was set, in column 2 of that line, and the stopping point itself. The compiler supplies `ldb` with the source and object-code locations of each stopping point, as described in Chapter 4.

To refer to locations in the object code, **ldb** does not use raw addresses; instead it reports the object-code distance from the closest stopping point at or preceding the location. Object-code locations are displayed in angle brackets to distinguish them from source locations, which are displayed in parentheses.

All of **ldb**'s breakpoint commands rely on machine-dependent, low-level breakpoints that are set at a single object-code location. The implementations of low-level breakpoints are described in Chapter 7. If a breakpoint is set at a source location, the corresponding object-code location must be computed. The relation between object-code and source locations is stored in the PostScript symbol-table information emitted by the compiler and linker, as described in Chapter 4.

Continuing execution of **fib** makes it run until it hits a breakpoint. When that event occurs, **ldb** prints the event and the new focus.

```
ldb fib (stopped) > c
=> break  [1] at fib:7 (fib.c:6,14) <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) >
```

By default, **ldb** shows the object-code and source locations at which it is stopped, plus the procedure and its arguments. The **F** command shows the focus in more detail, including the values of local variables.

```
ldb fib (stopped) > F
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
                                int a[20] = {1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                             0, 0, 0, 0, 0, 0, 0, 0, 0};
                                int i = 2;
ldb fib (stopped) >
```

At this location, stopping point 7 on line 6, the local variables `a` and `i` are visible, but `j` is not. As shown in Figure 3, **fib**'s variables are declared in nested scopes; as described in Chapter 4, **ldb** uses a tree structure to represent the nesting and to compute which variables are visible at any location.

ldb can change the focus by walking the target call stack. The caller of **fib** is **main**.

```
ldb fib (stopped) > -
* 1 <main:2+0x14> (fib.c:17,4)
        int main(int argc = 2, char **argv = 0xf7fffa0c)
```

ldb indicates that control will return to **main** at object location `<main:2+0x14>`, which is between stopping points `main:2` and `main:3`. Chapter 8 describes the code needed to walk the stack and find where control will return.

The debug nub strips special arguments like `--pause--` so they are not visible to the user's program, as I can see by looking at `argv`.

```
ldb fib (stopped) > p argv[0]
char * argv[0] = (0xf7fffa84) "fib"
ldb fib (stopped) > p argv[1]
char * argv[1] = (0xf7fffa92) "12"
ldb fib (stopped) > p argv[2]
char * argv[2] = (null)
ldb fib (stopped) >
```

The `p` command prints the result of evaluating an expression. Expressions are evaluated by sending their text to the expression-evaluation server, which then asks for the symbol-table information describing free variables, like `argv`. The server compiles the expressions into PostScript procedures, which `ldb` interprets to get the values, as described in Chapter 5.

`ldb` can display all available information about a symbol, all of which comes from the compiler. For a variable, this information includes where it is stored and where it is defined. Because this display command is seldom used, it has a long, mnemonic name, not a single-letter name.

```
ldb fib (stopped) > allabout argv
argv is a variable: char **argv (at $r25) [defined at fib.c:15,27]
ldb fib (stopped) >
```

`argv` is in register 25, so `ldb` must fetch from register 25 to get its value. At the time `fib` stops, however, register 25 holds not `argv` but some local variable of `fib`; the register holding `argv` was saved on the stack when `fib` was called. When the focus changes to `main`, `ldb` must make it appear that register 25 holds `argv`. It does so by using an *abstract memory*, which can make locations on the stack appear to be registers. The PostScript code emitted by the compiler and expression server can manipulate the abstract memory as if it were the machine's real memory and registers.

`ldb`'s `t` command shows the entire call stack. The current focus is marked with a star.

```
ldb fib (stopped) > t
  0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
* 1 <main:2+0x14> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0xf7fffa0c)
  2 <mAiN:2+0x14> (MiniNub.c:7,9)
         int mAiN(int argc = 2, char **argv = 0xf7fffa0c,
                  char **envp = 0xf7fffa1c)
  3 <start+0x4c>  start (no symbol table information)
ldb fib (stopped) >
```

`mAiN` is a procedure in the debug nub, used to set up the nub's data structures and to pause the target before calling `main`. `start` is the system startup procedure. Its arguments are not shown because it was not compiled with `lcc`, so `ldb` has no symbol-table information for `start`. `ldb` can walk the stack without this information, but to do so requires extra retargeting effort, as described in Chapter 10.

To avoid stopping at every execution of `fib:7`, I make the breakpoint a conditional breakpoint, so `ldb` will stop only at the end of the loop when `i == n`.

```
ldb fib (stopped) > b
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) > take 1 when i == n
           break  [1] at fib:7 (fib.c:6,14) when i == n
ldb fib (stopped) >
```

The `take` command uses the expression-evaluation server to compile the expression in the context of location `fib:7`, even though the current focus is in `main`, because the compiled expression is not evaluated until `ldb` hits the breakpoint at `fib:7`. When execution of `fib` is continued, the target hits `fib:7` several times, but the expression evaluates to zero, so `ldb` continues silently. Finally the target hits `fib:7` when `i == n`, and `ldb` stops the target. The first `n` values of `a` have been computed.

```
ldb fib (stopped) > c
=> break  [1] at fib:7 (fib.c:6,14) when i == n <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) > F
* 0 <fib:7> (fib.c:6,14)
         void fib(short n = 12)
             int a[20] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 0, 0,
                          0, 0, 0, 0, 0, 0};
             int i = 12;
ldb fib (stopped) >
```

Continuing execution again stops at the second breakpoint, on line 14.

```
ldb fib (stopped) > c
=> break  [2] at fib.c:13 (fib.c:13,2) <fib:12> <=
* 0 <fib:12> (fib.c:13,2)  void fib(short n = 12)
ldb fib (stopped) >
```

`ldb` has other breakpoint commands in addition to `b`. `finish` continues execution until the current focus returns. If other activations of the same procedure finish, `ldb` does not stop, but continues, waiting for the current focus to finish.

```
ldb fib (stopped) > finish
=> finish fib[f7fff8e8] <=
* 0 <main:2+0x14> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0xf7fffa0c)
ldb fib (stopped) >
```

The number in brackets after `fib` identifies the activation being finished. Before `fib` finishes, it prints a newline, and the Fibonacci numbers appear on the standard output in the `orchard` window, which now shows

```
orchard % fib --pause-- 12
=> DEBUG NUB <=
to debug: ldb target fib connect orchard 2062
1 1 2 3 5 8 13 21 34 55 89 144
```

In addition to `b` and `finish`, `ldb` offers two other breakpoint commands. `next` steps to the next stopping point in the current activation, and `stepi` to the next machine instruction. `stepi` is implemented only on the MIPS and SPARC.

All of `ldb`'s breakpoint commands—simple breakpoints, conditional breakpoints, `finish`, and source- and instruction-level single stepping—are implemented using a single, machine-independent mechanism. The mechanism relies on low-level breakpoints to generate breakpoint events. When such an event occurs, `ldb` may evaluate an expression, print a message, check a procedure activation, or take an arbitrary action, then either stop the target or tell it to continue. Chapter 7 describes the event-action mechanism used to implement the different breakpoint commands.

After finishing the execution of `fib`, `ldb` is stopped between stopping points at `<main:2+0x14>`. Line 18 of Figure 3 shows the call to `fib` is a top-level expression, so it may not be clear why there is more code to be executed after the call. Because `ldb` can decode SPARC instructions, I can investigate by having `ldb` disassemble and display the object code.

```
ldb fib (stopped) > assembly main:2 4
main:2:        ld       [%i1 + 4], %o0
main:2+0x4:    call     _atoi;      nop
main:2+0xc:    call     fib:0;      nop
main:2+0x14:   ba       main:4;      nop
ldb fib (stopped) >
```

The display shows that the code must branch around the `else` clause on lines 19–20, to the `return` at `main:4`. Some of the instructions shown are pairs. When `ldb` encounters a branch instruction with a delay slot, it treats the pair as a single, two-word instruction.

Instruction decoding can require considerable machine-dependent code. `ldb` uses a special specification language to reduce the amount of code, but this language works best for RISC machines, and `ldb` can decode only MIPS and SPARC instructions, not VAX or 68020 instructions. Appendix B shows the specification for the MIPS.

Even though `main` is about to finish, it is possible to call `fib` again from `ldb`. While executing the call, `fib` hits a breakpoint.

```
ldb fib (stopped) > p fib(99)
=> break  [1] at fib:7 (fib.c:6,14) when i == n <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 20)
ldb fib (stopped) >>
```

The double-arrow prompt, `>>`, indicates that `ldb` is debugging a suspended procedure call; the evaluation of `fib(99)` has stopped partway through. If it is impossible or not profitable to continue, for example, if the event indicates an unrecoverable error, the suspended call can be abandoned; `unwind` reverts to the previous focus.

```
ldb fib (stopped) >> F
* 0 <fib:7> (fib.c:6,14)
         void fib(short n = 20)
             int a[20] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
                          377, 610, 987, 1597, 2584, 4181, 6765};
             int i = 20;
ldb fib (stopped) >> unwind
ldb fib (stopped) > F
* 0 <main:2+0x14> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0xf7fffa0c)
ldb fib (stopped) >
```

Unwinding the execution before `fib` finishes means the call does not produce any output.

It is also possible to undo the breakpoints temporarily in order to continue executing the call. The command "`u *`" undoes all breakpoints.

```
ldb fib (stopped) > p fib(5)
=> break  [1] at fib:7 (fib.c:6,14) when i == n <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 5)
ldb fib (stopped) >> u *
(*undone*) break  [1] at fib:7 (fib.c:6,14) when i == n
(*undone*) break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) >> c
void  fib(5) = (void)
ldb fib (stopped) >
```

Finally, continuing execution from `<main:2+0x14>` lets the target program finish.

```
ldb fib (stopped) > f
* 0 <main:2+0x14> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0xf7fffa0c)
ldb fib (stopped) > c
=> target terminated <=
ldb fib (disconnected) >
```

The `orchard` window, in which `fib` ran, shows the output from the first and third calls, because they ran to completion.

```
orchard % fib --pause-- 12
=> DEBUG NUB <=
to debug: ldb target fib connect orchard 2062
1 1 2 3 5 8 13 21 34 55 89 144
1 1 2 3 5
orchard %
```

The breakpoints are retained for the next session. `ldb`'s `r` command re-activates an undone breakpoint. Redone breakpoints are automatically planted when `ldb` next connects to an instance of this target:

```
ldb fib (disconnected) > r *
           break  [1] at fib:7 (fib.c:6,14) when i == n
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (disconnected) >
```

This example shows most of `ldb`'s features: walking the call stack, printing values, evaluating expressions, and interacting with a target process. The chapters that follow describe the implementations of these features and show how the implementations are retargeted from one machine to another. Figure 4 summarizes `ldb`'s commands. Some commands have synonyms, and many have default arguments; Figure 4 does not show all the synonyms or all the variations.

`allabout`	Print all available information about a symbol.
`assembly`	Show disassembled object code.
`b`	Plant a breakpoint or list breakpoints.
`c`	Continue execution of the target.
`call`	Evaluate an expression and discard the result.
`cd`	Change `ldb`'s working directory.
`connect`	Connect to an instance of a target program.
`d`	Delete a breakpoint.
`disc`	Disconnect from the target program, which may continue, wait for another debugger, or exit.
`eval`	Evaluate an expression and discard the result.
`event`	Print the event that caused the target to stop.
`f`	Print the current focus.
`F`	Print the current focus in more detail.
`finish`	Continue execution of the target until the designated activation finishes.
`i`	Pass text directly to the PostScript interpreter.
`n`	Continue execution of the target until reaching a stopping point in the designated activation.
`q`	Exit both debugger and target program.
`r`	Redo an undone breakpoint (see also `u`).
`run`	Fork a new target process as a child of `ldb` and connect to it.
`s`	Step one machine instruction.
`t`	Trace the whole call stack.
`T`	Trace the whole call stack, showing more detail.
`take`	Make a breakpoint conditional or skip the next k hits.
`u`	Undo a breakpoint. Can be redone (see also `r`).
`unwind`	Stop debugging a suspended procedure call and return to the previous context.
`whatis`	Print information about a symbol.
`-`	Move the current focus up one frame.
`+`	Move the current focus down one frame.
n	Move the current focus to frame n.

Figure 4: Summary of `ldb` commands.

Chapter 3

Abstract Memories

An abstract memory provides a machine-independent model of a target's memory and registers. PostScript code generated by the compiler and expression-evaluation server includes fetches and stores that operate on abstract memories. For example, in Chapter 2, when `ldb` prints the focus,

```
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
```

the value of `n` is fetched from register 24 in an abstract memory. All commands in Chapter 2 that work on stopped targets, including setting breakpoints, walking the call stack, and printing assembly code, work through abstract memories.

An abstract memory is a collection of *spaces*, denoted by lower-case letters, e.g., `d` for data space, `r` for registers, etc. Locations within a space are determined by an integer offset. `ldb` provides several "addressing modes" to refer to locations, including an immediate mode. These modes make locations more general than simple addresses; for example, they can refer to register contents or to values located by indirection with respect to registers. Given an abstract memory and a location, `ldb` can fetch and store three sizes of integers (8, 16, and 32 bits) and three sizes of floating-point values (32, 64, and 80 bits). Abstract memories and locations are PostScript values as well as Modula-3 values, so fetching and storing is as easy to do in PostScript as in Modula-3.

`ldb` assumes that every machine has code and data spaces, which may refer to the same locations or to different locations depending on the target architecture. Other spaces are added as needed to provide a good model for a particular architecture, for example, to provide a separate space for each register set. Figure 5 shows the spaces that `ldb` uses for the MIPS and how many locations are in each space; `r` represents integer registers, `f` floating-point registers, and `x` "extra registers." The extra registers are the program counter and the virtual frame pointer; putting them into the abstract memory makes them easily accessible from PostScript. The MIPS has no actual frame pointer, and PostScript code generated by `lcc` uses the virtual frame pointer, not the stack pointer, to address local variables. On the MIPS, the code and data spaces refer to the same locations, but

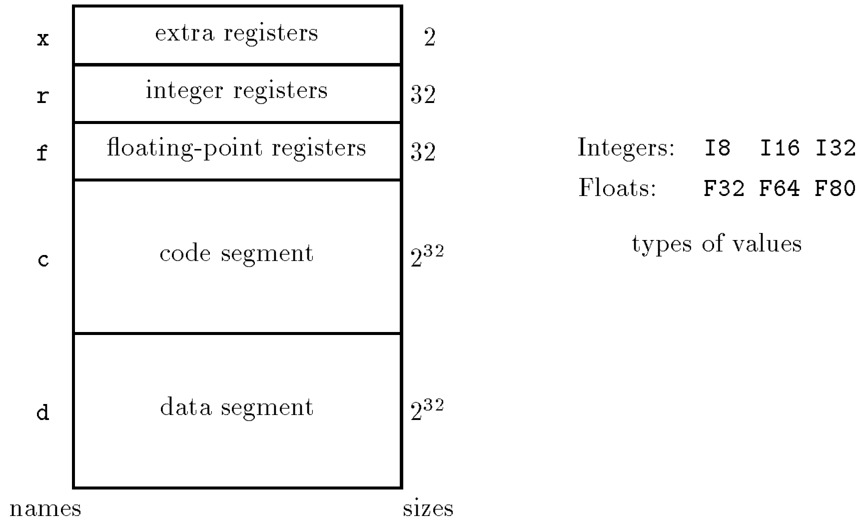


Figure 5: Structure of the abstract memory used on the MIPS.

they are treated slightly differently. For example, storing a value in the code space also flushes the instruction cache.

Given an abstract memory like that shown in Figure 5, `ldb` prints values and evaluates expressions without executing any machine-dependent Modula-3 code; only the PostScript emitted by the compiler or expression-evaluation server is machine-dependent. For the debugger, the machine-dependent part of the problem is to construct the abstract memory. Every abstract memory is associated with some stack frame, so abstract memories are constructed by the stack-walking code described in Chapter 8. To simplify the job, `ldb` defines several different kinds of abstract memories, which are used to implement different spaces, then combined. They are combined in the same way on all targets by machine-independent code. This code uses machine-dependent specifications of the names and sizes of the spaces in the memory, as shown in Section 3.2.2. Creating a full abstract memory out of simple, reusable pieces makes retargeting easier than if abstract memories had to be implemented from scratch for each target.

Section 3.1 explains the different kinds of abstract memories and how they are combined to form a full memory like that shown in Figure 5. Section 3.2 shows how values, locations, and abstract memories are defined in Modula-3; this material is needed to understand the detailed examples of register-restoring code in Chapter 8.

3.1 Kinds of abstract memories

The debug nub provides the foundation of all abstract memories. On all machines, it provides access to the code and data spaces, as shown at the right of Figure 6. On the left of Figure 6 is the full abstract memory from Figure 5, and in the middle are the instances of the other kinds of abstract memories. Fetch and store requests flow from left to right, replies from right to left. Only the debug nub, on the far right, changes the target memory directly. Although the nub is written in C, not Modula-3, it implements the same abstraction as the other instances.

The dashed line in Figure 6 separates the debugger process from the target process. The nub runs in the target process, and `ldb` must have access to the abstract memory in its own process. A *wire* is a kind of abstract memory that holds a connection to the nub; it forwards fetch and store requests to the nub, which executes them and returns the results.

Part of constructing an abstract memory is identifying the locations where registers are saved and making those locations appear as registers. Registers are saved either on the stack or in a *process context*, which is an area in memory that holds the state of a stopped program. The debug nub creates a process context every time the target stops. The *alias* memory translates requests for locations in register spaces into requests for locations in the data space, either in the process context or on the call stack. For example, on the MIPS, the process context holds integer registers at offset 12 and floating-point registers at offset 152. If the context is located at address `0x8000`, the alias memory translates a request for integer register `r23` into a request for the word at offset $12 + 4 \times 23$ from the beginning of the context, that is, the word at `0x8068`. The aliases for the MIPS extra registers refer not to locations on the stack but to *immediate* locations. An immediate location is like an immediate operand in a machine instruction; it holds a value but does not correspond to any location in the target data space. `ldb` can create new immediate locations at will. The MIPS virtual frame pointer, for example, is not represented in any location in the target memory; the debugger computes the virtual frame pointer by adding the frame size to the stack pointer and putting the result in an immediate location.

When the compiler puts a value in a register and the register is saved in memory, the resulting representation may be different than if the compiler simply put the variable in memory directly. For example, if a character is put in a register and the register is saved in memory, the address of the character may not be the same as the address of the saved register, depending on the byte order of the target. If the character variable `c` is allocated to register `r23`, and `r23` is saved at `0x8068`, as in the previous paragraph, `c` may be at `0x8068` or at `0x806b`. It is therefore impossible to fetch the byte representing `c` directly from memory unless the target byte order is known. `c`'s value can be recovered without knowing the byte order by composing two operations: first fetch the full 32-bit word from address `0x8068`, then narrow the result to 8 bits by masking with `0xff`. The *register* memory hides the need for such size conversions; it transforms each fetch into a fetch and a widen and each store into a narrow and a store. The register memory changes only the sizes, not the

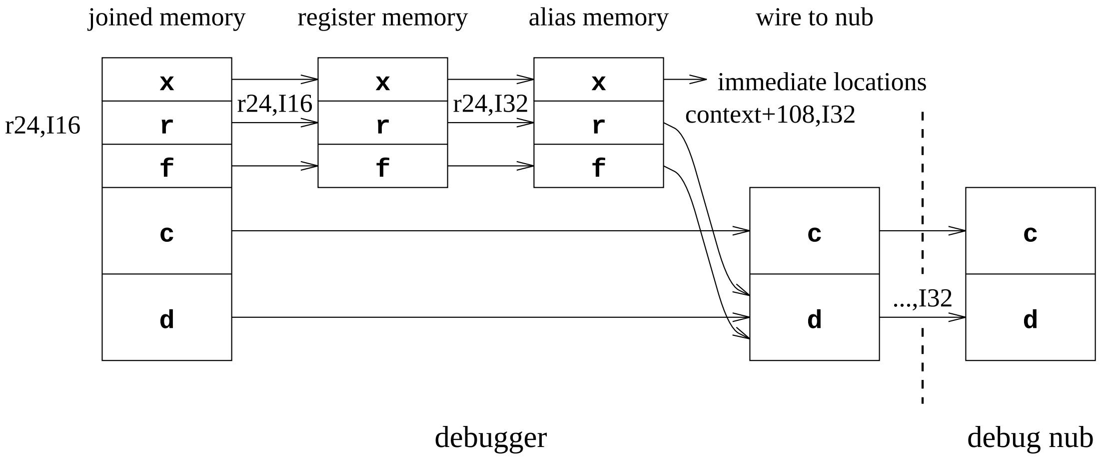


Figure 6: Implementation of an abstract memory for one stack frame.

types, of values; it never converts an integer to a floating-point value or vice versa. Conversions are described by a machine-dependent specification, which gives the sizes of registers used to hold each size and type of value. The default for 32-bit machines specifies that 32 bits are used to hold integers of all sizes, but that the number of bits used to hold a floating-point value is the number of bits in the value.

Original type	*Transformed type*
8-bit integer	32-bit integer
16-bit integer	32-bit integer
32-bit integer	32-bit integer
32-bit float	32-bit float
64-bit float	64-bit float
80-bit float	80-bit float

Register memories make byte order irrelevant, enabling `ldb` to execute the same code whether debugging a program on a little-endian or a big-endian MIPS, for example.

The register memory and wire together provide all the spaces shown in Figure 5. The *joined* memory combines them, forming an abstract memory containing all the necessary spaces. The abstract memory associated with each stack frame is a joined memory. The joined memory passes fetch and store requests to the appropriate underlying memory, and they travel through the graph shown in Figure 6. The figure shows the sequence of requests used to fetch the value of `n` in order to print

```
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
```

The C type `short` corresponds to the low-level type 16-bit integer, indicated in Figure 6 by `I16`. `n` is stored in integer register 24, so PostScript code fetches an integer of type `I16` from offset 24 of space `r`. The joined memory identifies the space `r` as being served by the register memory, so it fetches from the register memory. The register memory fetches the corresponding full 32-bit word from the alias memory. Register 24 is an alias for a location in the data space 108 bytes after the beginning of the context, so the alias memory fetches from the wire at that location. The wire sends a message to the nub, which fetches the word using the target's byte order and sends the value back to the debugger in little-endian order, where it gets returned up the call chain. The register memory narrows the value, returning only the low-order 16 bits, and that is the value finally returned by the joined memory. Requests to fetch values from the data or code spaces bypass the register and alias memories; the joined memory fetches directly from the wire.

3.2 Implementing abstract memories

This section shows how abstract memories are implemented in Modula-3. The code presented here is important primarily for understanding the detailed examples in Chapters 8 and 9.

The definitions needed to use abstract memories appear in the `Memory` and `Location` interfaces. The target holds either integer or floating-point values, represented in the debugger by `Memory.Integer` and `Memory.Real`. `Memory.Value` acts as a union type; allocation of Modula-3 objects is too expensive for the debugger to use a pair of object types to represent the disjoint union of integer and real. `Memory.Type` is used in requests to identify both the type and size of a value.

```
TYPE
  Memory.Integer = INTEGER;
  Memory.Real    = LONGREAL;
  Memory.Type    = { I8, I16, I32, F32, F64, F80 };
  Memory.Value   = RECORD
                     x : Memory.Real := 0.0d0;
                     n : Memory.Integer := 0;
                     integer := TRUE;
                   END;
```

This implementation assumes that `INTEGER` is big enough to hold any target integer, and similarly for `LONGREAL`. This assumption does not hamper `ldb`'s retargetability, because a machine's widest types are always wide enough to hold any value on that machine. It makes some cross-debugging difficult, however; `ldb` cannot run on a 32-bit host and debug a 64-bit target.

In the Memory interface, both abstract memories and locations therein are abstract data types, on which the only operations defined are fetch and store.

```
TYPE
  Memory.T <: InterpTypes.Other;
  Memory.Location <: Displayed.T;

PROCEDURE Memory.Fetch(m: Memory.T; where: Memory.Location;
                       type: Memory.Type) : Memory.Value;
PROCEDURE Memory.Store(m: Memory.T; where: Memory.Location;
                       what: Memory.Value; type: Memory.Type);
```

Fetch and Store raise exceptions when invalid addresses are used.

The operator <: denotes the subtype relation. InterpTypes.Other is the type of an "opaque PostScript object" (Chapter 9); because Memory.T is a subtype of InterpTypes.Other, values of type Memory.T can be manipulated directly by the PostScript interpreter. Displayed.T is a subtype of InterpTypes.Other that has a print method used by the user interface; because Memory.Location is a subtype of Displayed.T, it must implement a print method giving suitable output syntax. In the following example, $r30 is the output syntax for a location denoting register 30. The implementation of the allabout command prints it by calling the print method of that location:

```
ldb fib (stopped) > allabout i
i is variable: int i (at $r30) [defined at fib.c:5,8]
ldb fib (stopped) >
```

The Location interface provides ways of constructing locations, but it does not reveal their representation. A location in an abstract memory is somewhat like an operand specifier on a CISC machine. The basic locations specify either an immediate value or an offset and a space. The space is a lower-case letter, and the offset is of type Memory.Address. Again, INTEGER is assumed to be large enough to hold any address in the target space.

```
TYPE
  Memory.Space   = ['a'..'z'];
  Memory.Address = INTEGER;

PROCEDURE Location.Absolute (offset: Memory.Address;
                             space : Memory.Space) : Memory.Location;
PROCEDURE Location.Immediate(v : Memory.Value)     : Memory.Location;
```

Figure 28 on page 120 shows a use of Location.Absolute; Chapter 4 shows examples of its use in PostScript. Procedure BindLinkage on page 122 shows examples of Location.Immediate, including the creation of an immediate location for the MIPS virtual frame pointer.

A location can be constructed by shifting, i.e., adding an integer constant to the offset of an existing location without changing its space. Shifting an immediate location is nonsensical.

```
PROCEDURE Location.Shifted (offset: Memory.Address;
                            base  : Memory.Location) : Memory.Location;
```

Finally, a location can be indirect; the offset is fetched from another, base location.

```
PROCEDURE Location.Indirect(offset:  Memory.Address;
                            space :  Memory.Space;
                            base  :  Memory.Location;
                            type  := Memory.Type.I32) : Memory.Location;
```

The locations of local variables that `lcc` puts on the stack are indirect with respect to the frame pointer. Given an abstract memory, a shifted or indirect location can always be simplified to an absolute location. Section 9.1.2 shows `ldb`'s implementations of the PostScript operators based on the procedures in the `Memory` and `Location` interfaces.

The PostScript code emitted by the compiler and expression-evaluation server uses locations in stylized ways. Absolute locations refer to variables allocated to registers or to fixed locations in the data space, and to all locations in code. Indirect locations refer to variables allocated on the stack, which are indirect with respect to the frame pointer or argument pointer. Shifted locations refer to the elements of arrays, structures, or unions, whose offsets are recorded by the compiler. Immediate locations are used to represent integer constants, read-only values, and sometimes the results of expression evaluation.

Fetching from an immediate location or from a register memory may require a size conversion. The `Memory` interface defines a procedure used to do such conversions.

```
PROCEDURE Memory.ConvertValue(source:Memory.Value; type:Memory.Type):Memory.Value;
```

This procedure can be used only to change the size of a value; it raises an exception if asked to convert between integer and floating-point data.

The procedures in the `Location` interface allocate a Modula-3 object of type `Memory.Location` to hold their results. The `Memory` interface provides a shortcut for those cases in which an absolute location is created only to be fetched from; the shortcut avoids the allocation.

```
PROCEDURE Memory.FetchAbs(m: Memory.T; offset: Memory.Address;
                          space: Memory.Space; type: Memory.Type) : Memory.Value;
```

`FetchAbs(m,o,s,t)` is equivalent to `Fetch(m,Location.Absolute(o,s),t)`. `Memory.StoreAbs` is analogous.

`Memory.T` is an abstract supertype, i.e., a type that is not instantiated directly. Each kind of abstract memory defines a different subtype of `Memory.T`. The `Fetch` and `Store` procedures use subtype-independent code to resolve complex locations to absolute or immediate ones and to fetch from immediate locations. Fetching from and storing to absolute locations is implemented by different methods supplied by the different subtypes. The code that creates an instance of a subtype must indicate which spaces are valid arguments to that instance's fetch and store methods; the subtype-independent code guarantees that the methods are called only with those spaces.

As described above, `ldb` uses a joined memory to combine different abstract memories that recognize different spaces. The procedure `Memory.Join` creates such a memory.

```
PROCEDURE Memory.Join(m1, m2: Memory.T) : Memory.T;
```

It is not necessary to identify the subtype of `Memory.T` that implements the joined memory. The set of spaces valid for a joined memory is the union of the sets valid for the two components. The absolute fetch and store methods use whichever underlying memory contains the space in question, using `m1` if there is a conflict. The joined memory is the only subtype implemented in the `Memory` module itself; the other subtypes are defined in other interfaces and implemented in the corresponding modules.

Alias memories are used to create new spaces; for example, an alias memory creates the spaces `r`, `f`, and `x` in Figures 5 and 6. Locations in those spaces are aliases for other locations in an existing, underlying memory, e.g., the connection to the debug nub. These spaces contain only aliases; requests are never passed through unchanged to the underlying memory. `AliasedMemory.T`, the subtype that implements alias memories, has special methods used to create and query aliases. An alias memory is created by specifying the underlying memory, the names of the spaces to contain aliases, and the sizes of those spaces. Its declaration indicates that `AliasedMemory.T` is a subtype of `Memory.T` and that it has the additional methods shown:

```
PROCEDURE AliasedMemory.New(underlying: Memory.T;
                            READONLY spaces: ARRAY OF Memory.Space;
                            READONLY sizes:  ARRAY OF INTEGER) : AliasedMemory.T;
TYPE
  AliasedMemory.T <: Memory.T OBJECT METHODS
    bind   (offset: INTEGER; space: Memory.Space; where: Memory.Location);
    binding(offset: INTEGER; space: Memory.Space) : Memory.Location;
    bound  (offset: INTEGER; space: Memory.Space) : BOOLEAN;
  END;
```

When first created, the memory contains no aliases, and references to it raise exceptions. The `bind` method creates an alias for a location in the memory `underlying` passed to `AliasedMemory.New`. The `bound` and `binding` methods tell whether a location has an alias and return the alias associated with a location. All of the methods raise exceptions if the offset is not within the bounds given for the space when the memory was created. The `binding` method also raises an exception if no alias exists for the location requested. Figure 28 on page 120 shows examples of the use of these methods.

An alias memory creates bindings to the locations where registers are saved on the stack. A register memory uses an alias memory as an underlying memory and guarantees that operations on the underlying memory use only complete saved registers, not parts thereof. Unlike an alias memory, a register memory creates no new spaces; it only changes the treatment of existing spaces. A register memory is created by specifying the underlying memory, the set of spaces in which the conversions are used, and the conversions desired. The specification of the conversions maps the type of a request to the type used on the underlying memory. Most 32-bit targets use the default specification, `RegisterMemory.Bits32`, which maps integer requests to 32-bit requests and leaves floating-point requests unchanged. For clarity, I repeat the definition of `Memory.Type`, showing it by its unqualified name, `Type`.

```
TYPE
  Type = { I8, I16, I32, F32, F64, F80 };
  RegisterMemory.Specification = ARRAY Type OF Type;
CONST
  RegisterMemory.Bits32 = RegisterMemory.Specification
             { Type.I32, Type.I32, Type.I32, Type.F32, Type.F64, Type.F80 };
PROCEDURE RegisterMemory.New(underlying: Memory.T; spaces: SET OF Memory.Space;
                             READONLY spec := RegisterMemory.Bits32) : Memory.T;
```

`RegisterMemory.New`, like `Memory.Join`, returns an anonymous subtype of `Memory.T`.

Although the register memory uses a machine-dependent specification of type conversions, its fetch and store methods are machine-independent. They use `Memory.ConvertValue` to do the conversions. Here, for example, is the implementation of the absolute fetch method.

```
PROCEDURE RegisterMemory.Fetch(m:RegisterMemory.T; offset:INTEGER;
                               space:Memory.Space; type:Memory.Type) : Memory.Value =
BEGIN
  IF space IN m.registerSpaces THEN
    RETURN Memory.ConvertValue(
                    m.underlying.fetchAbs(offset, space, m.spec[type]), type);
  ELSE
    RETURN m.underlying.fetchAbs(offset,space,type);
  END;
END RegisterMemory.Fetch;
```

3.2.1 Trapped memories

The abstract memories provided by the nub and wire are actually more complicated than shown above. To understand why, a digression about breakpoints is necessary.

To implement breakpoints, a debugger replaces some of the original instructions of the target program with new instructions, traps or branches, that cause control to be transferred from the target to the debugger. The debugger must remember the original instructions so it can restore them later when it deletes or undoes a breakpoint. If the debugger crashes, the original instructions are lost, and the breakpoints cannot be deleted. When standard Unix debuggers are used, this problem is subsumed by a greater problem; if a standard Unix debugger crashes, the ability to debug its target is lost.

When remote debugging is possible, it is unacceptable to lose access to a target program just because the debugger or its machine crashes; a user must be able to connect a second instance of the debugger to the same target. If the second instance is to undo breakpoints planted by the first instance, information about original instructions must be saved in the target, not the debugger. Storing such information is the responsibility of the trapped memory, which is implemented in the debug nub. It is called a "trapped" memory because `ldb` uses traps, not branches, to implement breakpoints.

A trapped memory is machine-independent; it uses no information about the target instruction set. It needs only to store integers into the code space and to undo the effects of such stores, which it does by its `plant` and `suspend` methods. The caller of the `plant` method is responsible for using the appropriate type and bit pattern to describe trap instructions. Planting traps does not change the contents of the abstract memory as seen by the fetch and store methods.

```
TYPE
  Memory.IntegerTypes = [Memory.Type.I8..Memory.Type.I32];
  TrappedMemory.T = Memory.T OBJECT METHODS
    plant  (pc:Memory.Address; trapbits:Memory.Integer; type:Memory.IntegerTypes);
    suspend(pc:Memory.Address);
    traps  () : REF ARRAY OF Memory.Address;
  END;
```

The `plant` method is much like a store method, except that the space is known to be the code space `c` and the value is known to be an integer. `Memory.IntegerTypes` is the subrange of `Memory.Type` that describes integer types. The `traps` method is needed for the second instance of the debugger to identify and locate traps planted by the first instance.

3.2.2 Combining abstract memories

Combined abstract memories like that shown in Figure 6 are assembled by `ldb`'s generic stack-walking code, described in Section 8.2. The parameter to the generic code is a machine-dependent *configuration interface*. Within that interface, four constants hold the machine-dependent details needed to create the constituent parts of Figure 6. The constants for the MIPS are:

```
CONST
  AliasSpaces = ARRAY OF ['a'..'z'] { 'r', 'f', 'x' };
  AliasSizes  = ARRAY OF INTEGER    {  32,  32,  2  };
  RegSpaces   = SET   OF ['a'..'z'] { 'r',      'x' };
  RegSpec     = RegisterMemory.Bits32;
```

which describe the abstract memory shown in Figure 5. `AliasSpaces` specifies that a MIPS abstract memory is to have `r`, `f`, and `x` spaces in addition to the basic `c` and `d` spaces, and `AliasSizes` shows how many locations those spaces contain. `RegSpaces` indicate which of those spaces require type transformation by a register memory, and `RegSpec` specifies the transformation. On the MIPS, the `r` and `f` spaces represent integer and floating-point registers, and the `x` space holds the values of the program counter and virtual frame pointer. `RegSpec` uses default treatment for 32-bit machines, which is shown in full above. Because there is no change to the size of floating-point requests, it is not necessary to include `f` in `RegSpaces`.

The generic code that creates and combines abstract memories according to this specification is

```
PROCEDURE AbstractMemory(...) : Memory.T =
VAR this := AliasedMemory.New(m, AliasSpaces, AliasSizes);
BEGIN
  ⟨Create aliases for registers⟩
  RETURN Memory.Join(RegisterMemory.New(this, RegSpaces, RegSpec), m);
END AbstractMemory;
```

where `m` represents the wire connecting the debugger to the nub. The same code is used to create abstract memories for the 68020 and the VAX, shown in Figures 7 and 8.

The 68020 has three register sets in the hardware, plus an extra to hold the program counter. Its abstract-memory specifications are

```
CONST
  AliasSpaces = ARRAY OF ['a'..'z'] { 'a', 'r', 'f', 'x' };
  AliasSizes  = ARRAY OF INTEGER    {  8,   8,   8,   1  };
  RegSpaces   = SET   OF ['a'..'z'] { 'a', 'r', 'f', 'x' };
  RegSpec     = RegisterMemory.Specification { I32, I32, I32, F80, F80, F80 };
  I32 = Memory.Type.I32;
  F80 = Memory.Type.F80;
```

The 68020 does not use the default register-memory specification because on the 68020 all floating-point registers are 80 bits wide.

The VAX has a single register set, which includes the program counter. Its specifications are:

```
  AliasSpaces = ARRAY OF ['a'..'z'] { 'r' };
  AliasSizes  = ARRAY OF INTEGER    { 16  };
  RegSpaces   = SET   OF ['a'..'z'] { 'r' };
  RegSpec     = RegisterMemory.Bits32;
```

Creating aliases for registers requires detailed information about where those registers are saved on the stack or in the process context. For that reason, alias creation is discussed with the other stack-walking code in Section 8.2.

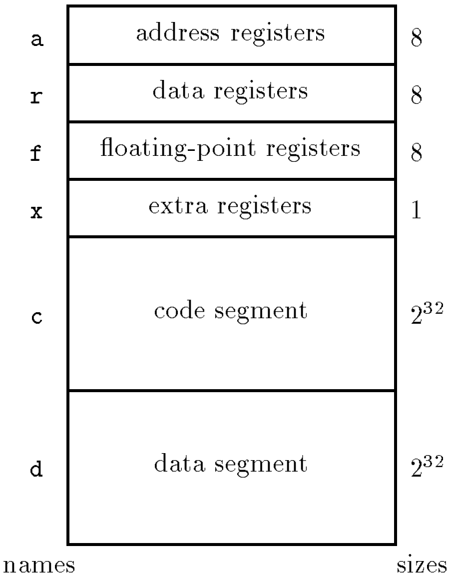


Figure 7: 68020 abstract memory.

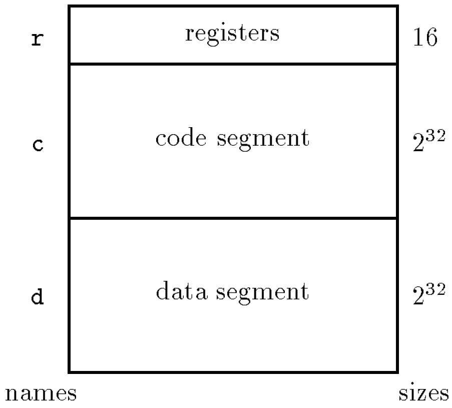


Figure 8: VAX abstract memory.

3.3 Discussion

The structures of abstract memories differ from architecture to architecture. Retargeting is straightforward because complex abstract memories are created by combining simple ones. As shown above, the machine-dependent aspects of those simple abstract memories can be specified in just a few lines.

An earlier version of abstract memories had a richer set of types modeled on the C basic types. For example, it had three different 32-bit integer types: signed integer, unsigned integer, and pointer. Since every fetch and store method handles every type, reducing the number of types simplified the implementation of every kind of abstract memory. The simpler implementations are easier to understand and maintain. The cost of the change was slight; some PostScript procedures, as shown in Section 4.4, must now sign-extend integer values before using them.

`ldb`'s locations are similar to but simpler than the "locatives" provided by DEC SRC's Loupe debugger for Modula-2+ (DeTreville 1986). Locatives include a length as well as a location, and they use bit-level rather than byte-level granularity. The extra complexity confers some advantages; for example, it is possible to refer to a bit field using a single locative, whereas `ldb` must use a triple: the location of the word containing the field, the field's size, and the field's offset within the word. `ldb`, however, does not manipulate bit fields directly, but only by interpreting PostScript emitted by the compiler. The compiler does not generate code that refers to locations at the bit level, nor does it expect for each location to be associated with a length, so adding these features would not simplify the implementation of the compiler, the expression server, or the debugger.

Although locations are used idiomatically, it would have been unnecessarily restrictive to have implemented just the idioms; the general mechanism is no more difficult to implement. Languages other than C may require different idioms. For example, the location of a field of a Modula-3 object might be computed by indirection, not shifting, with respect to the location of the object, because the DEC SRC compiler uses indirection in its representation of objects.

The assumption that an integer on the debugger is big enough to hold an integer on the target is made explicit in the `Memory` interface, but the assumption pervades `ldb`. For example, the PostScript code emitted by the expression-evaluation server uses standard PostScript operators to implement arithmetic on target integers. Eliminating the assumption would require using an arbitrary number of debugger words to represent one target word, with appropriate changes to arithmetic. Modula-3 provides no way to implement such changes easily, e.g., by operator overloading. Moreover, if abstraction were used to hide the representation of target data, the resulting implementation might be much less efficient, because the only available Modula-3 compiler does not inline procedure calls. This extra overhead would be acceptable if there were a demonstrated need for such cross-debugging, but the most common case is to run the debugger on a machine of the same word size, if not the same architecture, as the target. Even when sizes differ, using a machine with large words to debug a machine with small words seems more likely than the converse; for example, a 32-bit workstation might be used to debug a program running on a 16- or 8-bit pocket computer. `ldb` could handle both these scenarios on machines of various word sizes without change to its representation of target integers.

`ldb` does not debug core files. Doing so would require creating an abstract memory that corresponds to a core file. Machine-dependent code would be needed to get the data space and registers from the core file and the code space from the corresponding executable file. Programs that can be debugged with `ldb` do not normally create core files because the debug nub intercepts signals that lead to core-file creation. A core file is created only if there is an internal error in the nub.

Chapter 4

PostScript Symbol Tables

ldb needs information from the compiler. Users refer to variables, procedures, and so on by name, so the debugger must find the meanings of the names. Names are defined in nested scopes, so the same name can mean different things at different locations in a program. Nested scopes can be described by a tree, provided that every symbol's parent is the previously declared symbol in either the current or an enclosing scope. ldb uses stopping points to refer to locations, as described in Chapter 2. Each stopping point is associated with a symbol, and the meaning of a name at a stopping point is determined by starting at the associated symbol, then walking up the tree until finding a symbol with that name.

Figure 9 shows the procedure fib from Figure 3 of Chapter 2. Figure 10 shows the tree containing fib's local symbols; the box at the right shows which symbol is associated with each stopping point. For example, stopping point fib:7 is associated with the entry for the symbol i, and i, a, and n are visible. The F command from Chapter 2 uses the tree to print the values of all visible local variables, i and its ancestors.

```
ldb fib (stopped) > F
* 0 <fib:7> (fib.c:6,14)
        void fib(int n = 12)
            int a[20] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 0, 0,
                         0, 0, 0, 0, 0, 0};
            int i = 12;
```

Most local scopes have few symbols, so linear search up the tree works well. ldb puts top-level symbols in hash tables for faster lookup. Top-level symbols declared extern are put in a single hash table shared among all compilation units. Top-level symbols declared static are put in separate, per-compilation hash tables, since such symbols are private to one compilation unit.

ldb associates each meaningful name with a symbol, and each symbol has a type. The ways in which symbols and types are used determine what ldb needs to know about them. When debugging,

```
void fib(int n) ^0{
  static int a[20];
  if (^1n > 20) ^2n = 20;
  ^3a[0] = a[1] = 1;
  { int i;
    for (^4i=2; ^7i<n; ^6i++)
      ^5a[i] = a[i-1] + a[i-2];
  }
  { int j;
    for (^8j=0; ^11j<n; ^10j++)
      ^9printf("%d ", a[j]);
  }
  ^12printf("\n");
^13}
```

Figure 9: Procedure `fib` with stopping points.

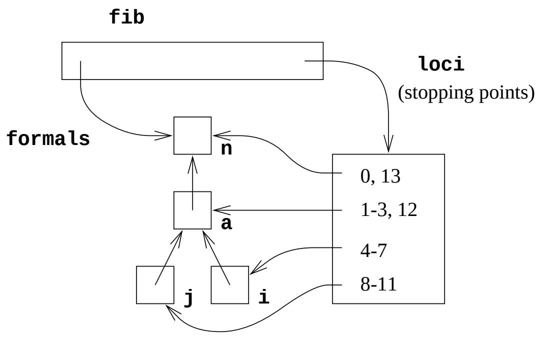


Figure 10: The tree structure of `fib`'s symbol table.

symbols and types are used primarily to print the values of variables, which requires the name, location, and type of the variable. Symbols for procedures also contain information about the stopping points of that procedure, as well as a way of finding the procedure's formal parameters, as shown in Figure 10.

Symbols provide the locations of values, but types determine how they are printed. Most debuggers define a fixed set of type constructors and provide a fixed set of printing procedures, one per constructor. `ldb` uses a novel technique: the compiler supplies a printing procedure with each type in the symbol table. This technique, described in Section 4.4, enables `ldb` to print values without knowing the layout of run-time data structures, avoiding a source of machine dependence. It also makes `ldb` independent of the type system of the target programming language.

Information in symbols and types is also used to evaluate expressions. As described in Chapter 5, `ldb` uses a modified instance of the compiler to evaluate expressions. The symbols and types used by `ldb` must therefore contain enough information to make it possible to reconstruct the compiler's representation.

Most of the information needed by `ldb` is created by the compiler, some by the linker. `lcc` determines all symbol and type information and the locations of automatic and register variables; the Unix linker determines the locations of procedures, stopping points, and global and static variables. `ldb` encapsulates information from the linker in a *linker state*, an object with methods that provide the location of an external symbol, the address of the procedure containing a given program-counter value, the address of the next procedure following a procedure, or the symbol table of the linked program.

4.1 Representing debugging information in PostScript

`ldb` uses PostScript (Adobe 1985) to represent symbol-table information. Although PostScript is a language for printers, it has no linguistic support for printing and imaging; it is a general-purpose, stack-based programming language similar in spirit to FORTH (Moore 1974). The printing and imaging support comes from an extensive collection of built-in operators. The software on graphics workstations is analogous; the hardware provides special operations to support graphics, but there is no linguistic support for graphics. Graphics programs are ordinary C programs that call library procedures to use the graphics hardware (SGI 1991). In printing and in `ldb`, PostScript operators play the role of library procedures, but the two applications use different sets of operators.

Unlike FORTH, PostScript has both scalar and composite types, and the types of all operations are checked at run time. PostScript has integers, reals, Booleans, arrays, dictionaries, names, strings, and procedures. PostScript dictionaries are like hash tables; they associate keys with values. A predefined dictionary associates names with operators, which are invoked by their names. Both procedures and operators find their arguments and return their results on the PostScript operand stack, usually called just the stack. Dictionaries and arrays are built by applying special bracketing operators, shown below, to the contents of the operand stack.

There is also a dictionary stack, which is used to associate names with values, including operators; dictionaries on the stack play the role of nested scopes used to resolve operator names. PostScript procedures can change the dictionary stack at run time, changing the meanings of names.

Because symbol-table information comes from both compiler and linker, `ldb` organizes it in two levels. The compiler provides a *top-level dictionary* for each compilation unit. The top-level dictionary provides access to symbols and types in two ways: through an array of procedures defined in the unit and through a dictionary associating the names of global symbols with their symbol-table entries. Top-level dictionaries from different units can be combined; at debug time `ldb` uses a single top-level dictionary that describes the whole program. Information determined at link time is stored in a *linker table*, which incorporates by reference a top-level dictionary for each compilation unit in the program. `ldb` uses the information in the linker table to implement its linker state.

The rest of this section describes `ldb`'s representation of symbol-table information, working from symbols and types up to top-level dictionaries and linker tables. A *symbol-table entry* is a PostScript dictionary describing a source-language identifier: a variable, procedure, type, or constant. The compiler generates names beginning with S or T to refer to symbols and types. The symbol-table entry for `i`, on line 6 of the sample program shown in Figure 9, is associated with the generated name `S8`:

```
/S8 <<
  /name (i)
  /type 4 LccType
  /sourcefile (fib.c) /sourcey 6 /sourcex 8
  /kind (variable)
```

```
  /where 29 'r' Absolute
  /uplink S7
>> def
```

Slashes precede literal names, and parenthesis delimit literal strings. Brackets (`<<`···`>>`) surround code that constructs dictionaries; within each dictionary, names preceded by slashes are associated with the values that follow. `name`, `type`, `sourcefile`, `sourcey`, `sourcex`, `kind`, and `uplink` appear in all symbol-table entries; `where` appears only in entries for variables and procedures. The value associated with `where` represents `i`'s location and is computed when the symbol table is interpreted. `Absolute` is a PostScript operator that calls the Modula-3 procedure `Location.Absolute`, described in Section 3.2. "`29 'r' Absolute`" evaluates to offset 29 in space `'r'` of an abstract memory, i.e., integer register 29.

The code `4 LccType` fetches the type dictionary for one of `lcc`'s predefined types, `int`. The dictionary is

```
/T4 <<
  /decl (int %s)
  /printer {PI}
  ...
>> def
```

`ldb` loads the predefined type dictionaries when a target architecture is selected; different architectures use different definitions. Using a single copy of the predefined types avoids repeating their definitions in each symbol table.

`ldb` proper expects source-level type dictionaries like `T4` to contain keys `decl`, associated with a string used to declare variables of the type, and `printer`, associated with a PostScript procedure used to print values of the type. The printing procedure has access not just to an abstract memory and a location therein, but also to the type dictionary. `lcc` uses a single procedure, `PI`, to print values of all signed integer types except `char`. The ellipsis stands for information not used by `ldb` proper, but by the printing procedure or by PostScript code that supports expression evaluation. The implementation of `PI` is described in Section 4.4, expression evaluation in Chapter 5.

In symbol-table entries, the key `uplink` is associated with the entry for the symbol's parent. `i`'s `uplink` value is the symbol-table entry for `a` (Figure 9, line 2), which is associated with the generated name `S7`. The following excerpt from `fib`'s symbol table shows how the `uplink` keys form the tree shown in Figure 10.

```
  /S6 << /name (n) /uplink null ... >> def
  /S7 << /name (a) /uplink S6   ... >> def
  /S8 << /name (i) /uplink S7   ... >> def
  /S9 << /name (j) /uplink S7   ... >> def
```

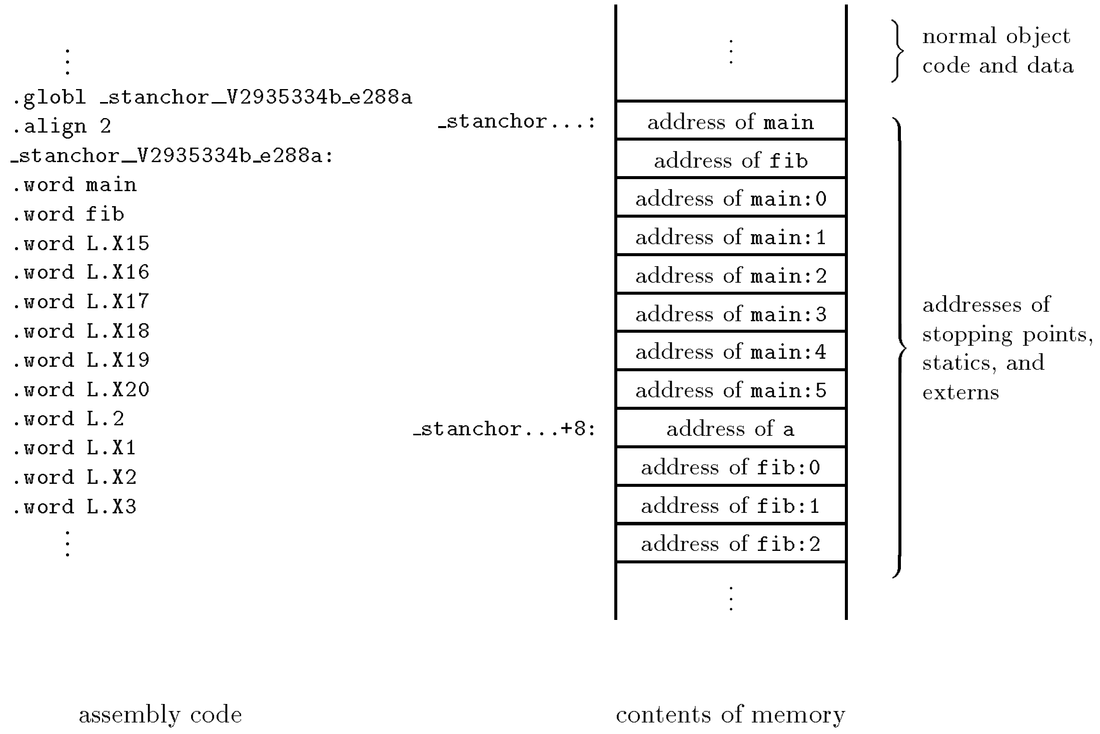


Figure 11: Use of anchor symbols.

Within `fib`, `a` is declared `static`, so its location is not determined until link time. Most linkers provide location information by relocating symbol-table entries stored in machine-dependent format in object files. Using PostScript avoids the retargeting burden of generating and reading such formats, but makes it impossible to use a standard linker's symbol-relocation ability to provide locations. A linker could be modified either to generate appropriate PostScript or to transform the PostScript emitted by the compiler, but unless a retargetable linker were used, such modification would have to be repeated for each new target. There is an alternative; every linker already relocates references to top-level symbols whenever such references appear in memory. I have therefore modified `lcc` to place references to labels and variables at the end of its assembly-language output, following a distinguished symbol called the *anchor symbol*. These references make programs about 10% larger. If the locations of the anchor symbols are known, all other locations can be found by indirection with respect to the appropriate anchor symbol. Figure 11 shows how an anchor symbol is used to find the location of the static variable `a` from line 2 of Figure 9. The name of the anchor symbol is `_stanchor__V2935334b_e288a`, and the address of `a` is stored at the 8th word following the anchor symbol.

`a`'s symbol-table entry, associated with `S7`, shows PostScript that defines a location relative to an anchor symbol. `a`'s type dictionary is associated with `T5`.

```
/T5 <<
  /decl (int %s[20])
  /printer {ARRAY}
  ...
>> def
```

```
/S7 <<
  /name (a)
  /type T5
  /sourcefile (fib.c) /sourcey 2 /sourcex 13
  /kind (variable)
  /uplink S6
  /where {(_stanchor__V2935334b_e288a) 8 LazyData}
>> def
```

Associated with `where` is a procedure interpreted at debug time; it computes `a`'s location by calling `LazyData`. `LazyData` gets the location of the anchor symbol, `_stanchor__V2935334b_e288a`, from `ldb`'s linker state and fetches the address of `a` from the 8th word following that location, as shown in Figure 11.

A symbol-table entry for a procedure holds the source and object-code locations and the associated symbols of the stopping points of that procedure. It also holds the entry for the procedure's last formal parameter; other formal parameters can be reached by following `uplink` keys. The procedure itself has a null `uplink` because it is a top-level symbol, and `uplink` is used to look up local symbols only. Stopping-point information is associated with the key `loci`, and the last formal is associated with the key `formals`. `fib`'s symbol-table entry is associated with `S5`:

```
/LS5 [ [ S6  S7  S7  S7  S8  S8  S8  S8  S9  S9  S9  S9  S7  S7 ]
         ⟨source and object-code locations⟩
     ] def
/S5 << /name (fib) /uplink null /loci LS5 /formals S6 ... >> def
```

Square brackets (`[`···`]`) delimit PostScript arrays. For performance reasons, stopping-point information is stored in an array of two elements. The first element, shown above, is an array containing the symbols associated with the stopping points. The associations are those shown in Figure 10; for example, `fib:7` is associated with `S8`, the symbol-table entry for `i`. The second element, shown in Section 4.5, contains the source and object-code locations of the stopping points. The symbol-table entry for a procedure also contains register-save information, as described in Section 8.2.

Access to symbols and types is provided by top-level dictionaries, which can describe a single compilation unit or any combination of compilation units, including an entire program. A top-level dictionary contains an array of symbol-table entries for procedures, a dictionary associating external symbol names with their symbol-table entries, a dictionary associating file names with arrays of symbol-table entries for the procedures defined in the files, and an array of the names of all anchor symbols used. The anchor-symbol names are compared with the anchor-symbol names in the linker

```
⟨code to build a full top-level dictionary⟩≡
  <<
    /procs [ S1 S6 ]
    /externs <<
       /main S1
       /fib  S6
    >>
    /sourcemap <<
      /fib.c [ [ S1 S6 ] ]
    >>
    /anchors [/_stanchor__V2935334b_e288a]
    /architecture (sparc)
  >>
```

Figure 12: Top-level dictionary for `fib.c`.

```
<<
  /anchormap <<
    /_stanchor__V2935334b_e288a
         16#000023d8
    ...
  >>
  /proctable [
    16#00002270 (_fib)
    16#00002374 (_main)
    ...
  ]
  /symtab ⟨top-level dictionary⟩
>>
```

Figure 13: Linker table for the program `fib`.

table (see Figure 13) to ensure that the top-level dictionary matches the object code. A top-level dictionary also contains the name of the architecture for which the program was compiled. `ldb` uses this name to look up a Modula-3 object containing machine-dependent code and data. Assuming that `S1` and `S6` represent the symbol-table entries for procedures `main` and `fib`, the top-level dictionary for `fib.c` can be built by the code shown in Figure 12.

The linker table contains the program's top-level dictionary, a dictionary associating the names of anchor symbols with their addresses, and an array of (address, name) pairs for each procedure in the program. `fib`'s linker table is shown in Figure 13. `anchormap` provides the information needed to look up the locations of anchor symbols by name. Binary search of `proctable` is used to find procedures given a program-counter value. `symtab`, the program's top-level dictionary, is returned to `ldb` when it requests the symbol table.

Rather than modify linkers to generate PostScript, I have arranged for `lcc`'s compiler driver to build linker tables like the one in Figure 13. After linking a program, the driver uses the Unix program `nm` to find the locations of anchor symbols and procedures. Anchor symbols are identified by their names, which begin with `_stanchor__`. All procedures are in the text segment, so it is sufficient to include the locations of all symbols in the text segment. Superfluous symbols do not matter as long as they do not occur in the middle of procedures.

4.2 Using PostScript symbol tables

The contents and structure of linker tables and top-level dictionaries are determined by what `ldb` does with the information. Debugging information is voluminous, and a typical debugging session uses only a small part of it. For good performance, it is best to read only what is needed. `ldb`

Operation	Steps in `ldb`'s implementation
find current stopping point	**L** program counter → procedure address **T** procedure address → symbol-table entry **P** binary search list of stopping points
resolve a name	**S** follow parent pointers from current stopping point (locals) **P** search table of names private to current compilation **T** search table of global names
show user where stopped in source	**S** source location of current stopping point
set breakpoint by source location	**T** source-file name → array of procedures **P** enumerate stopping points, sort by source coordinate find nearest stopping point by binary search **S** set breakpoint at stopping point's object-code location
find locations bound by linker	compiler plants references in memory near distinguished symbols post-link pass locates distinguished symbols **L** other locations fetched relative to distinguished symbols

Sources of information: **L** linker table
T top-level dictionary
P procedure's symbol-table entry
S stopping point

Table 1: Debugger operations that need symbol-table information.

builds its internal data structures lazily, working with one compilation unit, source file, procedure, or symbol at a time.

Table 1 shows what debugger operations need information from the compiler or linker. The right-hand side summarizes `ldb`'s implementation of these operations, indicating what parts of the PostScript symbol table are used at each step. **L** indicates use of the linker table, **T** the top-level dictionary, **P** the symbol-table entry for a procedure, and **S** a single stopping point.

`ldb` uses two binary searches to find which stopping point corresponds to the current program counter. It searches the linker table's `proctable` for the address of the procedure containing the stopping point. The procedure's address is used to look up its symbol-table entry in a hash table. `ldb` builds the table from the top-level dictionary's `procs`, which lists the symbol-table entries of all the procedures in the top-level dictionary. Each entry contains an address. Having found the procedure, `ldb` searches its stopping points to find the one nearest the program counter. A single-level search would require the addresses of all stopping points, which would mean touching every procedure's symbol-table entry. Using this two-level search means that only one procedure's symbol-table entry is needed. It also reduces the size of the global table by a large factor; even `fib`, a small procedure, has 14 stopping points. Other debuggers use similar two-level search techniques (Linton 1990; Russell 1992).

`ldb`'s name-resolution algorithm has been discussed above. The current stopping point provides access to the local symbols visible from that point, and the current procedure provides access to symbols private to its compilation unit. Global symbols are looked up in the top level dictionary's `externs`.

The array of stopping points defines the relation between source and object-code locations. Given an object-code location, `ldb` can find the corresponding source location quickly because the linker table and symbol-table entries contain arrays of procedures and stopping points that are sorted by object-code location. The PostScript tables provide less support for the inverse problem, so it is more expensive to solve. Again a search is made in two stages. The first stage identifies an array of procedures that contain code from the source file in question. This information is provided directly by top-level dictionary's `sourcemap`; it requires only a hash-table lookup. Preparing for the second stage is expensive; `ldb` examines all of the stopping points of all procedures from that file, building a list of such points that is sorted by source coordinate. For a source file of 1000 lines, this process can take a second or two. Once the list is prepared, it is saved and re-used for later searches; `ldb` finds the stopping point of interest by binary search. The initial delay could be reduced by associating procedures not just with a file name but with a range of source locations within that file, but the resulting implementation would be more complicated. Other debuggers use different strategies; for example, `dbx` keeps tables of files and line numbers sorted by object-code location, then does linear searches through these tables (Linton 1990). `ups` identifies the function containing the source location, then does a linear search through a set of line numbers (Russell 1992).

`ldb` uses the anchor-symbol technique to find locations bound by the linker; the steps therein are summarized in Table 1. The locations of the anchor symbols are provided by the linker table's `anchormap`; the PostScript procedures that compute the locations appear in symbol-table entries. Other debuggers use symbol tables in machine-dependent formats that appear in object files, and linkers relocate references in these symbol tables.

4.3 Representing symbols and types in Modula-3

Internally, `ldb` uses Modula-3 objects to represent symbols and types. These objects provide the same information as the PostScript representations, but with quicker access. Every access to a PostScript object requires a run-time type check, but access to Modula-3 objects requires only a compile-time type check. `ldb` performs the run-time type check at most once, when converting a symbol or type from the PostScript to the Modula-3 representation.

Symbols are represented internally using objects of type `Symbol.T`, of which there are four different subtypes to represent the procedures, variables, types, and constants. No symbol is converted from PostScript dictionary to `Symbol.T` until the `Symbol.T` is needed. When the conversion is done, the result is saved back in the PostScript dictionary, associated with the key `*CACHE*`. Saving the result avoids having to repeat the conversion the next time `ldb` needs the internal form of that dictionary. To make the association possible, `Symbol.T` is defined to be a subtype of PostScript object.

Some symbols contain locations, which are represented in PostScript either directly by objects of type `Memory.Location` or by procedures that use the anchor-symbol technique. When converting a location from PostScript to Modula-3, `ldb` interprets the PostScript object that represents it. A simple PostScript object like `Memory.Location` is executed by pushing it on the operand stack, and procedures that use the anchor-symbol technique compute a location and leave it on the stack, so in either case the result is that the location winds up on the stack, and `ldb` pops it off and saves it in the Modula-3 representation. `ldb` uses the location to overwrite the PostScript object in the original symbol-table entry, so that PostScript procedures using the symbol need not repeat the computation.

Locations in procedures, including the locations of stopping points, are assumed to be in the code space. `ldb`'s Modula-3 representation does not use a full `Memory.Location` for such locations; instead it uses just the offset in the code space. This representation uses less memory and speeds searches because integer offsets can be compared more quickly than full `Memory.Location`s.

`ldb` assumes that there is a symbol-table entry for every procedure in a program, and to satisfy that assumption it builds "dummy" entries for procedures not compiled with `lcc`. Such dummy entries lack information about stopping points and local variables. On some targets, machine-dependent stack-walking code adds machine-dependent data, like register-save information, as described in Section 8.2.

The contents of a type dictionary are determined by what `ldb` needs to do with source-language types and values. As described in Chapter 5, `ldb` relies on a server for expression evaluation; the only thing the debugger itself does with source-language values is print them. As mentioned above, printing requires only a location and a printing procedure. Locations of values are supplied either by the compiler, as with constants and variables, or by the expression-evaluation server, as with values that are the results of expressions. The compiler supplies printing procedures with types. It is useful to print the type of a variable or expression as well as its value, as shown in Chapter 2.

```
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
```

To print "`short n = 12`" instead of just "`n = 12`", `ldb` uses a string supplied with the type, which is also stored in the type dictionary by the compiler.

4.4 Printing values

When a PostScript printing procedure is called, it finds on the stack an abstract memory containing the value, the location of the value in the abstract memory, and the type dictionary describing the value's type. The compiler writer can use the type dictionary to store extra information for the printing procedure to use. This technique, a poor man's object-oriented PostScript, can simplify the compiler writer's job. For example, `lcc` does not define a different printing procedure for each C type; instead, it defines one procedure for each type *constructor*. `lcc` puts information in the type dictionaries so the print methods can distinguish different types that use the same constructor, e.g., to distinguish an array of 1024 integers from an array of 10 doubles. `lcc` need not generate specialized procedures on the fly; it uses predefined printing procedures and specializes them by the information in the type dictionaries.

Information in the type dictionary also helps the basic types to share print methods. Although the `lcc` front end supports eight basic types (`float`, `double`, `unsigned`, `unsigned short`, `unsigned char`, `int`, `short`, and `char`), there are only four basic print methods. All four use a `Memory.Type` stored by the compiler to fetch the value from memory. Three of the methods differ only in the way values are printed: in decimal, in hexadecimal, or as a character literal. They are used for printing the floating-point types, the unsigned types, and `char` respectively. The fourth method, which is used for signed integer types other than `char`, sign-extends the value before printing it in decimal.

`lcc` emits a call to `PF` to print values of floating-point types. `PF` uses one value from the type dictionary; the key `mtype` is associated with the `Memory.Type` used to fetch the value. `PF` is

```
/PF { /mtype get Memory.Fetch Put } def
```

`get` gets the `Memory.Type`, after which the stack holds abstract memory, location, and `Memory.Type`, the arguments needed by `Memory.Fetch`. `Put` writes a value to standard output. The printing procedures for unsigned integers and for characters are similar. Printing signed integers requires a second access to the type dictionary; the key `bitsize` is associated with the number of bits in the integer, and it is used to do the sign extension.

```
/PI { dup 4 1 roll /mtype get Memory.Fetch
      exch /bitsize get SignExtend Put } def
```

"`dup 4 1 roll`" saves a copy of the type dictionary before extracting `mtype` and fetching the value. `SignExtend` is a procedure implemented using the standard PostScript operators for bit manipulations. The following dictionary, which is usable with `PI`, describes the predefined type `int` on the SPARC.

```
/T4 <<
  /decl (int %s)
  /printer {PI}
  /mtype Memory.Type.I32
  /bitsize 32
  ...
>> def
```

Not all of C's type constructors are nullary. The printing procedures for non-nullary constructors call other print procedures associated with the key `printer` in type dictionaries. For example, the

```
1  /ARRAY {
2    16 dict begin
3      /&type exch def
4      /&loc exch def
5      /&m exch def
6      /&elemtype &type /type get UnCache def
7      /&arraysize &type /size get def
8      /&elemsize &elemtype /size get def
9      /&limit $ArrayLimit &elemsize mul def
10     ({) Put 0 Begin
11     0 &elemsize &arraysize 1 sub
12     { dup 0 ne { (, ) Put 0 Break } if
13       dup &limit ge { (...) Put pop exit } if
14       &m &loc 3 -1 roll Shifted &elemtype dup /printer get exec
15     } for
16     (}) Put End
17     end
18 } def
```

Figure 14: Printing procedure for arrays.

procedure that prints arrays expects to find the element type's dictionary associated with the key `type` in the array type's dictionary. `lcc` also associates the size of each type with the key `size` in the type dictionary. The size of the array type is divided by the size of the element type to determine the number of elements. The representations of types and sizes within the type dictionaries were chosen to match `lcc`'s representations, but they also simplify the printing procedures. Matching representations used in PostScript to those used within `lcc` makes it easier to re-use `lcc` as an expression-evaluation server, as described in Section 5.3.1.

To print arrays, `lcc` emits calls to the PostScript procedure `ARRAY`, which is shown in Figure 14. It begins on line 2 by creating a temporary dictionary to hold local variables. `&type` is associated with the array type's dictionary, `&loc` with the location of the array, and `&m` with the abstract memory. `&elemtype` is associated with the element type's dictionary; `UnCache` is discussed in Section 4.5. `&arraysize` is associated with the size of the array, `&elemsize` with the size of one element. `ldb` does not print all the elements of very large arrays; instead it prints only the first `$ArrayLimit` elements. `$ArrayLimit` is initially 100, but it can be changed by the user at debug time. `&limit` is associated with the offset of the "limit element;" line 13 stops printing elements if the limit element is reached.

Here, reproduced from Chapter 2, is sample output from `ARRAY`, which prints the elements of an array in braces, separated by commas.

```
int a[20] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
             377, 610, 987, 1597, 2584, 4181, 6765};
```

The placement of elements on the page is computed by a prettyprinter supplied with the Modula-3 library (Kalsow and Muller 1992). The prettyprinter's procedures are available as PostScript operators, and the printing procedures use these operators to print text. Calls to the prettyprinting operators also specify at what points text may be split across lines and what fragments of text should be grouped together at the same level of indentation. For example, a line break is permitted following any element of the array, but the elements are grouped together, forcing the element `377` to be indented to the same level as the first element. Oppen (1980) describes similar operators.

Lines 10–16 of Figure 14 contain the calls to the prettyprinter that do the printing. Lines 10 and 16 print the opening and closing braces. `Begin` and `End` cause the grouping of the array elements; the 0 argument to `Begin` specifies the relative indentation of later elements with respect to the first. The `for` loop on lines 11 and 15 loops through the *offsets* of the elements from the beginning of the array. Line 12 prints a comma before every element but the first. "`0 Break`" on that line tells the prettyprinter that a line break is possible after each comma, and that subsequent lines are not to be indented with respect to the first line. Line 13 checks to see if the limit element has been reached; if so, it prints an ellipsis and exits from the element-printing loop. Line 14 prints the element. The offset of the element is on the stack, so after "`&m &loc 3 -1 roll`" the stack holds the abstract memory, the location of the array, and the offset. `Shifted` is the PostScript version of `Location.Shifted`, described in Section 3.2; it converts the location and offset to a new location. `&elemtype` puts the element type's dictionary on the stack, and "`dup /printer get exec`" fetches the printing procedure from that dictionary and executes it. After the array is printed, line 17 pops the dictionary stack, removing the dictionary holding the local variables.

The array example shows how the policies that determine the appearance of structured data are embodied in calls to the prettyprinter. One policy is that as many array elements are packed onto a line as will fit, but structure members always appear on separate lines unless the entire structure fits on one line. The policy is implemented by having the structure-printing procedure call `UnitedBreak` after each member, whereas `ARRAY` calls `Break` after each element. Printing policies can be changed by changing the implementations of the printing procedures.

Some of the printing procedures, like `ARRAY`, use the values of PostScript variables to alter their behavior. By default, `ldb` prints only the first 100 elements of arrays and the first 30 characters of strings. Users can change these values dynamically by redefining the PostScript variables `$ArrayLimit` and `$StringLimit`. By default, `ldb` does not dereference pointers when printing them. Users can make it dereference pointers by setting the PostScript variable `$followpointers` to 1. Larger values cause the printing procedure to follow chains of non-null pointers of length at most `$followpointers`.

A few types have special printing procedures. A `char *` is printed as a string literal, not as a pointer to a character. The function-pointer procedure prints the name of the function as well as its address. Finally, the expression server uses a special procedure to print character arrays as string literals, because it is more convenient to synthesize a tiny type dictionary containing this procedure than to construct a general type dictionary for an array of characters.

4.5 Performance enhancements

It is expensive to interpret the PostScript that builds a top-level dictionary. For example, it takes about 1.5 seconds to build the top-level dictionary for `lcc`'s lexical-analysis module, which is about 800 lines of C and includes about 1200 non-blank lines from header files. Building the top-level dictionary for the MIPS version of `lcc`, about 16,000 lines of C and header files, takes 52 seconds. `ldb` uses lazy evaluation to defer the interpretation of PostScript and thereby to reduce the delays seen by users. Lazy reading of PostScript files defers reading the PostScript that builds a top-level dictionary until that dictionary is used. Lazy transformation of type and symbol dictionaries defers lexical analysis and interpretation of some of the PostScript associated with a symbol and type dictionaries until those dictionaries are used. These techniques complicate the representation of top-level dictionaries, symbols, and types.

A *lazy version* of a top-level dictionary is used to defer reading and interpreting the PostScript that builds the full top-level dictionary. `ldb` builds a lazy version of `lcc`'s top-level dictionary in 5 seconds, not 52. In place of the usual information provided by a top-level dictionary, the lazy version provides the pathname of a file that holds the full information. In Table 1, the operations that the top-level dictionary must provide are marked with a **T**. They are to look up an external symbol by name, to look up a procedure by address, and to map a source-file name to an array of procedures.

The lazy version of a symbol-table entry for an external symbol has one key, `lazy`, which is associated with the pathname of a file that can be read to build a full top-level dictionary containing that external symbol. The lazy version of a procedure symbol-table entry, appearing in a top-level dictionary's `procs` array, has an additional key, `where`, used to compute the procedure's location in the usual way. The location is needed so the procedure can be looked up by address, a part of finding the current stopping point as shown in Table 1. In both cases the symbol-table entry can be identified as a lazy version by the presence of the key `lazy`.

It is not quite true that the top-level dictionary's `sourcemap` maps a source-file name to an array of procedures. To support lazy reading of PostScript files, the `sourcemap` maps a source-file name to a meta-array. Each element of the meta-array is either an array of procedures, indicating a full version, or a pathname, indicating a lazy version.

Figure 15 shows the code that builds a lazy version of `fib`'s top-level dictionary. The pathname used to read the file is not compiled into the dictionary; it appears on the operand stack. A single symbol-table entry is re-used for all the `externs`, because the lazy versions are all identical. Each

⟨*code to build a lazy top-level dictionary*⟩≡
```
Architecture.sparc begin
10 dict begin
/stabpathname exch def
/lazydict << /lazy stabpathname >> def
<<
  /anchors [ (__stanchor__V2935334b_e288a) ]
  /architecture (sparc)
  /externs <<
        /main lazydict
        /fib lazydict
  >>
  /procs [
    << /lazy stabpathname
       /where {(__stanchor__V2935334b_e288a) 0 LazyCode}>>
    << /lazy stabpathname
       /where {(__stanchor__V2935334b_e288a) 1 LazyCode}>>
  ]
  /sourcemap <<
    (fib.c) [stabpathname]
  >>
>>
end % temporary dict
end % Architecture.sparc
```

Figure 15: Building a lazy version of `fib`'s top-level dictionary.

procedure in `procs` must have its own lazy symbol-table entry, because each procedure has a different location.

When `ldb` encounters a `lazy` key in a symbol-table entry or a pathname in a meta-array, it reads the named file and builds a new top-level dictionary, which it merges into the top-level dictionary for the entire program. It stores the new versions of symbol-table entries for external symbols in the top-level dictionary's `externs`, overwriting the lazy versions. It stores the new procedure entries in its internal hash table for lookup by address. The new source map is merged in by appending the new meta-array to the existing meta-array, and by replacing any occurrences of the pathname with empty arrays. All merging is done by PostScript code, except storing procedures in the internal hash table.

`lcc` cuts clutter by writing a single PostScript file from which `ldb` can build either a lazy or a full top-level dictionary. To read such a file, `ldb` sets the variable `$lazy` to `true` or `false`, depending on whether a lazy or full dictionary is desired. It puts the file's pathname on the stack (so it can be inserted into lazy dictionaries if necessary) and interprets the file.

The PostScript file written by `lcc` has the form

```
$lazy {
  ( ⟨code to build a lazy top-level dictionary⟩ )
  cvx exec currentfile closefile
} if
⟨code to build a full top-level dictionary⟩
```

If `$lazy` is false, the code to build the lazy top-level dictionary is skipped quickly, because it is a string. Otherwise, the string is executed and the file closed, preventing `ldb` from reading more PostScript.

Another lazy-evaluation technique is used to defer some of the computation needed to build a single symbol-table entry or type dictionary. Many of the values in a symbol-table entry are simple and immutable: integers, strings, locations, and so on. Such values appear in the PostScript as literals. The literals and the keys associated with them can be put into a string, which need not be interpreted until the values are needed. The string is associated with the key `*CACHE*` in the dictionary. This technique reduces the time that `ldb` needs to build top-level dictionaries by about 35%. In the symbol-table entry for `i` in the program `fib`, every value except `uplink` is hidden in the string.

```
/S8 << /uplink S7
       /*CACHE* (
       /type 4 LccType
       /name (i)
       /sourcefile (fib.c)
       /sourcey 6
       /sourcex 8
       /kind (variable)
       /where 29 'r' Absolute
 ) >> def
```

Even the type is hidden because it is a predefined type, `int`. The version after conversion, without `*CACHE*`, appears on page 37.

`ldb` must perform two transformations to use a symbol or type dictionary. It must first interpret the string associated with `*CACHE*`, inserting the resulting key-value pairs into the dictionary, and then convert the dictionary to a `Symbol.T` or `Expression.Type`. To avoid repeating this conversion, the key `*CACHE*` is then associated with the result of the conversion. Sometimes PostScript code needs to use the dictionary before it has been converted. If so, it interprets the string associated with `*CACHE*` and stores the results in the dictionary. To avoid repeating this step, it then associates the dictionary itself with `*CACHE*`. The PostScript function `UnCache` does the job, finding on the stack a dictionary in which some information may be hidden in a `*CACHE*` string, and leaving a

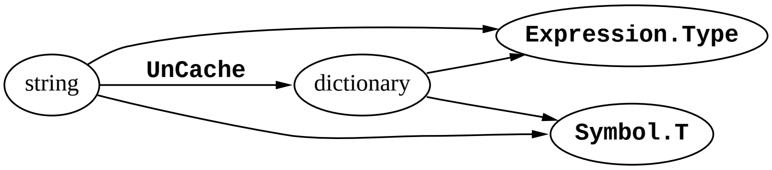


Figure 16: Types of values associated with key `*CACHE*` in a symbol or type dictionary.

version of the same dictionary in which all information is associated directly with the appropriate keys. Figure 16 shows all the possible states of a symbol or type dictionary according to the type of the value associated with `*CACHE*`.

Information about the source and object-code locations of stopping points is also kept in a string, leading to the two-part representation described in Section 4.1. The locations of `fib`'s stopping points are represented as follows:

⟨*source and object-code locations*⟩≡
```
([ [ (fib.c)  1  16  {(_stanchor__V2935334b_e288a)  9 LazyCode} ]
   [ (fib.c)  4   6  {(_stanchor__V2935334b_e288a) 10 LazyCode} ]
     ⋮
])
```

Source locations are described by file name, line, and column; object-code locations are computed using the anchor-symbol technique.

4.6 Generating PostScript symbol tables

Production versions of `lcc` generate symbol-table "stabs" for `dbx` and `gdb`. Stab generation is isolated behind an 8-function interface internal to `lcc`'s front end. `lcc` generates PostScript for `ldb` by using different implementations of these functions, minimizing changes to the rest of `lcc`. The interface was extended to support `ldb`; the code supporting `dbx` and the code supporting `ldb` implement overlapping subsets of the functions.

An `up` field was added to `lcc`'s symbol type to represent the trees described in Section 4. `lcc` sets the `up` fields by maintaining in each scope a pointer to the most recently declared symbol in that scope. At the end of the compilation, the `up` field in the global scope provides access to all of the global symbols, both `extern` and `static`. `lcc` saves information during compilation, but does not emit PostScript until the entire program has been compiled. This strategy makes it easy to write a single PostScript file that contains both lazy and full top-level dictionaries without using temporary files.

Stopping points provide access to local symbols. A stopping point has source and object-code locations location, and it is associated with a symbol. These three items are not known simultaneously, only in pairs. `lcc` makes two passes over each function. Associated symbols are computed in the first pass, object-code locations in the second.

`lcc`'s first pass parses, type-checks, and build an intermediate representation. Every time it encounters a stopping point, it appends to a list the point's source location and the most recently declared symbol. After the function is parsed, this list is the only record of the relationship between the stopping point and the symbol. The second pass calls the back end to generate assembly code. During code generation, stopping points are associated with labels, which are used to compute their object-code locations using the anchor-symbol technique. This association is computed by the symbol-table code, not by the front end. After the compilation is finished, the symbol-table code combines the two lists of stopping-point information, making it possible to emit each point's source and object-code locations and associated symbol. If there are N stopping points, this code makes N^2 comparisons.

In addition to information about stopping points, the symbol-table code records register-save information for each function. To emit the PostScript symbol table, it also needs the top-level symbols. The front end supplies the last-declared top-level symbol; the others are linked by `uplink` fields. All local symbols are accessible from the symbols associated with stopping points. `lcc` begins by emitting definitions of the symbols and types defined in the compilation unit. `lcc`'s type and symbol data form a directed graph; the representations of recursive types create cycles in the graph. `lcc` traverses this graph and emits PostScript in a single pass. The code that emits PostScript need not consider the meanings of the cycles, but it must choose an order in which to define dictionaries and identify arcs that point from earlier to later dictionaries. Such arcs are not added to the graph until all the symbol and type dictionaries (the nodes) are defined. They are added by adding the appropriate key-value pairs to existing dictionaries. For example, the following code adds to `T2` an arc to `S4`, which is defined after `T2`:

```
T2 /symbol S4 put
```

After emitting the definitions of the symbol and type dictionaries and the forward arcs `lcc` emits the PostScript code that builds the top-level dictionary. Such code is shown in Figure 12.

Work with `dbx` suggests that 80–90% of C debugging information is redundant. The source of redundancy is C header files; debugging symbols for a header file appear in every compilation unit that includes the file. `lcc` reduces duplication by omitting definitions of symbols not defined in the compilation unit. If passed the `-SE` flag, it includes these definitions, making the symbol table larger. This flag is useful if users want to examine variables or call procedures defined in library code not compiled with `lcc`. The increase in symbol-table size varies widely with the program. The table for `lcc`, a large program that uses few library procedures, is only 5% larger with external symbols included. The table for `agrep`, a 6,000-line string matcher (Wu and Manber 1992), is 46% larger. The table for `notangle`, a 600-line macro processor (Ramsey 1992), is 3.5 times larger.

`lcc`'s standard implementation of the symbol-table functions, which supports `dbx`, is about 300 lines of C. The support for `ldb` is about 1,000 lines. One source of the difference is that the interface is designed to support `dbx`; all of the standard implementations simply emit information and return. The front end does the work of making sure the functions are called at the right times. The `ldb` implementations, by contrast, accumulate information in linked data structures; they emit no PostScript until the end of the compilation. If `lcc` supported only `ldb`, the rest of the front end could be simplified by eliminating support for `dbx`. `lcc` must also store enough information in the PostScript to enable the expression-evaluation server to reconstruct `lcc`'s data structures, as described in Chapter 5. It is not enough to store extra information at each symbol or type node; the PostScript must also represent the links between nodes, which form a graph containing cycles. Representing cyclic structures in PostScript makes `lcc`'s symbol-table functions more complex.

Changes made outside of symbol-table generation were modest. The front end required about 25 lines of changes to maintain the tree of symbols and to associate symbols with stopping points. Changes to the stab interface affected another 10 lines. About 20 lines were added to the back end to provide register-save information; those lines include code for all four targets.

4.7 Discussion

Linking decisions may sometimes be postponed until run time. For example, some environments use a dynamic linker that links all programs into a single address space (Swinehart *et al.* 1986). Some versions of Unix support shared libraries in which the locations of library procedures are not determined until such procedures are called (Sun 1990b). `ldb` could support such linkers by basing its linker state on linker-dependent, run-time information, not the static linker table it now uses. More ambitious linkers might take over some of the work done by the compiler, for example determining the locations of all variables at link time (Wall 1992). Supporting such linkers might require extensions to the existing linker state, not just a re-implementation, although the existing implementation can be used if the linker can be modified to generate both a linker table and a top-level dictionary. A similar technique is used to support debugging in the PCedar environment; the linker dynamically generates a synthetic `a.out` file containing `dbx` debugging symbols that represent the state of the dynamically linked program (Weiser, Demers, and Hauser 1989).

`ldb`'s symbol tables might change if the linker could be modified. One possibility would be to put the debugging information in the same file as the object code as current Unix linkers do for `dbx`. This scheme has the virtue of simplicity; object code and debugging information are guaranteed to match, and the debugging information cannot easily be misplaced. The scheme has an important disadvantage; debugging information is voluminous, and reading and relocating it slows down the linker. If `ldb` is compiled for use with `dbx`, the relocation and debugging symbols occupy twice as much space as the object code. If reliable configuration management is available (Leblang and Chase 1984), debugging information can safely be stored in separate files.

Despite current practice, debugging information could be put in the object code without requiring special support in the linker. Linkers already relocate references in memory; it is not necessary to define a second entity, "debugging symbols," in which references must also be relocated. Given the large address spaces available today, debugging information can be put in the target address space as initialized data. If a separate section is reserved for such information, it can be allocated a single contiguous fragment of the address space, and that fragment need not be mapped to physical memory unless a program is being debugged. `ldb` could get the information from the running process, but it would be more efficient to get it from the executable file itself. The executable file is sufficient because debugging information is read-only. If a machine-independent linker were used, such access could be done without machine-dependent code (Fraser and Hanson 1982). Even if standard, machine-dependent formats were used, the gain in efficiency might warrant the extra retargeting effort.

One potential benefit of putting special support in the linker is that debugging information could be encoded more compactly. For example, the linker could store the distance between stopping points instead of the addresses of stopping points, reducing the amount of storage required by at least a factor of four.

`ldb`'s current implementation would benefit from a linker that generated linker tables directly, eliminating the post-link pass now used to find procedure and anchor symbols. Such a linker might also emit machine-dependent data to be included in procedures' symbol-table entries. This technique could eliminate the use of the MIPS run-time procedure table (see Section 8.2) to get procedures' frame sizes.

`ldb` assumes that the locations of variables do not change during the execution of the program. An earlier version provided for motion of a variable between registers and memory. The technique used was indirection in the location field of a variable's `Symbol.T`. Instead of being a location, it was a PostScript object that could be executed to produce a location. The object was executed every time the variable's location was desired, so a compiler that moved variables could have emitted a procedure that fetched the program counter and used it to compute the location. This code was never exercised because `lcc` does not move variables, and the indirection was removed. The removal changed only a few dozen lines of code, but it made the design much easier to explain because `ldb` already uses one level of indirection to compute the location. That indirection makes the anchor-symbol technique possible. To make `ldb` work with compilers that do move variables, the code would have to be reinstated. Moving variables is necessary to make best use of caller-save registers; variables may appear in different registers or in memory depending on the value of the program counter (Chow and Hennessy 1990). Such changing locations could be computed easily by

PostScript procedures, provided that the program counter were included in the abstract memory available to the PostScript.

If the state seen by the debugger is to be consistent with the source code, `lcc` must prevent instruction scheduling from moving instructions across stopping points. This restriction reduces the number of opportunities for filling delay slots. As a result, the code size increases by anywhere from 2% to 25%, because more delay slots must be filled with no-ops. The average increase is 12–15%. The penalty can be avoided only if the debugger can present the effects of interleaving instructions from different statements. Brooks, Hansen, and Simmons (1992) describe the design of one such debugger.

`ldb`'s classification of symbols into types, variables, procedures, and constants might not suffice for other languages. It is not clear how it would accommodate Modula-3 interfaces or Standard ML functors, for example. On the other hand, the existing classification is more complex than needed; variables and constants are treated nearly identically, and the distinction could be eliminated without changing much code.

A symbol-table format should be machine-independent, extensible, compact, and possible to read quickly and incrementally. PostScript symbol tables are machine-independent and extensible, but they consume lots of disk space and take a long time to build.

When `lcc` compiles itself on the SPARC, it compiles 22 files totalling 17,000 lines of C; each compilation includes 2,000 lines of header files. That generates 4.0MB of PostScript, but only 0.38MB of `dbx` stabs and linker symbols. It also adds 46KB to the initialized data to hold locations following anchor symbols. The relative cost of using PostScript is lower for `ldb`, which when compiled to C is 110,000 lines. `ldb`'s symbols take 18.6MB, and `dbx`'s symbols take 2.6MB. Comparing raw sizes is slightly unfair because the PostScript is encoded in ASCII whereas the stabs have a compact binary encoding. Compressing the PostScript files gives a plausible estimate of the space they would require given a binary encoding; if each PostScript file is compressed by Unix `compress` (Welch 1984), `lcc`'s files occupy 0.79M, still twice the size of the `dbx` stabs. This savings could be realized by using an I/O library that includes compression. The factor of two might be reduced further by reorganizing the representation to reduce redundancy, for example, organizing stopping points by source-file name so the name need not be repeated for each stopping point. The PostScript symbol tables also contain information that stabs do not, e.g., the data needed to reconstruct the compiler's symbol and type representation. The extra information makes it possible to re-use the compiler at debug time, rather than implementing a C interpreter as do other debuggers.

PostScript symbol tables cost more to generate than `dbx` stabs. On the SPARC, generating PostScript symbol tables for `lcc` triples compilation time; generating `dbx` stabs increases compilation time by 23%. I/O accounts for only 26% of the added time. Overhead is higher for `lcc` because `lcc` uses a large header file containing many type definitions that must appear in the symbol table. When `lcc` compiles `noweb`, a 1000-line literate-programming tool (Ramsey 1992), compilation time increases by 80%; the corresponding increase for stabs is 13%. I/O accounts for only 37% of the added time. On the MIPS, the overhead is lower; compilation takes 2.6 times longer for `lcc` and 41% longer for `noweb`; I/O accounts for 27% and 39% of the added time. Relatively more time is spent in the MIPS assembler than in the SPARC assembler, which accounts in part for the lower overhead on the MIPS. MIPS `lcc` does not generate `dbx` symbol-table information, so times cannot be compared to those for `dbx`. `lcc` is faster than many other C compilers (Fraser and Hanson 1991b), so the relative costs of generating PostScript symbol tables might be lower if a different compiler were used.

The savings realized by lazy top-level dictionaries varies with the size of the program. Savings are proportionally larger for large programs; the following table shows the startup times for `ldb` using both full and lazy top-level dictionaries. The times for `dbx` and `gdb` are also shown; even with lazy dictionaries, `ldb` takes much longer to start than these debuggers. Targets are shown on the top, debuggers on the left:

	`ldb`	`lcc`	`noweb`	no target
`ldb` (full dictionaries)	220	52	5.4	1.9
`ldb` (lazy dictionaries)	14	4.6	3.4	1.9
`dbx`	4.6	1.1	0.7	0.3
`gdb`	1.2	0.7	0.6	0.6

All times are elapsed times in seconds measured on a DEC 5000 model 240. The targets used in these measurements vary widely in size; `ldb` is 110,000 lines, `lcc` is 17,000 lines, and `noweb` is 600 lines. The lazy dictionaries save a factor of 15 on the largest target and a factor of 2 on the smallest.

It is difficult to assess precisely the cost of using PostScript because the cost varies with the target program, but some conclusions can be drawn anyway. The ASCII encoding of PostScript offers poor performance; the files take up too much space, and the costs of generation, output, input, and lexical analysis are significant. PostScript is linear; the interpreter reads and interprets one token at a time. It is therefore difficult to read PostScript incrementally. The lazy-evaluation techniques described in Section 4.5 help, but they put a greater burden on the compiler writer, and it's not clear how to get much better performance out of similar techniques.

Although the full generality of PostScript is useful in expression evaluation and in printing values, it is not needed to represent symbol tables. Performance could be improved by using a data format to represent symbol tables. The data format could have the same structure as `ldb`'s top-level dictionary, but it and the symbol-table entries could be encoded more compactly in a way that could be read incrementally and in a way that required only one allocation per symbol-table entry. The encoding should be managed so that the compiler can write information for one function at a time, instead of having to wait for the compilation to end before writing anything. Three features of the current representation are worth preserving. It can represent arbitrary PostScript objects, especially procedures. Symbol-table entries are extensible, so the compiler or debugger can insert whatever information is convenient, like the sizes and alignments of types or the frame sizes of procedures. The representation can be manipulated from PostScript, for example, to support expression evaluation.

A data format using a compact binary encoding of top-level dictionaries could use less disk space and provide fast, incremental access for the debugger using random access. If such a representation were treated by the interpreter as an ordinary PostScript dictionary, it would be possible to use the existing PostScript representation during development, when understanding and debugging are important, and to switch to the compact format for production use. A similar scheme within the debugger, treating a `Symbol.T` as a PostScript dictionary, would eliminate the conversion between PostScript and Modula-3 representations. Ordinary PostScript could continue to support printing and expression evaluation, where performance is not critical.

In a multipass program, the earlier passes must transmit information to the later passes. This information is often transmitted most efficiently in a somewhat machine-like language, as a set of instructions for the later pass; the later pass is then nothing but a special purpose interpretive routine, and the earlier pass is a special purpose "compiler." This philosophy of multipass operation may be characterized as telling *the later pass what to do, whenever possible, rather than simply presenting it with a lot of facts and asking it to* figure out *what to do.*

—Don Knuth (1973), page 198

Chapter 5

Expression Evaluation

A debugger must evaluate expressions that include variables in the target program, and it must make assignments to such variables. The problem is like compilation, and most debuggers use a structure like a compiler's to solve it. A common approach is to write a front end that scans and parses the expression, turning it into some intermediate form, then to interpret the intermediate form. The debugging symbols emitted by the original compiler provide symbol-table information. `dbx`, `gdb`, and `ups` use this approach. Each of these debuggers implements expression evaluation from scratch. One drawback is that the debugger and compiler are likely to implement different languages. `dbx` implements only a subset of C; for example, it omits assignment operators like `+=` (Adams and Muchnick 1986). The `ups` documentation lists some details of ANSI C that it gets wrong; for example, `ups` cannot correctly call procedures that return floating-point or structured values.

Another approach to expression evaluation uses the original compiler to translate the expression into machine code, places the code in the target address space, and executes it (Fritzson 1983). This approach guarantees that the debugger and compiler implement the same language, and it avoids interpretation. It requires closer integration of debugger, compiler, and target run-time system than the previous approach; the run-time system must allocate space for the generated code, the compiler must generate machine code instead of assembly code, and the generated code may have to be relocated. An incremental compiler has similar requirements.

`ldb` uses an intermediate approach to expression evaluation. It uses the same structure as the first approach, but it re-uses existing compiler code, yielding some of the benefits of the second approach without the additional demands on the compiler and run-time system. `ldb` uses an expression-evaluation server; `lcc`'s front end is used with a new back end that generates PostScript. The front end's symbol-table code is modified to get information from the debugger when it encounters an unknown symbol. The design is shown in Figure 17; the debugger provides the text of the expression, plus symbols and types, and the compiler returns PostScript. `ldb`'s PostScript symbol tables simplify

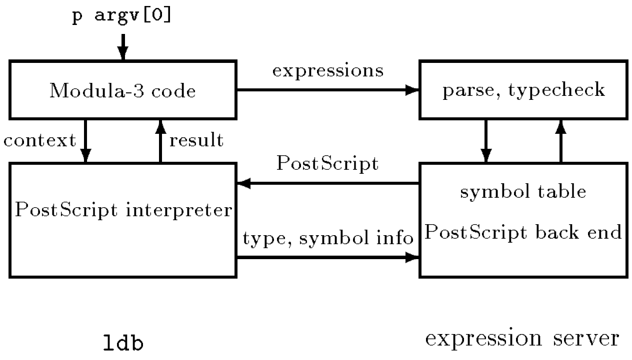


Figure 17: Communication paths between `ldb` and an expression server.

re-using the compiler front end; when the target program is compiled, the compiler emits a PostScript dictionary that is used to reconstruct its data structures when the program is debugged. The expression-evaluation server implements all C expressions, including all assignment operators and procedure calls and expressions with nontrivial control flow, e.g., conditional expressions and short-circuit Boolean operators.

Figure 17 illustrates `ldb`'s implementation as well as its design. The expression-evaluation server executes in a separate address space, so the original compiler's input, output, and memory allocation can be used even though these components are incompatible with the Modula-3 run-time system used in `ldb`. `ldb` communicates with the server via byte streams (Unix pipes).

Expression evaluation takes place in several steps, as shown in Figure 18. Only the first step, compilation, requires interaction with the expression-evaluation server. To compile an expression, `ldb` sends it to the server. The server attempts to parse and type-check the expression and to produce intermediate code. When it fails to find an identifier like `argv` in its symbol table, it does not print an error message and stop; its symbol-table code has been modified to send the PostScript "`(argv) ExpressionServer.lookup`" back to `ldb`. The PostScript procedure `ExpressionServer.lookup`, when interpreted by `ldb`, finds the PostScript dictionary representing `argv`'s symbol-table entry and sends information from that dictionary back to the expression server. The server's modified symbol-table code uses that information to reconstruct `argv`'s symbol-table entry on the fly, and it returns the newly created entry to the parser just as if the entry had always been present. The lower two arrows in Figure 17 show the server's PostScript request and the debugger's reply.

When the parser has built intermediate code, that code is passed to the PostScript back end, which sends `ldb` a PostScript procedure and a type dictionary. These results from the first step of Figure 18 are used in the subsequent steps. In the second step, evaluation, the procedure is

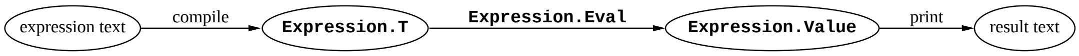


Figure 18: Steps in evaluating an expression.

interpreted, producing the location of the value of the expression. The type dictionary is used in the third step, printing, as described in Section 4.4.

This chapter describes the implementation of each step in expression evaluation, including the modifications that make it possible to use a variant of `lcc` as an expression-evaluation server.

5.1 Debugger end of the expression server

`ldb` takes an abstract view of expression evaluation. An object of type `Expression.Compiler` is used in the first step, which turns the text of the expression into an object of type `Expression.T`. The use of an expression-evaluation server to implement this step is hidden from most of the debugger. This section describes the `Expression.Compiler`, both at the abstract level and at the level needed to understand the implementation.

`ldb`'s internal representation of a compiled expression is a Modula-3 object of type `Expression.T`. As depicted in Figure 17, the debugger's only contribution to the compilation step is to provide symbols when they are requested by name. It does so by supplying an object of type `Scope.T`, which maps a name into `ldb`'s internal representation of a symbol, `Symbol.T`:

```
TYPE
  Scope.T = Displayed.T OBJECT METHODS
    lookup(name: TEXT) : Symbol.T;
  END;
```

The `lookup` method raises an exception if the symbol is not found.

The debugger treats the expression server as a black box. Its type is

```
TYPE
  Expression.Compiler = OBJECT METHODS
    compile(source: TEXT; scope: Scope.T; interp: Interp.T) : Expression.T;
  END;
```

Because an `Expression.T` contains a PostScript procedure, `ldb` supplies a PostScript interpreter, type `Interp.T`, that the `Expression.Compiler` can use to construct the procedure.

A sample invocation of an `Expression.Compiler` is

```
target.compiler.compile("argv[0]", focus.visible(), target.interp);
```

where `target` holds global information about the target, including an `Expression.Compiler` and a PostScript interpreter, and `focus` is the current focus. `focus.visible` returns a scope used to look up names; as described in Chapter 4, the lookup procedure walks up a tree of local symbols, then searches hash tables of top-level symbols.

The implementation of `Expression.Compiler` uses the PostScript interpreter to talk to the expression server. `Scope.T` and `Symbol.T` are subtypes of `Displayed.T`, which is a type of PostScript object, so they can be manipulated directly by the PostScript interpreter. The debugger's end of the interaction is simple. It begins by sending the text of the expression, suitably delimited, to the server, for example, "`: argv[0] ;`". The colon makes the server compile an expression into PostScript; there are other requests that make it print parts of its internal state. After sending the expression, the debugger puts the `Scope.T` and the streams to and from the server onto the PostScript operand stack and interprets the PostScript procedure `ExpressionServer.Serve`. `ExpressionServer.Serve` creates names for the scope and for the output stream to the server, so that other PostScript procedures can refer to them by name. Then, by applying "`cvx stopped`" to the stream from the server, it reads and interprets PostScript sent by the server, running until the server sends `stop`.

```
/ExpressionServer.Serve {
  /ExpressionServer.scope exch def
  /ExpressionServer.wr exch def
  cvx stopped not {(Expression-server EOF) Interp.Error} if
} def
```

The scope is bound to the name `ExpressionServer.scope` for the duration of the interaction with the server. The server must send a `stop` after the evaluation of each expression. If the pipe is closed without a `stop`, for example if the server crashes, `ExpressionServer.Serve` raises an exception. Having the server send PostScript to be evaluated simplifies the debugger, which need only ensure that the symbol and type information is accessible from PostScript; the code sent by the server does the rest of the job.

5.2 Evaluating and printing compiled expressions

As described in Section 4.4, printing a value requires an abstract memory containing the value, the location of the value, a type dictionary containing an appropriate printing procedure, and a PostScript interpreter. `ldb`'s internal representation of a value, `Expression.Value`, is a Modula-3 object containing this information, except it uses its internal representation of a type instead of a type dictionary. The print step shown in Figure 18 sets up the PostScript stack and interprets `print`, which extracts and interprets the printing procedure in the type dictionary:

```
/print { dup UnCache /printer get exec } def
```

`UnCache`, as described in Section 4.5, makes all the contents of the type dictionary accessible.

The middle step in Figure 18, evaluating a compiled expression to produce a value, is implemented by the Modula-3 procedure `Expression.Eval`. The compiled expression contains a type, but no abstract memory, and it contains a PostScript procedure instead of a location. The caller of `Expression.Eval` must supply an abstract memory, which `Expression.Eval` puts on the PostScript operand stack before interpreting the procedure. When interpreted, the PostScript procedure returns (on the operand stack) the location of the result of the expression. The procedure may have side effects on the state of the target, either by storing into the abstract memory or by calling procedures in the target. These side effects arise from assignment operators or procedure calls in the original expression.

5.3 Making `lcc` act as an expression server

Three kinds of changes make it possible to use `lcc` as an expression server. The server must reconstruct its symbol and type data using information sent from the debugger. It must specify certain machine-dependent data (e.g., the sizes of the basic types) at run time, not when it is compiled. Finally, it must generate PostScript suitable for use by `ldb`.

In addition to these major changes, a few minor changes are needed. The server's main procedure, instead of compiling an entire program, loops, handling requests from the debugger. In a C program, it is permissible to call a function that is not declared. The function is assumed in another compilation unit, and if it is not, the linker detects the error. The expression server does not permit calls to such functions. It has access to the symbols for the entire program, and if there is no symbol for a function it cannot be called.

5.3.1 Reconstructing `lcc`'s symbol and type data

When `lcc`'s lexical analyzer encounters an identifier, it attempts to look it up in the symbol table. When `lcc` is acting as an expression server, the lookup operation may require sending a message to the debugger asking about the identifier. It must stop reading the expression containing the unresolved identifier and switch to a different input stream to read the information coming back from the debugger. Because it is more convenient for the expression server to receive information from the debugger as a stream of C tokens, not as a stream of bytes, the state of lexical analysis is suspended before input is switched to the alternate stream, then restored when input is returned to the original stream.

`lcc`'s input and lexical-analysis modules keep state in global variables. Stacks are used to save and restore these variables, making it possible to switch to the alternate input stream without losing the state associated with the original stream. It is possible to save the current input state on the input stack, to restore the current input state from the stack, and to swap the current input state with the state on top of the stack. The input state includes an input buffer, pointers to locations in the buffer, source-file name and coordinates, and a file descriptor. An existing function initializes the current input state to read from a file descriptor.

Another stack is used to save and restore the state of the lexical analyzer. The state of the lexical analyzer includes the last token read, its source location, and all symbols, values, and strings that are associated with it. The expression server does not need to keep this state after reading the alternate input stream, so no swap operation is needed. The stack support adds 80 lines of C to the input module and 40 lines to the lexical analyzer.

`lcc` uses the function **`lookup`** to find the symbol associated with a name. Normally, **`lookup`** returns 0 when no symbol is associated with a name, but in the expression server it calls **`getsymbol`**, which asks the debugger about the name. The debugger looks up the name as described in Section 4.2, and it sends the corresponding symbol back to the server. In the process, three representations of symbols and types are used. Symbols and types are first created by `lcc` during the original compilation, using `lcc`'s internal representation. At the end of the compilation, `lcc` writes a PostScript representation of the symbols and types into the symbol table. At debug time, **`ldb`** reads the PostScript and, when the user evaluates an expression, it sends the symbols and types over the byte stream using a third representation. The expression server receives the byte-stream representation and reconstructs `lcc`'s original internal representation. The compiler writer controls the internal and transmitted representations. The PostScript representation is somewhat constrained; symbols and types must be represented as dictionaries, and the dictionaries must contain certain keys, as shown in Chapter 4. The compiler writer can add information by adding more keys to the dictionaries. The compiler writer chooses both PostScript and transmitted representations that make it easy to reconstruct symbols and types in the server.

The changes to the compiler that make it possible to recover symbols and types total about 500 lines of C, mostly in the type module. Transmission of symbol and type information from **`ldb`** to the server is straightforward. There are two problems whose solutions are not obvious: communicating type information from the server back to **`ldb`** and handling circularity in the graph of symbols and types. The rest of this section describes the solutions in detail.

When sent from `ldb` to the expression server, a symbol is represented by a stream of C tokens described by the following grammar, in which nonterminals are capitalized and shown in *italics*, terminals are not capitalized, and literals are shown in `typewriter` font:

Symbol	$\rightarrow$	`(` *Kind Type Value* `)`
	|	`?`
Kind	$\rightarrow$	`"constant"` | `"variable"` | `"procedure"` | `"type"`
Value	$\rightarrow$	integer constant | ϵ

The grammar for *Type*, which describes types, appears below. The question mark indicates that no symbol is associated with the given name.

In the server, `getsymbol` reconstructs the symbol according to its kind. For a constant, `getsymbol` creates a symbol representing an enumeration literal and assigns it the integer *Value*. Only enumeration identifiers are categorized as constants. Variables whose values are declared to be constant are assigned an appropriate type, e.g., `const int i = 3` has type `const int`. `getsymbol` treats both variables and procedures as if they had been declared with storage class `extern` in a local scope. The `extern` storage class enables the expression server's back end to distinguish target-program variables from compiler-generated temporaries, which appear to the back end as variables with storage class `auto`. `getsymbol` treats types as if they had been declared locally with `typedef`.

The representation defined by the grammar omits most of the fields used in `lcc`'s representation of symbols (Fraser and Hanson 1991a). To recreate symbols, `getsymbol` calls the same front-end functions that `lcc` calls when parsing declarations. These functions initialize the fields omitted from the representation defined by the grammar. Before installing a symbol, these functions call `lookup` to make sure no other symbol in the same scope has the same name. Because `getsymbol` is called only when a name is undefined, `lookup` does not find a symbol with the same name, so it calls `getsymbol` to ask the debugger about the name. If implemented naively, `getsymbol` would send a message to the debugger, receive the reply, and try to install the new symbol, resulting in infinite recursion. `getsymbol` avoids an infinite recursion by using a global variable to keep track of its activations. When `getsymbol` is active, recursive calls to `getsymbol` return 0 without asking the debugger for information.

The support for symbol recovery takes about 95 lines for communication with `ldb` about 15 lines to empty the symbol tables between expressions.

`lcc`'s internal representation of types, the representation used in PostScript, and the representation used on the stream are all similar. Every type has an operator, which corresponds either to a basic type like `int` or `char`, or to a constructor like array or pointer. Every type has an operand type, but the operand types are meaningful only for some constructors; arrays have element types, pointers have referent types, and functions have return types. In other cases, the operand types are zero. `lcc` also records the size and alignment of each type, and it associates additional data with function prototypes and with the fields of structures and unions. All these parts of types are represented directly in PostScript.

Symbols are transmitted only from the debugger to the expression server, but types flow in both directions. The debugger must tell the server about the types of identifiers, and the server must tell the debugger about the types of expressions. The server and debugger maintain parallel type caches that associate small integers with C types. The debugger determines when types are added to the caches, which are initialized with only the predefined types. The debugger identifies a type already in the cache, called a "known" type, by the presence of an `index` key in its dictionary. The debugger and server must assign indices in the same order.

The type cache makes it easy for the server to generate PostScript code that refers to a type; the code uses the type's index to fetch the type dictionary from the debugger's type cache, which is a PostScript array. The type cache incidentally improves expression-evaluation performance; no type is transmitted more than once. The performance benefits are noticeable mostly in Modula-3 programs, because the Modula-3 compiler generates deeply nested C structure types. For example, it requires 387 types to describe the structure `ldb` uses to represent a target program. When `ldb` is used to debug itself, it takes 11 seconds to transmit these types to the expression server. Once transmitted, these types can be used in expressions without perceptible delay.

When the debugger needs to send a known type to the server, it sends only the type's index. To send an unknown type, it first sends the operand type, then the other fields: the operator, size, alignment, and so on. After sending an unknown type, it assigns the next unused index to the type, which becomes known. This algorithm terminates because operand types alone cannot form cycles; cycles are caused by recursive types, which in C can be created only by using a field of a structure or union. The debugger breaks the recursion by sending information about structures and unions without any information about the fields; the expression server must send an explicit request for the fields after creating the type. This scheme is analogous to filling in the fields of an incomplete structure declaration,[1] but the expression server does not use the existing code that handles incomplete structure declarations; it manipulates the type representation directly.

`ldb` sends a type to the server using a stream of C tokens described by the following grammar.

$$\begin{array}{lcl} \textit{Type} & \rightarrow & \text{index} \\ & | & \textit{Type}\ \texttt{[}\ \text{index op size align}\ \textit{Typesym}\ \texttt{]} \end{array}$$

"index," "op," "size," and "align" are integer constants. Index 0 always represents the zero pointer.

[1] In C, it is legal to declare a structure type without listing its fields, e.g., `struct node`. If necessary, the fields are defined later by giving a full declaration of the type.

The *Typesym* can contain a function prototype, which is part of a function type, or it can contain the tag of a structure, union, or enumeration type. Its representation is

Typesym	$\rightarrow$	index	typesym of known type (or 0)
	$\mid$	tag cfields vfields	structure, union, or enum
	$\mid$	`typedef 0`	function with no prototype
	$\mid$	`typedef` `<` *Typelist* `>`	function with prototype
Typelist	$\rightarrow$	*Type Typelist* $\mid \epsilon$	

where "tag" is a string constant and the other terminal symbols are integer constants. "cfields" and "vfields" are part of `lcc`'s internal representation of structure and union types; they indicate whether the structure or union contains fields declared constant or volatile.

When `ldb` evaluates the expression `argv[0]`, the `argv`'s symbol and type are transmitted to the server in the following form.

```
("variable" 1[18 7 4 4 0][19 7 4 4 0])
```

`argv` is a variable, and the rest of the string is its type, `char **`. The initial 1 is an index into the type cache; it refers to the predefined type `char`. The first sequence of numbers in brackets introduces a new type, number 18, with operator `POINTER` (7), size and alignment of 4 bytes, and a null *Typesym* (0); type number 18 is `char *`. Type number 19 is like number 18 except that its operand is number 18, not number 1, making number 19 the type `char **`, which is the type of `argv`.

A separate procedure, `fill_in_fields`, is used to handle the fields of structure and union types. `getsymbol` passes the type of its symbol to `fill_in_fields`. If the type, or any operand type, is an incomplete structure or union type, it sends a request to the debugger for the names and types of the fields. The representation sent back is

Fields	$\rightarrow$	name *Type* offset from to *Fields*
	$\mid$	ϵ

where "name" is a string constant and the other terminal symbols are integer constants. "offset" is the offset of the field from the beginning of the structure or union, and "from" and "to" identify the position of a bit field within the word at "offset." Because the fields of a structure may themselves be structures, pointers to structures, etc., `fill_in_fields` calls itself on the type of each field.

`fill_in_fields` is needed to handle the recursive type

```
typedef struct list {
  void *x;
  struct list *link;
} *List;
```

If the user evaluates an expression containing `List`, `ldb` transmits this representation of `List`:

```
("type" 0[20 9 8 4 "list" 0 0][21 7 4 4 0])
```

`"type"` identifies `List` as a name defined by `typedef`. The first 0 indicates an operand type of 0, i.e., the null pointer. The next new type is number 20. Its operator is `STRUCT` (9), and it is 8 bytes long and aligned on a 4-byte boundary. The structure tag is `list` and the structure contains no constant or volatile fields. Type number 21 is another pointer type, in this case `struct list *`. `fill_in_fields` identifies type 20 as an incomplete structure type, and it sends `20 ExpressionServer.sendfields` to the debugger. That PostScript procedure sends the following reply:

```
("x" 14 0 0 0 "link" 21 4 0 0)
```

`x`, of type 14 (the predefined type `void *`) is at offset 0, and `link`, of type 21 (`struct list *`) is at offset 4. The "from" and "to" values of both fields are 0, indicating that they are not bit fields. Because a structure type is transmitted before the types of its fields, the type `struct list` is already in the cache when the time comes to identify the type of the `link` field as a `struct list *`.

To maintain the consistency of the type caches, the server must add a type to the cache only when the debugger does. The server may create types that do not go into the cache; for example, in compiling the expression `&n`, where `n` is an integer, the server may need to create the pointer type `int *`. `lcc`'s type-creation function has been modified to distinguish between types sent from the debugger and types created by the front end.

Not only does the expression server need C code that recognizes the grammars given above, but the debugger needs PostScript code that emits sentences described by those grammars. The PostScript procedure `ExpressionServer.lookup`, called directly by the expression server, uses `putsymbol` and `puttype` to send the symbol, if found, and its type. `ExpressionServer.sendfields` uses `putfields` to send the fields of structures and unions.

The differences between `lcc`'s standard type module and the version that communicates with `ldb` are about 500 lines. About 150 of those lines are diagnostic functions for printing the state of the type cache.

5.3.2 Delaying assignment of machine-dependent data

`lcc` normally requires some machine-dependent information to be specified at compile-compile time (Fraser and Hanson 1991a, Section 2). Examples include the sizes and alignments ("metrics") of the predefined types and the byte order of the target machine. It is desirable, if possible, to avoid having a different version of the expression server for each target, so the expression server gets such information at run time. At startup, when only the predefined types are in the type cache, the server calls the PostScript procedure `ExpressionServer.typeInitMetrics` to send the size and alignment of every type in the cache. It also calls a PostScript procedure that tells it whether structures must be passed to functions by making copies and by passing pointers to the copies. These calls are made by sending PostScript to `ldb`, and the information that is returned applies to the target machine, not the machine that `ldb` and the expression server run on.

`lcc` handles bit fields entirely in the front end by transforming bit-field accesses into full-word accesses composed with shifting and masking operations. Machine-independent handling of bit fields required a change to `lcc`'s front end. Bit fields are assigned to positions starting at either the most significant or the least significant end of a word, depending on the byte order of the target machine. The original version of `lcc` assigned fields to positions in a machine-independent way, but the positions had machine-dependent meanings. Bit 0 meant the least significant bit for little-endian targets and the most significant bit for big-endian targets. The front end used machine-dependent macros to compute the shifts and masks needed to get access to bit fields. For example, when the declaration

```
struct flags {
  unsigned defined:1;
};
```

was parsed, `defined` was always noted as being at position 0, and machine-dependent code determined whether a shift by 0 or a shift by 31 was needed to fetch the field. In the new version of `lcc`, bit 0 always means the least significant bit. The front end does the machine-dependent computation when it assigns fields to positions, so `defined` is assigned either to position 0 or to position 31 depending on the byte order of the target. The transformation of bit-field accesses is now machine-independent; accessing a field at position k always requires a right shift by k, followed by masking.

This change to `lcc` changes the time at which machine-dependent computation is required from the time an expressions are converted to intermediate code to the time declarations are processed. Declarations are parsed only during an original compilation, when a machine-dependent compiler must be used. Intermediate code is generated at debug time by the expression server, which can be machine-independent as a result of the change. The alternative would have been to provide one expression server for each target byte order. Handling bit fields in the PostScript back end was not an option because bit-field accesses are eliminated by the front end.

5.3.3 PostScript back end

The expression server's back end uses `lcc`'s code-generation interface (Fraser and Hanson 1991a), but it interacts with the front end in some nonstandard ways, and the code it generates does not follow exactly the same pattern as the usual assembly code. The front end makes up a dummy function with no arguments and no result, and every time it compiles an expression it makes the expression the "body" of the dummy function and calls the back end to generate code for it. The back end emits a PostScript procedure that has the effect of evaluating the expression and placing the location of the result on the PostScript stack.

A compile-time switch configures the expression server to generate trees instead of dags. This change simplifies the back end; code is emitted by a postorder traversal of the trees, in which a call to a PostScript procedure is emitted for each node. Like other PostScript procedures, the procedure finds its arguments, corresponding to the values computed by the node's children, on the PostScript operand stack, and it places its result there.

`lcc`'s back end uses several integer and floating-point types and a pointer type. The integer and floating-point types are a subset of the types of values found in an abstract memory, and the expression server uses PostScript integers and reals to represent them. Pointers are represented by locations in an abstract memory.

`lcc`'s front end creates temporary variables. These temporaries may be used to hold string or floating-point literals or intermediate results, like structures returned by functions. The expression server may need to pass the address of such a temporary to a procedure in the target, e.g., to evaluate `strcmp(s, "mumble")`, so some locations in the target data space must be reserved to hold front-end temporaries. The debug nub reserves a *global area* for this purpose and supplies a pointer to it in the variable `DebugNub_gp`. Because some compiler-generated variables must go in the global area, it is simplest to put all compiler-generated variables there. This simplicity has a cost in performance, as shown in Section 7.7. The expression server determines the size of the global area by looking up `DebugNub_gp` and walking the type structure to find the size of the referent. As a result, the size of the global area can be changed simply by recompiling the nub; no change to the expression server is needed.

The global area is divided into three sections: one for compiler-generated static variables, one for compiler-generated automatic variables, and one for procedure arguments. The last two sections correspond to the local-variable and argument-build sections of stack frames used by standard back ends. The first section, which holds string constants and floating-point constants, is allocated globally by `lcc`'s normal back ends, but in the PostScript back end it is allocated on a per-expression basis. Different expressions may use different compiler-generated statics, and it is undesirable to fill the global area with statics during a long-running debugging session, so the definitions of such states are not emitted at compile time as by standard `lcc`. Instead, the first part of the expression-evaluation procedure stores statics into the global area. The back end forces the front end to emit these constants at the right time.

`ldb` uses about 200 lines of PostScript to implement the procedures to which the back end emits calls. Most such procedures correspond to nodes in `lcc`'s dag language; 97 of the language's 111 operators have PostScript analogs. The operators that emit constants or the address of variables do not have PostScript analogs; the back end emits the appropriate numeric literals or location-creating code directly. The dag nodes that hold the values returned by `return` statements do not have PostScript analogs either; they need not be implemented because `return` cannot appear in an expression. Most of the PostScript implementations use the standard PostScript operators plus `ldb`'s extensions for abstract memories. Some require special extensions to the PostScript interpreter. For example, standard PostScript does not supply relational and arithmetic operators that operate on unsigned integers or 32-bit floating-point values. A single new PostScript operator is needed to implement the differently typed variants of procedure call; this operator sends a message to the debug nub asking it to make the call (see Section 6.4.1), and it hides events, like breakpoints, that might occur between the call and the return, as described in Section 7.6. Because `lcc`'s front end un-nests procedure calls, the PostScript for arguments is relatively simple; it copies the arguments sequentially into the argument-build section of the nub's global area.

The PostScript procedure generated by the server uses an "expression closure," which is a PostScript dictionary that holds the locations of the free variables of the expression. For example, the PostScript code that computes the location of `argv[0]` is

```
{ Exp$closure /argv get
  INDIRP
}
```

The first line gets the location of `argv` from the expression closure, and the second fetches a pointer from that location. The expression closure is computed by the code

```
<< /argv dup EXTERN >>
```

`EXTERN` is executed before `>>`, and it computes the location of `argv`, so the dictionary associates the key `argv` with the location of the target variable `argv`. `EXTERN` looks up the variable's symbol-table entry in the `Scope.T` provided to the server, then gets the location using the `where` key:

```
/EXTERN { ExpressionServer.scope exch Scope.T.lookup /where get } def
```

As emitted by the compiler, `where` may be associated with either a location or an anchor-symbol procedure; `Scope.T.lookup` guarantees that it is associated with a location.

Because `lcc`'s front end un-nests calls, it may announce calls to the back end before the results of those calls are used. For example, in the expression `atan2(sin(1.0), cos(1.0))`, the result of `sin` cannot immediately be used as the first argument to `atan2`, because the argument-build area is needed for the call to `cos`. When calls are announced before their results are used, the PostScript code generated by the back end puts the result in a temporary PostScript variable instead of leaving it on the PostScript operand stack. The variable is used when the result is needed. The result can always be stored in a simple variable because the front end allocates temporary space to hold non-scalar results. Figure 19 shows the code generated for the example; I have inserted comments describing the calls. The temporary variables have names beginning with `Exp$result`. "`8 8 ARGD`" indicates a `double` argument with size and alignment of 8 bytes. `Immediate` is applied to the final

```
{ 1.0   8 8 ARGD
  Exp$closure /sin get   CALLD
  /Exp$result1006187c exch def  % PStemp87 := sin(1.0)
  1.0   8 8 ARGD
  Exp$closure /cos get   CALLD
  /Exp$result1006193c exch def  % PStemp93 := cos(1.0)
  Exp$result1006187c   8 8 ARGD
  Exp$result1006193c   8 8 ARGD
  Exp$closure /atan2 get   CALLD
  /Exp$result100619fc exch def  % PStemp9f := atan2(PStemp87, PStemp93)
  Exp$result100619fc
  Immediate
}
```

Figure 19: Code to evaluate `atan2(sin(1.0), cos(1.0))`.

result to produce a location because printing procedures use the locations of values, not the values themselves.

The examples shown so far omit the technique used to handle nontrivial control flow. The expression server uses an indirection; the PostScript used to evaluate an expression is actually stored in the expression closure. The indirection is best introduced using an expression with simple control flow, `argv[0]`. The procedure that evaluates an expression is associated with the key `$eval` in the expression closure. The procedure P that is sent back to the debugger is

```
{ /Exp$closure ⟨expression closure⟩ def dup /Exp$m exch def Exp$eval }
```

The abstract machine to be used with this procedure is left on the stack *and* associated with the key `Exp$m`. The procedure `Exp$eval` looks in the expression closure for the PostScript code that actually evaluates the expression. The expression closure is computed dynamically, then stored in P. The code that is actually emitted for `argv[0]` is

```
{ /Exp$closure null def dup /Exp$m exch def Exp$eval }  % procedure P
dup 1 << /argv dup EXTERN
         /$eval {
            /Exp$gp /DebugNub_gp EXTERN INDIRP def
            /Exp$ap Exp$gp 1024 Shifted def
            Exp$closure /argv get
            INDIRP
         } % evaluation code
      >> put % save closure in procedure P
```

Procedure P appears first, with `null` in the place of the expression closure. It uses `Exp$eval` to evaluate the procedure in the closure. The remaining code builds the closure and stores it in procedure P. Within the closure, the procedure associated with `$eval` actually evaluates the expression. The lines beginning with `/Exp$gp` and `/Exp$gp` define the location of the global area and the argument-build section of the global area, respectively. They are not used in this simple expression. The rest of the procedure is the code shown earlier to put the location of `argv[0]` on the operand stack.

When an expression requires nontrivial control flow, the expression closure contains not a single procedure but one for each extended basic block. `Exp$eval` executes one basic block after another until the expression is evaluated. The procedure that corresponds to the first extended basic block is associated with the key `$eval`. Blocks are labeled; the expression closure associates the names of labels with the procedures implementing the blocks. For example, the expression `argv != 0 && argv[0]` requires a conditional branch to avoid evaluating `argv[0]` when `argv` is zero. The corresponding PostScript code is shown in Figure 20; it has three extended basic blocks. The answer goes into a compiler-generated temporary at `Exp$gp`. If either `argv` or `argv[0]` is zero, the test fails by branching to `L$7`, which assigns 0 to that temporary; if both tests succeed, the code assigns 1 to the temporary. `INDIRI` and `INDIRP` fetch, `CVPU` converts a pointer to an integer, `EQU` compares integers, `ASGNI` assigns, and `JUMPV` transfers control.

The PostScript `stopped` and `stop` operators (a form of catch and throw) are used to transfer control between basic blocks; a jump to a label L puts the procedure associated with L on the PostScript operand stack, then executes `stop`. When the `stop` is caught, the new procedure is executed. The implementation of `JUMPV`, the argument to which is the name of a label, is

```
/JUMPV { Exp$closure exch get stop } def
```

`Exp$eval` catches the `stop` and executes the procedure on the stack. When the procedure does not `stop`, like `L$8` in Figure 20, `Exp$eval` exits. `Exp$eval` begins by executing the first extended basic block, which is associated with the name `$eval`.

```
/Exp$eval { Exp$closure /$eval get { stopped not { exit } if } loop } def
```

This technique is general enough to handle all the control flow within a C function, not just control flow for expressions.

The expression server's back end is less than 500 lines of C. The supporting PostScript is another 200 lines. 30 lines of Modula-3 are needed to implement the special operators used to implement the PostScript, not counting procedure call, which is more complex.

```
{ /Exp$closure null def dup /Exp$m exch def Exp$eval }
dup 1 <<
  /argv dup EXTERN
  /$eval {
    /Exp$gp /DebugNub_gp EXTERN INDIRP def
    /Exp$ap Exp$gp 1024 Shifted def
    Exp$closure /argv get INDIRP CVPU
    0
    /L$7 EQU                          % if ((int)argv == 0) goto L$7
    Exp$closure /argv get INDIRP INDIRP CVPU
    0
    /L$7 EQU                          % if ((int)(argv[0]) == 0) goto L$7
    Exp$gp 1 ASGNI                    % tmp0 := 1
    /L$8 JUMPV
    }
  /L$7 {
    Exp$gp 0 ASGNI                    % tmp0 := 0
    /L$8 JUMPV % fall through
    }
  /L$8 {
    Exp$gp                            % location of the result is tmp0
    }
  >> put % save closure in proc
```

Figure 20: Code to evaluate `argv != 0 && argv[0]`.

5.4 Discussion

Re-using the compiler has a limitation: the debugger cannot provide an extended language for debugging. An extended language might be useful for defining debugging functions to check or print the values of linked data structures. If the language were polymorphic, such functions could be used with data structures of different types. An extended language could include operations that control the debugger, such as planting or removing breakpoints or stopping for a dialog with the user (Crawford *et al.* 1992). An extended language could be implemented within `ldb` or as an expression server.

`ldb`'s expression server implements a minor extension to C: a one-line change to the lexical analyzer treats `$` as a letter instead of an invalid character. `ldb`'s machine-dependent PostScript code defines names for registers, like `$r1` (see Section 10.2). These names begin with a dollar sign, so they do not clash with names used by the target program, but they can be used in expressions.

The techniques used to implement expressions in PostScript could also be used to implement statements. Control flow presents no problems, but some thought would be required to handle declarations in a compound statement. The existing code handles automatic variables, but static variables would require changes; the PostScript currently generated discards such variables after the evaluation of each expression.

`ldb` can re-use a variety of compiler implementations because it puts the expression server in a separate address space. The compiler and debugger need not be written in the same language, need not support the same data types, and need not agree on how to manage storage or share input and output. The cost of this flexibility is that the compiler writer must devise PostScript procedures that take symbol-table data and send it to the expression server over a byte stream, as well as procedures within the server that receive the data. If the `lcc` front end could easily have shared an address space with `ldb`, `ldb` could have made procedure calls to expression-server code instead of sending messages between processes. If compiler and debugger are built together, it is not difficult to make them compatible so they can share an address space, but if debugging is added as an afterthought, the expression-server approach can be useful.

If the expression server is to re-use code from the original compiler, it must not only reconstruct the symbol and type structures originally used by the compiler; it must also re-establish internal compiler invariants involving those structures. Because such invariants are often undocumented, the re-use is more likely to work if the original compiler writer plans for it. No such planning took place in the design of `lcc`, which had to be retrofitted as an expression server. Even so, it demonstrates that the effort required to make a compiler work as an expression server can be small.

If it had multiple expression servers, `ldb` could support multiple programming languages, perhaps by associating a server with each procedure. At present, it supports only ANSI C. Almost all of a language is implemented in its expression server, but name resolution is implemented in the debugger by the `lookup` method of the type `Scope.T`. The current name-resolution algorithm assumes that a name has at most one denotation, so it limits `ldb` to languages that do not support operator overloading. The existing algorithm could handle languages with nested procedures, like Pascal and Modula-3. A name-resolution algorithm that resolved names to sets of symbols would require additional support in the PostScript symbol tables but might be suitable for languages that support overloading, like Ada and C++.

Chapter 6

The Debug Nub

Early debuggers ran in the same address space as their targets (Digital 1975). Such debuggers have direct access to target memory and registers using ordinary machine instructions. Similarly, control can be transferred between debugger and target using simple branch instructions. Such simplicity offers good performance (Aral, Gertner, and Schaffer 1989; Kessler 1990), but it also presents problems. Debugger and target compete for the same resources; the debugger must be small in order to fit in the same space with large applications. It may be difficult to protect the debugger from a faulty program that sprays bits randomly through memory. The debugger may not be able to debug itself, preventing the use of an old, reliable version of the debugger to debug a new, unreliable version.

These problems can be solved by letting the debugger and target run in separate processes, a strategy that introduces a new set of problems. The operating system must provide support for manipulating target processes. A debugger must be able to select a target process, stop it and start it, and manipulate its address space and CPU state. Most operating systems provide additional support, like support for breakpoints. The two-process model is commonly used on engineering workstations. Most debuggers for personal computers use the one-process model because most personal computers provide only a single process, although "world-swap" techniques make it possible to use the two-process model on a one-process system (Redell *et al.* 1980).

In standard Unix systems, run-time support enables a debugger to manipulate a child process. The debugger starts and stops the child and reads and writes its memory and registers by calling `ptrace`. `ptrace` also provides support for intercepting signals delivered to the target process. Most vendors have found it necessary to improve on `ptrace` (Adams and Muchnick 1986). One extension makes it possible for the debugger to refer to registers without knowing their locations in kernel data structures, somewhat like an abstract memory. Another makes it possible to debug all processes running on the same machine as the debugger, not just child processes of the debugger.

The Topaz teledebugging protocol, TTD, shows another approach to run-time debugging support (Redell 1989). Every Topaz kernel includes two "teledebug servers," one for debugging user processes and one for the kernel. The debugger selects a target process or kernel and manipulates it by sending special packets over the network. The network driver recognizes these packets and routes them to the appropriate debug server. TTD treats all debugging as remote debugging; there is no need to optimize the case in which the debugger and target run on the same machine. The Topaz kernel provides threads, so, in addition to the standard services, TTD provides primitives for manipulating threads. The significant advantages of TTD are reliability and availability; it is always possible to debug the lowest level of the software that is broken, and a debugger can debug any machine to which it can send network packets. One useful application of remote debugging is to use a debugger running an old, reliable version of the operating system to debug a machine running a new, unreliable version.

A third approach uses special files to support debugging. These files, one for each running process, usually appear in a directory called `/proc`; the debugger opens a file and then uses normal file-system operations to control the process (Killian 1984). Typically the file can be read or written to manipulate the target's address space; other operations are performed using the `ioctl` system call. The Plan 9 operating system provides similar support, except that it uses a collection of files for control, instead of `ioctl` operations on a single file (Pike *et al.* 1992). Because Plan 9 can mount such files remotely, they form a remote-debugging facility.

If `ldb` used existing run-time support for debugging, the accidental differences in support between machines and vendors would introduce unnecessary retargeting effort. Adding machine-independent support to existing kernels might be even more burdensome. As a compromise, `ldb` uses a debug nub that runs in user mode in the target address space. The debugger and nub use kernel services to establish a bidirectional byte-stream connection. This model of debugging is based on TTD; the nub is the analog of TTD's teledebug server. This compromise makes it possible to explore how retargeting effort is affected by the set of services implemented in the nub. Byte streams are a simpler abstraction than debugging support, so there is a better chance that implementations on different operating systems will be similar. For example, `ldb` works on four different versions of Unix from different vendors, but all provide the same byte-stream abstraction: sockets.

The principle guiding the design of the nub has been to keep it small without compromising reliability or retargetability. The nub is loaded with every program, so the cost for programs that are never debugged should be small. A simple nub is easier to retarget and to make reliable. Adding services to the nub is justified only if implementing such services in the debugger would make the system significantly less reliable or more difficult to retarget. Significant performance improvements would also justify adding services to the nub, provided those improvements affected users.

6.1 Debugger's view of the nub

From the debugger's point of view, a target is always in one of three states: running, stopped, or disconnected. A disconnected target may in fact be running or stopped, but the debugger does not attempt to keep track. When the target is running, the nub provides no services to the debugger, except that when the target stops, the nub sends the debugger a message describing the event that caused it to stop. When the target is stopped, the nub provides three sets of services: it can act as a trapped memory (Section 3.2.1), call a procedure, or restart or disconnect the target. When the target is disconnected, it can either run, exit, or wait for a connection from a new instance of the debugger.

The debugger sees the nub as a Modula-3 object of type `Wire.T`. A `Wire.T` is a subtype of trapped memory, an abstract memory with methods that plant, suspend, and remove traps. `Wire.T` also has methods that support the nub features: `getevent` to be notified of an event by the nub, `call` to call a procedure, and `putorder` to resume execution or break the connection. Because the `Wire.T` is a subtype of abstract memory, it is also a PostScript object.

```
TYPE
  Wire.T = TrapMemory.T OBJECT METHODS
        getevent() : Wire.Event;
        call(entry, argbase, argbytes: INTEGER; resulttype: Memory.Type;
             structresult, mdata: INTEGER);
        putorder(order: Wire.Order);
        close();
      END;
  Wire.Order = { Continue, Disconnect, Exit, GetNewDebugger, Unwind };
  Wire.Event = RECORD sig, code, contextoffset: INTEGER; val := Memory.Value{} END;
```

The `close` method is used to recover resources, like open file descriptors, that might be associated with a particular implementation of `Wire.T`.

A protocol governs the order in which the debugger may call the methods of a `Wire.T`. When the target is disconnected, the debugger may call only `close`. To get a target that is not disconnected, the debugger must connect to a new, running target and create a new `Wire.T`.

When the target is running, the debugger may call only `getevent`. If the target stops, `getevent` returns an event, but if it becomes disconnected, `getevent` raises an exception. Termination of the target process is treated as a disconnection. The event contains a Unix signal number and code and the address of a process context, an area in the target data space containing the state of the processor when the event occurred. The format of the process context is machine-dependent, but it always includes a program counter and registers. Unix does not use signal 0, but the nub uses it to encode special events like the return of a procedure or the unwinding of a suspended call. The `val` field of the event is meaningful only for procedure-return events, in which case it contains the value returned by the procedure. Even then the value may not be meaningful, for example for a `void` procedure.

When the target is stopped, the debugger may call any of the trapped-memory methods to fetch or store values or to plant, suspend, or list traps. Each of these methods leaves the target stopped. The debugger may also call `putorder`. The orders `Continue` and `Unwind` leave the target running. `Unwind` is actually a synchronous operation; a target restarted with an `Unwind` order immediately stops with an "unwound" event, but it is convenient to treat it as asynchronous because that makes it unnecessary for the nub to distinguish a procedure call that is unwound from one that simply returns. The orders `Disconnect`, `Exit`, and `GetNewDebugger` tell the nub to disconnect from the debugger. After disconnection, the nub continues execution of the target, terminates the process, or waits for a connection from another debugger. Finally, when stopped, the debugger may invoke `call`. A call restarts the target, which runs until the next event occurs. Usually the next event is that the procedure returns a value, but `ldb` handles other events too, as described in Section 7.6.

6.2 Implementing the nub protocol

As shown in Figure 21, the debugger and nub use several layers of abstraction to build up from a bidirectional byte stream to the protocol described above. The byte stream is established in one of three ways. The most common is for the nub to create a socket that can accept connections and to advertise the socket by printing its host name and port number; `ldb` makes the connection using the `connect` system call. A more sophisticated mechanism might advertise by registering the connection with some network-wide service. `ldb` can also fork a target program as a child process, and the nub can fork a debugger as a child process, in which cases the byte stream is a pair of Unix pipes.

Marshalling code writes integers, floating-point values, and `Memory.Value`s on a byte stream. The format for integers is machine-independent; integers are transmitted one byte at a time, least significant byte first, even if both nub and debugger run on big-endian machines. Transmitting floating-point values is more problematic, primarily because different floating-point formats support different non-numeric values. Floating-point values are converted to 64 bits for transmission, then cast to a pair of 32-bit words. The words are transmitted as integers, least significant word first. This scheme works correctly whenever `ldb` and its target are running on the same architecture. It can also support cross-debugging, provided the host and target use the same floating-point format. They need not use the same byte order. Since the MIPS, SPARC, and 68020 all use the IEEE 754 floating-point format, cross-debugging works correctly among these architectures. If a user running on one of these architectures attempts to cross-debug a VAX target, however, all of the floating-point values are wrong. Possible solutions to this problem are discussed at the end of this chapter.

Above the marshalling layer, the nub and debugger exchange typed messages. At this level there is a seven-operation request/reply protocol. The debugger may send "fetch," "store," "plant,"

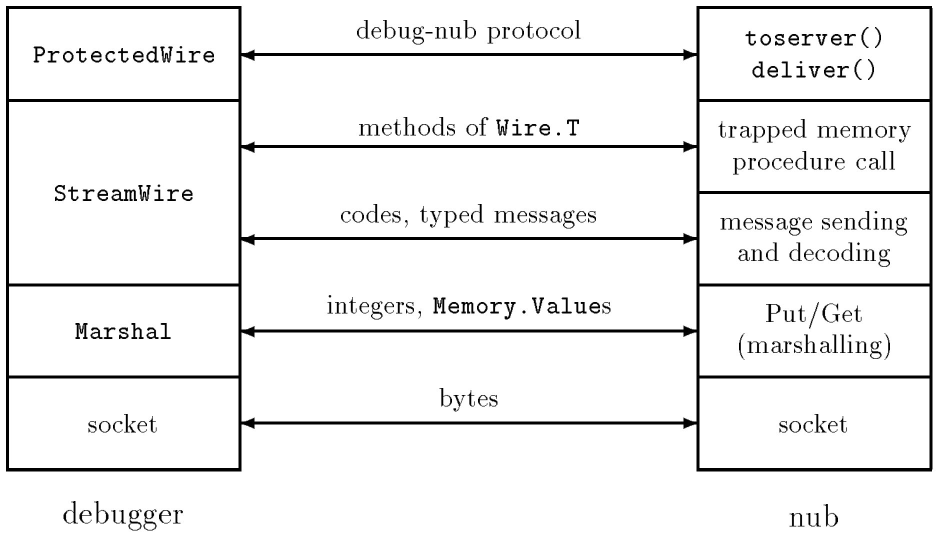


Figure 21: Layers implementing the debug-nub protocol.

"suspend," "traps," "order," or "call." The nub may send "event," "value," "stored," "address," "fail," "called," or "trapsare." The messages include appropriate data; for example, the "fetch" message includes a location and a type.

`StreamWire` is the standard implementation of `Wire.T`. It combines two layers of protocol: the layer that assembles messages and the layer that uses those messages to implement the `Wire.T` methods. The nub uses two functions, `toserver` and `deliver`, to ensure that it sends and receives messages in accordance with the nub protocol. `toserver` handles changes of state caused by events and orders, and `deliver` handlers requests from the debugger that leave the target stopped. In response to those requests, `deliver` calls the nub's implementations of the trapped-memory and procedure-call methods.

On the debugger side, the `ProtectedWire` interface ensures that the methods of a `Wire.T` are called only in orders permitted by the protocol. `ProtectedWire.T` implements the same methods as `Wire.T`, and it is the `ProtectedWire.T` that is exported to the rest of the debugger. Its methods track the state of the target, and each method raises an exception if called in a forbidden state; otherwise it calls the corresponding method of the underlying `Wire.T`. Only the `getevent` method is different; if it is called while the target is stopped, it does not raise an exception, but blocks until the target state changes. Blocking makes it possible for the debugger to have a separate "listener" thread that calls `getevent` repeatedly (see Section 7.6). The `ProtectedWire.T` contains a lock used to protect the target state. The same lock prevents race conditions on the nub, which might be caused by two different threads trying to fetch at the same time, for example.

The message level of the nub protocol is described by a PROMELA specification (Holzmann 1991). PROMELA tools have been used to validate the protocol, making sure that neither side ever sends a message that the other is not expecting.

6.3 Illustrating the nub protocol

The debugger's first interaction with the nub illustrates the nub protocol. At program startup, the nub installs a Unix signal handler that gets control when the target process faults or encounters a breakpoint. A single signal handler suffices for all events; it catches several different signals. The nub's startup code sees `--pause--` and calls `DebugNub_Pause`, which causes a trap, and a signal is delivered to the nub. The trap signal is delivered to the handler, and the handler calls the nub's `connect` function, which prints a message and waits for the socket connection to be established.

```
orchard % fib --pause-- 12
=> DEBUG NUB <=
to debug: ldb target fib connect orchard 2062
```

The debugger's `connect` command establishes the network connection, uses it to create a running `ProtectedWire.T`, and forks a listener thread that starts calling the `getevent` method of the wire. After the fork, the debugger is running two threads: the user-interface thread, which responds to user input, and the listener thread, which responds to target events. In principle, `ldb` could handle multiple targets simultaneously by forking a listener thread for each target, but the current user interface can manipulate only one target at a time.

The nub's `connect` function returns to the signal handler as soon as the network connection is established; subsequent calls to `connect` do nothing as long as the connection remains open. The signal handler then calls `toserver`, which calls `deliver`, which sends an "event" message to the debugger. The message contains signal 5 (`TRAP`), code 0, and the address of the process context. This message is shown at the top of Figure 22, which shows the exchange of messages described in this section.

Because the target is running, the debugger's listener thread does not block in `getevent`. It reads the message from the nub, and the `getevent` method returns the event, marking the target as stopped. The listener thread delivers the event to the user interface, which prints it:

```
ldb fib (disconnected) > connect orchard 2062
=> DebugNub_Pause(99) <=
* 0 <DebugNub_Pause:0> (SparcNub.nw:9,29)
        void DebugNub_Pause(int arg = 99)
ldb fib (stopped) >
```

After delivering the event, the listener thread calls `getevent` again. This time the call blocks, because the target is stopped. The listener thread stays blocked until the user-interface thread starts the target running again.

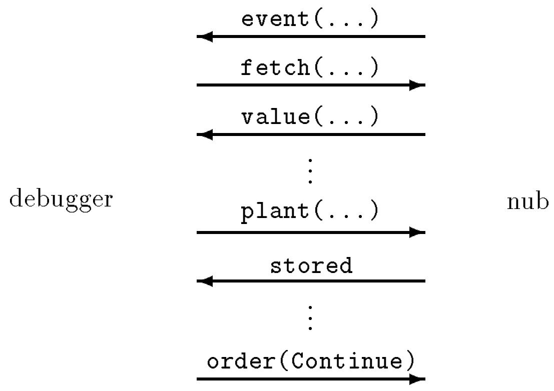


Figure 22: Messages exchanged during the processing of one event.

The user-interface thread prints the current focus. This printing involves fetches from an abstract memory; each fetch ends up in the `StreamWire` implementation, which sends a message to the nub and waits for a response.

Meanwhile the nub's `deliver` function, having delivered the event, is waiting for such a request. It decodes the request and calls the appropriate function within the nub, in this case `fetchabs`. `fetchabs` sends the reply, which includes the value fetched, to the debugger, and `deliver` loops waiting for the next request. The next debugger command makes a different kind of request, i.e., to plant a breakpoint.

```
ldb fib (stopped) > b fib:7 fib.c:13
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) >
```

This request, too, is decoded by `deliver`, which calls `trap_plant` instead of `fetchabs`. The nub continues to execute in `deliver` as long as the target stays stopped.

Eventually the user continues execution of the target. The `c` command results in the user-interface thread's calling the wire's `putorder` method with the `Continue` order, which sends a message to the nub telling it to continue execution. The protected wire recognizes that this order tells the nub to make the transition from stopped to running, and it unblocks the listener thread to await the next event.

When the `putorder` message is received at the nub, the `deliver` function decodes it and returns the request to `toserver`. `putorder` always changes the state of the target, and `deliver` is used only while the target is stopped. `toserver` identifies the order as `Continue`, and it returns to the signal handler, which also returns, restoring the registers and returning execution to the user's program. Control does not return to the nub until the next event, which again activates the nub's signal handler, starting a similar sequence of interactions.

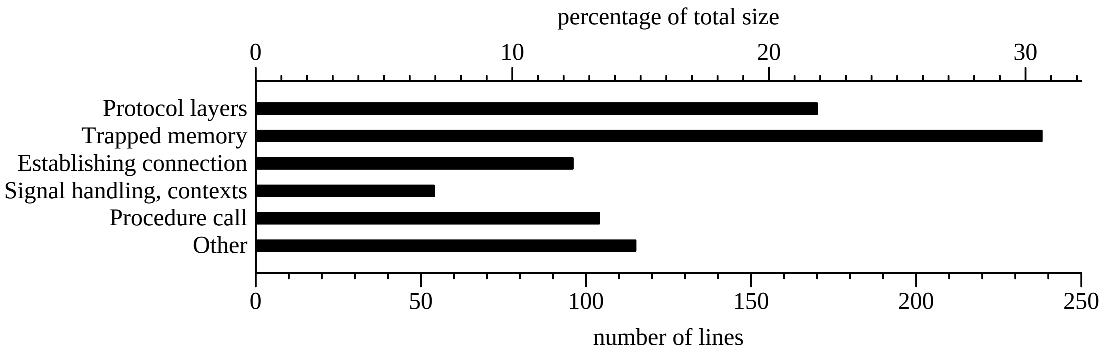


Figure 23: Parts of the debug nub.

6.4 Implementing the nub

Almost one-quarter of the code in the nub is devoted to the marshalling, message, and top layers of the protocol. The trapped-memory methods take 30% and procedure call 15%, as shown in Figure 23. Signal handling and establishing a connection to a debugger account for most of the remainder, with 15% going to initialization, inclusion of header files, and declarations of prototypes for system functions.

To implement a trapped memory, the nub must keep a record of each trap. The record includes the original instruction over which the trap was written and the `Memory.Type` used to write the trap. The records are stored in a hash table with separate chaining, keyed by address. The records are allocated statically because it is not always possible to call a dynamic allocator in a process being debugged. The implementations of the trapped-memory methods maintain the table, send messages to the debugger, and manipulate the target memory. For example, when a breakpoint is set, the debugger sends the nub a message asking to plant a trap. The debugger supplies the address, type, and bits. The nub looks up the address in the trap table and creates a new record to hold the original contents of the address. It fetches those contents, saves them in the record, stores the trap at that address, and sends a message back to the debugger indicating that the trap has been planted. If the nub runs out of records or finds a pre-existing record for that address, it sends a message indicating that it failed to plant the trap.

A fetch from the code space, for example to print assembly code, must first check the address of the fetch in the trap table. If the address is in the table, the fetch returns the original bits stored in the trap record, not the trap instruction actually in memory. In this way the traps are hidden from the abstract memory. Stores in the code space must also check the trap table, modifying the bits in the trap record, if any.

Both the trap-planting and fetching and storing methods use the functions `fetch` and `store` to manipulate memory. When storing into the code space, `store` invokes the machine-dependent macro IFLUSH, if defined, to invalidate the appropriate line in the instruction cache. On most targets, IFLUSH is undefined, but on the MIPS, it is

```
#define IFLUSH(OFF,N) cacheflush((char *)OFF, N, ICACHE)
```

where OFF is the address to be invalidated and N is the number of bytes stored.

A user might cause the debugger to try to fetch from or store to an invalid address, for example, by evaluating an expression that dereferences an invalid pointer. It is not appropriate to deliver a memory-fault event if the debugger itself refers to an invalid address; the nub simply replies to the debugger's request with a message indicating that the fetch or store failed. It installs a special signal handler that catches invalid memory references and longjmps to a buffer set by the implementations of the fetch, store, and trap methods, which use setjmp to detect the failure and notify the debugger. setjmp and longjmp are the Unix implementations of catch and throw. The special handler is installed when control enters the nub. The original handler is restored before control returns to user code.

The connect function, which establishes a connection to a debugger, chooses or creates the file descriptors used to send messages to and receive messages from the debugger. If the target process is a child of the debugger, as indicated by a special argument when the program starts, the nub chooses descriptors 3 and 4, which are arranged for by the debugger. Otherwise, it calls either NetDebugger or ForkDebugger when it needs to establish a connection. NetDebugger creates a socket, prints its number, and waits for a debugger to connect. ForkDebugger forks a debugger as a child process, creating pipes on which to communicate. If the connection is ever lost, for example if the debugger crashes, the nub treats the resulting read or write error as a message from the debugger telling the nub to disconnect and wait for another instance of the debugger.

6.4.1 Procedure call

The nub must be capable not just of calling a procedure, but of handling faults and other events during the execution of the procedure. Moreover, if a fault occurs during the execution of a procedure, the debugger may ask not to continue execution of the procedure, but to unwind the stack past the suspended call and to resume debugging at the context from which the procedure was called, as shown in the example session on page 18. The nub uses setjmp before each call, so it can use longjmp to unwind a suspended call. Figure 24 shows some of the C code that the nub uses to implement procedure call. On the left, the "call context" includes the signal, code, process context, old signal handlers, the global area used by the expression server (Section 5.3.3), and unwind, which holds the context used to unwind to the previous call. The call context is saved in a local variable, and another local variable becomes the new global area, so all allocation is done on the stack. Before transferring control to the requested procedure, the nub restores the signal handlers to their original state; invalid memory references are handled normally during the call, for example. The event that

```
void call(...) {
  register unsigned *argbuild;
  ⟨other local variable declarations⟩

  ⟨push call context onto stack⟩
  ⟨set up signal handlers for call⟩
  if (setjmp(unwind.buf) != 0) {
    unwound(scpoffset);
  } else {
    ⟨set up the call⟩
    ⟨make the call⟩
    returned(scpoffset, val);
  }
  ⟨restore signal handlers⟩
  ⟨pop call context from stack⟩
}
```

```
⟨set up the call⟩≡
  if (always_false) {
    assert(0);
    { RESERVE_SPACE(dummy); }
  }
  { SET_argbuild(); }
  for (i = 0; 4*i < argbytes; i++)
    argbuild[i] = argbase[i];
  if (structptr != 0) {
    STORE_CALLB(structptr);
  }
  { PREPARE_CALL(mdata); }
```

Figure 24: The nub's implementation of procedure call.

marks the end of the call is delivered to the debugger by `unwound` or `returned`; a `returned` event includes the value returned by the procedure. `scpoffset` points to the process context, which is re-used when the call is unwound or returns.

The information supplied to the nub's `call` function is what is passed to the `call` method of a `Wire.T`. The debugger supplies the procedure's address, a block containing its arguments, information about its result, and one word of machine-dependent data. The machine-dependent data is used only for MIPS targets, for which it describes the types of the first two arguments. The MIPS passes those arguments in different registers depending on their types.

The PostScript generated by the expression server puts the procedure's arguments into an argument-build area. The nub's `call` function copies this area into its own argument-build area, and it may put some arguments in registers. The PostScript procedure-call operator produces a simple, scalar result. Integer and floating-point results are returned in different ways on some targets; the nub knows which sequence to use because the debugger provides a `Memory.Type` that describes the result. If a procedure returns a structure, the expression server allocates temporary space to hold it. If the procedure returns a structure result or no result, the debugger provides a meaningless `Memory.Type` and ignores the scalar returned by the nub.

The parameters to `call` are `entry`, the address of the procedure, `argbase`, a pointer to the arguments, `argbytes`, the number of bytes of arguments, `type`, the type of the value returned by the procedure, `structptr`, if nonzero, the address of the temporary space allocated to hold a structure result, and `mdata`, a word of machine-dependent data.

The implementation of the call puts the arguments into the right places, branches to the procedure, and saves the value returned. Much of the implementation of procedure call is machine-dependent, but the nub's `call` function uses a machine-independent template, shown on the right side of Figure 24, that invokes four machine-dependent macros: `RESERVE_SPACE`, `SET_argbuild`, `STORE_CALLB`, and `PREPARE_CALL`. A fifth macro, `REGISTER_ARGS`, is used in the call itself, as shown below. `RESERVE_SPACE` makes sure that `call`'s argument-build area is big enough to hold all the arguments. `SET_argbuild` makes `argbuild` point to the argument-build area, and the `for` loop copies the arguments to the argument-build area. `STORE_CALLB` puts the pointer to the result, if any, in the right place. If necessary, `PREPARE_CALL` satisfies machine-dependent constraints by putting arguments into registers. The macro invocations are surrounded by braces because their definitions may contain assembly code; the braces ensure that `lcc` treats them as statements.

Part of the machine-dependent work, e.g., putting arguments in registers and finding the value returned, is done by the machine-dependent code that the compiler generates when the nub is compiled. The source itself is machine-dependent, except the call must mention explicitly whatever arguments are passed in registers:

⟨*make the call*⟩≡
```
if (val.integer = isinteger(type))
  val.u.n = ((int   (*)()) entry)(REGISTER_ARGS);
else
  val.u.x = ((double(*)()) entry)(REGISTER_ARGS);
```

`val` is the nub's C representation of the debugger's `Memory.Value` (Section 3.2). The `if` statement assigns 1 to `val.integer` if the result type is an integer type, 0 otherwise. The last machine-dependent macro, `REGISTER_ARGS`, lists the arguments that are passed in registers.

The definitions of the machine-dependent macros are best explained by considering the work involved in setting up and making a call. The first step is to ensure that there is enough space in the argument-build area to hold arguments for any call. It is necessary and sufficient for the argument-build area to be as large as the temporary area used by the expression server, because the expression server places all arguments in this temporary area. By default, `RESERVE_SPACE` ensures that `call`'s frame contains a sufficiently large argument-build area by passing a large structure, `dummy`, to a procedure. This call is never actually executed (right side of Figure 24), but the compiler must reserve the space anyway because it cannot deduce at compile time that the call is not executed. On the SPARC, passing a structure reserves only one word, because structures are passed by indirection. The following definition of `RESERVE_SPACE` is used instead of the default one:

```
#define RESERVE_SPACE(dummy) do {  \
  char create_stack_space[sizeof(dummy)]; \
  create_stack_space[0] = 0; \
```

```
} while(0)
```

This macro allocates space on the stack next to the argument-build area, effectively enlarging the argument-build area.

The second step is to make `argbuild` point to the argument-build area. On the MIPS, the argument-build area is located at the bottom of the frame, pointed to by the stack pointer.

```
#define SET_argbuild()  asm("move %argbuild,$sp")
```

Within `asm(...)`, `%argbuild` refers to the register containing the C variable `argbuild`.

If `structptr` is nonzero, the function to be called returns a structure, and `structptr` is the address of the space allocated to hold the result. On the MIPS, such a pointer is the first argument to the function:

```
#define STORE_CALLB(RES)  (argbuild[0] = RES)
```

`PREPARE_CALL(mdata)` is used only on the MIPS, which requires that the first two arguments be passed in floating-point registers when they are of floating types (Kane 1988, page D-22). `mdata` encodes the types of the first two arguments. `PREPARE_CALL` takes 10 lines of C with `asm(...)` to decode `mdata` and to put the arguments in the right registers.

`REGISTER_ARGS` is a list of the arguments passed in registers. The MIPS normally puts the first four words of arguments in integer registers:

```
#define REGISTER_ARGS  argbuild[0], argbuild[1], argbuild[2], argbuild[3]
```

Even when the MIPS puts arguments in floating-point registers, the corresponding integer registers are unused, so it is always safe to put arguments in them.

The procedure-call macros are complex because they must handle four different calling sequences, but not all macros must be defined on all targets; some defaults apply on each target. The 68020, for example, uses the stack pointer to point to the argument-build area, handles functions returning structures by passing a pointer in address register 1, and passes no arguments in registers:

```
#define SET_argbuild()          asm("movl sp,%argbuild")
#define STORE_CALLB(RES)        asm("movl %" #RES ",a1")
#define REGISTER_ARGS
```

The notation `#RES` makes the ANSI C preprocessor emit a string literal containing the name of the variable passed to `STORE_CALLB`; the preprocessor concatenates the three literals to form the argument to `asm`. The VAX is like the 68020, except that it uses register `r1`, not `a1`, for structure results. On the SPARC, the argument-build area is in the middle of the stack frame, a pointer to structure results immediately precedes the arguments, and the first six arguments are passed in registers. Except for `RESERVE_SPACE`, shown above, its macros are

```
#define SET_argbuild()          asm("add %sp,68,%argbuild")
#define STORE_CALLB(RES)        (argbuild[-1] = RES)
#define REGISTER_ARGS           argbuild[0], argbuild[1], argbuild[2], \
                                argbuild[3], argbuild[4], argbuild[5]
```

6.4.2 Process context

When a signal is delivered to the nub's handler, the handler must capture the processor state, e.g., program counter and register contents, and store it in the process context. On the MIPS and SPARC, the `struct sigcontext`, which the operating system passes to the signal handler, has enough register information for `ldb` to reconstruct a stack frame, and it is used as the process context. On the VAX and 68020, the `struct sigcontext` is insufficient; the operating system saves the registers on the stack, but it does not include them in the `struct sigcontext`. The VAX and 68020 handlers use machine-dependent assembly code to store the registers in the context. Once the context is built, the nub passes the Unix signal number, code, and context to `toserver`, which manages the rest of the interaction with the debugger, as described above.

On the 68020, I have defined a process context that holds the program counter and the data, address, and floating-point registers. Three long words are needed to hold each floating-point register, because the floating-point registers are saved in extended format:

```
typedef struct context {
  int data[8];
  int address[8];
  int pc;
  int floatx[8*3];
} Context;
extern Context DebugNub_context;
```

When a signal is handled, the context is saved in the global variable `DebugNub_context`. The 68020 defines the macro `save_pc`, which indicates to the machine-independent part of the nub that the nub's handler should execute `DebugNub_saveregs`, a special assembly-language procedure that saves and restores the registers. The handler invokes `save_pc` to save the program counter included in the `struct_sigcontext`, replaces that program counter with the address of `DebugNub_saveregs`, and returns. The effect is to branch to `DebugNub_saveregs`, which saves the remaining registers. This technique makes the nub independent of the undocumented format that the kernel uses to save the registers on the user stack (Cormack 1988).

The definition of `save_pc` for the 68020 is

```
#define save_pc(PC) (DebugNub_context.pc = (PC))
```

The implementation of `DebugNub_saveregs` uses the 68020's `moveml` instruction to save the data and address registers and `fmovemx` to save the floating-point registers. It then calls `DebugNub_continue`, which recovers the signal, code, and context, and calls `toserver`, as described in Section 6.4.2.

The implementation is

```
_DebugNub_saveregs:
moveml d0-d7/a0-a7,_DebugNub_context
fmovemx fp0-fp7,_DebugNub_context+68
jbsr _DebugNub_continue:l
fmovemx _DebugNub_context+68,fp0-fp7
moveml _DebugNub_context,d0-d7/a0-a7
movl _DebugNub_context+64,sp@-
rts
```

`DebugNub_continue` does not return until the debugger has instructed the nub to continue execution, at which point the registers are restored from the context and the program branches to the saved program counter by first pushing it on the stack, then executing `rts`, the only effect of which is to make the branch. `rts` is used because it does not require a register to hold the destination address. The VAX uses the same technique with minor differences because of its different instruction set: it saves the registers on the stack, then copies them to the context, and it has an indirect addressing mode that enables it to jump to the saved program counter using a single instruction.

6.5 Discussion

Of existing operating-system support for debugging, `ldb`'s nub interface most resembles the Topaz TeleDebug protocol, TTD, from which it is derived (Redell 1989). Some of the virtues of TTD have been sacrificed to avoid kernel changes and to improve retargetability. Because the TTD server is in the kernel, it can provide access to every process on the target machine; the nub provides access only to the process in which it is running. The nub uses Unix kernel services to establish the connection, but the nub itself runs in user space. The Topaz kernel implements threads, and the TTD server is a client of the thread implementation; for each thread it provides state that is analogous to `ldb`'s process context. `ldb` has no support for threads. TTD and `ldb` use different data models; TTD reads and writes arrays of bytes, not values. A debugger can efficiently cache copies of target memory using TTD's model, but it is easier to make a debugger independent of target byte order using `ldb`'s model. The TTD server plants, suspends, and enumerates breakpoints instead of traps. The only difference is that the server, not the debugger, handles the details of resuming execution after a breakpoint; both the TTD server and `ldb`'s nub hide the presence of breakpoints or traps from fetch and store operations. TTD also provides an instruction single-step operation. The original implementation used VAX trace mode; a later implementation for the MIPS simulated single stepping in software, much as described in Section 7.5.

`ldb`'s nub protocol provides some but not all of the advantages of TTD. As with TTD, all debugging is remote debugging, and responsibility for debugging a particular target can be passed from one user to another at a different workstation without changing the target. Like TTD, `ldb` provides good availability; since the nub is always linked with the target program, it can catch unexpected faults and wait for a connection from `ldb`. The target program need not be a child of the debugger or even run on the same machine. `ldb` attempts to provide reliability like that of TTD. Failure of `ldb`, or of the machine or network on which it is running, almost never prevents further debugging; a fresh debugger can almost always be started and connected to the target. Exceptions occur when `ldb` fails in such a way that it sends garbage to the nub; such a failure may put the nub into a state from which it cannot recover. TTD is not susceptible to such problems; TTD works with packets, not byte streams, so it can discard ill-formed packets. The nub cannot discard ill-formed messages from `ldb` because it cannot detect message boundaries. This problem is not inherent in `ldb`'s design; the lowest level of the protocol implementation could be changed to use packets, and the nub could check their validity as TTD does. If packets cannot be implemented in a retargetable way, they can be simulated by reserving a character to separate messages on a byte stream.

Unlike TTD, `ldb` is vulnerable to failure in the target process; because the nub runs in user space, a sufficiently faulty program can destroy the nub's data structures, making it impossible to debug that target. This problem could be eliminated by having the nub use virtual-memory primitives to protect its code and data, although memory protection might require machine-dependent code. The nub can also fail if the user process runs out of stack space; the TTD server does not fail in that case because it runs in the kernel. TTD supports debugging of all levels of system software, including the lowest level of kernel code. It does so by providing two servers: a high-level server used to debug user programs and a low-level server used to debug the kernel. `ldb`'s nub is analogous to the high-level server; it would not be suitable for debugging the kernel. To debug the kernel, the nub would have to use fewer kernel services; for example, it would be better for it to accept packets directly from the network device driver than to use a high-level abstraction like reliable byte streams (Redell 1989). There is nothing in the nub protocol, however, that precludes kernel debugging.

`ldb`'s nub is more complex than TTD in that it supports procedure call. In Topaz, procedure call is implemented entirely in the debugger, which simulates, by manipulating registers and memory, the compiler's actions in setting up a frame for the call.

The nub provides more services than the minimum described at the beginning of this chapter; trapped memory and procedure call could be implemented purely in the debugger using fetch and store. Implementing trapped memory in the nub improves reliability; if the debugger should become disconnected because of machine or network failure or internal or user error, there is enough information in the nub for another instance of the debugger to continue execution of the target. Implementing procedure call in the nub improves retargetability. When code is compiled in the nub, the compiler can set up a stack frame for the called procedure, can recover the return value, and can help with argument passing. The nub can use `setjmp` and `longjmp` to unwind suspended procedure calls. If procedure call were implemented in the debugger, much more effort would be required to retarget procedure calls because the debugger would have to re-implement these machine-dependent functions for each target. When procedure call is implemented in the nub, the compiler provides so much assistance that retargeting is easy; as shown above, only a few lines of machine-dependent code are needed on most targets.

If a target program installs its own signal handlers, the nub does not catch those signals first. This problem could be solved by including in the nub a re-implementation of the library procedure that installs signal handlers, making sure that the nub's handler always gets control first.

Moving the nub into the kernel would result in some simplifications as well as better reliability. No extra programming would be required to recover a process context, since the kernel must already be able to recover process contexts. Handling invalid memory references would be simpler, because the kernel has access to the memory map. The nub's handler would not need user stack space. Conflicts with users' signal handlers would be eliminated. Implementing `call` entirely in the kernel might mean losing the compiler's assistance, since the call has to go from kernel space to user space. A better solution would be to keep `call` in user space, having the kernel call `call` instead of calling a user's procedure directly. A kernel that supports signal handlers can invoke a procedure of a known type; invoking the nub's `call` is no more complex. System calls would be a poor way for a debugger to interact with a nub in the kernel; such an interface would preclude remote debugging. TTD has demonstrated that a datagram-oriented network interface is suitable. An interface representing processes as files would also be suitable, provided those files could be mounted remotely on other computers, as in Plan 9 (Pike *et al.* 1992).

The cost of interacting with the nub can be measured by having `ldb` interpret a PostScript procedure that fetches from the same location 1000 times; each fetch requires one request and one reply. The cost depends on which connection is used; it takes 2.6 msec using pipes, 3.9 msec using a network connection to the machine that `ldb` runs on, and 4.3 msec to a different machine. The experimental error in the measurements is 0.2 msec. It is difficult to make comparable measurements of `gdb` and `dbx`, but Section 7.7 contains comparative measurements of the cost of breakpoints.

Floating-point data complicates cross-debugging. The most obvious problem is different representations of numbers on `ldb`'s machine and the target machine. It could be solved by using a machine-independent representation to transmit floating–point numbers, for example, integers representing the sign, base, exponent, and significand of a floating-point number. An ASCII representation could also be used; numbers can be converted between binary and ASCII representations with no loss of information (Clinger 1990; Steele and White 1990). Floating-point values that are not numbers present a more serious problem; for example, it is not clear what the relationship is between a VAX "reserved operand" and an IEEE NaN ("not a number"). The Sun `xdr` protocol cannot transmit floating-point values that are not representable in IEEE format (Sun 1990a), so it would not be suitable for use by the debug nub. Machines with different representations may also differ in the semantics of their floating-point operations. One possible solution to these problems is to extend the nub and its protocol to do all floating-point operations in the target.

Global floating-point state like IEEE rounding mode presents problems even for single-processor debugging because the state can change dynamically. The user, not the compiler, must ensure that a procedure that changes the floating-point state saves and restores its caller's floating-point state. The location of the saved state is not specified by the calling sequence, so the debugger cannot find it. A user doing floating-point computations in the debugger, e.g., by evaluating the expression `x = x * 0.5`, has no guarantee that the effect on the target is the same as if the target program itself had executed `x = x * 0.5`, because the result might have been rounded differently in the target. This problem affects all debuggers, not just `ldb`; solving it requires either eliminating floating-point state or using the compiler to save and restore the state, just as the compiler saves and restores registers.

Chapter 7

Breakpoints and Events

Breakpoints stop the target program so the user can probe its state. Simple breakpoints are planted at a single location, and they stop the target when control reaches that location. As shown in Chapter 2, the `take` command adds a condition to a simple breakpoint making the target stop only when the condition is true. A counter may also be attached to a breakpoint; if the counter has value k, the target stops the $k+1$st time control reaches the location.

Planting a simple breakpoint does not automatically restart the target; the user must start the target explicitly with the `c` command. `ldb` offers three more complex breakpoint commands, which plant breakpoints, restart the target, and remove the breakpoints when the target hits one. `finish`, shown in Chapter 2, uses this technique to execute the target until the current activation finishes. `next` executes until the target reaches the next stopping point in the current activation, and `stepi` executes until the target reaches the next machine instruction.

The implementations of all the breakpoint commands must be able to react to events that occur in any order, not necessarily the order that is expected. For example, if the user types `finish`, the event that is expected is for the target to hit the breakpoint that marks the end of the activation, but the target may fault or hit some other breakpoint in the current activation. The implementation of `finish` must be prepared for these unexpected events, delaying its action until the activation finishes.

The implementations of breakpoint commands work in two steps. In the first step, the debugger plants one or more breakpoints and installs a handler that waits for the target to hit one of the breakpoints. The second step is to execute the handler when the target hits one of the breakpoints; the handler may take any action. `ldb` uses six mechanisms to implement this scheme. A *low-level breakpoint* at an instruction I causes the target to stop at I and to deliver a *low-level event* to `ldb`. An *event handler* matches a low-level event with a *breakpoint action* that implements a breakpoint command. One possible action is to notify the user that something interesting has happened; that is done by delivering a *user-level event* to an *event continuation* associated with the user interface.

Each mechanism does a simple job. A low-level breakpoint and its event handler cooperate to set a breakpoint at a single instruction. Breakpoint actions implement the different breakpoint commands, taking different actions and relying on low-level breakpoints. They decide whether control should be kept in the debugger or returned to the target. Event continuations combine the results of all the breakpoint actions. User-level events describe events at a high level; they are delivered to the user interface and printed.

The active mechanisms divide the event system into three levels. Event handlers identify the events indicating that the target has hit a breakpoint; they invoke breakpoint actions. Actions make all decisions and take all actions. For example, they evaluate conditions to decide whether to stop the target, and they deliver user-level events to event continuations. Event continuations combine the events delivered by and decisions made by breakpoint actions; they stop the target if any action has decided to do so. Low-level events are delivered to the lower two levels, event handlers and breakpoint actions; user-level events are delivered to the highest level, event continuations.

After an example and a brief discussion of events, this chapter explains how each breakpoint command is implemented by a *user-level breakpoint*, which combines a set of low-level breakpoints, a breakpoint action, and a user-level event. It describes traps, the abstraction underlying low-level breakpoints, before describing the low-level breakpoints themselves. The implementations of low-level breakpoints may require machine-dependent code, but traps are implemented using machine-dependent data only, and the higher levels are machine-independent.

When the debugger asks the debug nub to call a procedure, it has to deal with the same kinds of events as do the breakpoint commands. The expected event is that the procedure executes successfully and returns a value, but other events are possible; for example, the procedure may fail or hit a breakpoint. Procedure calls are discussed in this chapter because `ldb` manages them using the same event-handling mechanisms that it uses to implement the breakpoint commands.

7.1 Example

An abbreviated debugging session using the example program from Chapter 2 provides a concrete framework in which to discuss the implementation of breakpoints. The example program's source code is shown again in Figure 25. I plant the same two breakpoints used in Chapter 2:

```
dynastar % ldb target fib connect orchard 1316
=> DebugNub_Pause(99) <=
* 0 ... void DebugNub_Pause(int arg = 99)
ldb fib (stopped) > b fib:7 fib.c:13
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) >
```

"`=>` ··· `<=`" brackets a user-level event.

Continuing execution hits the first breakpoint:

```
ldb fib (stopped) > c
=> break  [1] at fib:7 (fib.c:6,14) <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) >
```

`ldb`'s "`take` ... `skipping` k" command can be used to skip three iterations of the loop:

```
ldb fib (stopped) > p i
int  i = 2
ldb fib (stopped) > take 1 skipping 3
           break  [1] at fib:7 (fib.c:6,14) skip 3
ldb fib (stopped) > c
=> break  [1] at fib:7 (fib.c:6,14) <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) > p i
int  i = 6
ldb fib (stopped) >
```

The target hits breakpoint 1 on the iterations in which `i` is 3, 4, and 5, but the debugger restarts it because the counter associated with the breakpoint is greater than zero, so no user-level event is delivered to the user interface.

The `next` command single steps through the next iteration:

```
ldb fib (stopped) > next
=> next fib[f7fff898] <=
* 0 <fib:5> (fib.c:7,6)  void fib(short n = 12)
ldb fib (stopped) > next
=> next fib[f7fff898] <=
* 0 <fib:6> (fib.c:6,19)  void fib(short n = 12)
ldb fib (stopped) > next
=> next fib[f7fff898]
   break  [1] at fib:7 (fib.c:6,14) <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) >
```

The `next` event announces that the program has arrived at the next stopping point; `f7fff898` identifies the frame. When `ldb` single steps to `fib:7`, two user-level events are delivered to the user interface: one associated with `next` and one with the original breakpoint at `fib:7`.

7.2 Events

Low-level events are machine-dependent and describe events at the machine level, like hitting a trap. User-level events are machine-independent and describe events at the user level, like stepping to the next stopping point.

```
void fib(short n) ^0{
  static int a[20];
  if (^1n > 20) ^2n = 20;
  ^3a[0] = a[1] = 1;
  { int i;
    for (^4i=2; ^7i<n; ^6i++)
      ^5a[i] = a[i-1] + a[i-2];
  }
  { int j;
    for (^8j=0; ^11j<n; ^10j++)
      ^9printf("%d ", a[j]);
  }
  ^12printf("\n");
^13}
```

```
main (int argc, char **argv) ^0{
  if (^1argc == 2)
    ^2fib(atoi(argv[1]));
  else
    ^3fib(10);
  return ^40;
^5}
```

Figure 25: Source file **fib.c**. Superscripts show stopping points.

Low-level events are delivered by the debug nub. As described in Chapter 6, these events carry a Unix signal number and code and enough context to recover the state of the program at the time the event occurred. A low-level event is delivered to every event handler that registers an interest. Handlers can register interest in all events, or they can restrict their interests on the basis of the events' signal number and code. For example, event handlers associated with breakpoints register interest only in the signal and code that indicate a breakpoint trap. When an event is delivered, the handler can ignore it or take action, which may include delivering a user-level event to an event continuation. For example, the breakpoint handler for the breakpoint at `fib:7` ignores trap events indicating traps at locations `fib:5` and `fib:6`, but at trap at `fib:7` makes it deliver the "`break [1] at fib:7`" event to the user interface.

User-level events are machine-independent and correspond to breakpoint commands or to errors in the target program. Examples in Section 7.1 include hitting a breakpoint and stepping to the next stopping point; page 16 shows the event associated with **finish**. The default event continuation remembers all the user-level events that are delivered during a single interaction with the target, and it notifies the user interface of the events. As shown in Section 7.1, a single low-level event like hitting a breakpoint may result in the delivery of many, one, or no events to the user interface. If any event warrants keeping control within the debugger, the default continuation does so; otherwise it returns control to the target.

A low-level event, type **Event.Low**, is delivered to a handler by calling the handler's **matches** method. **Event.Low** contains a **Wire.Event** (Section 6.1) plus a program counter and a handle that can be used to reach the target's call stack. The `matches` method returns a Boolean indicating whether the handler recognizes the event:

```
TYPE Event.Handler = OBJECT METHODS
       matches(e: Event.Low; cont: Event.Continuation) : BOOLEAN;
     END;
```

Recognition is not automatic. For example, the handler for the breakpoint at `fib:7` recognizes only trap events at `fib:7`. More than one handler can recognize the same event, e.g., `next` and `fib:7`.

A user-level event is delivered to a continuation by calling the continuation's `throw` method:

```
TYPE Event.Continuation = OBJECT METHODS
       throw(e: Event.T; stop: BOOLEAN);
     END;
```

`e` represents the event, and `stop` tells the continuation whether the occurrence of this event warrants keeping control in the debugger. Breakpoints, for example, keep control in the debugger, but trace points do not. Events and event continuations are supplied by code outside the event mechanism. `ldb` uses only one event continuation, which accumulates events and delivers them to the user interface.

7.3 User-level breakpoints

The precise set of breakpoint commands that `ldb` provides and the details of their implementations are less important than the fact that they are all implemented using a single technique. Each breakpoint command is implemented by a user-level breakpoint, which plants a set of low-level breakpoints. Each low-level breakpoint is planted at exactly one object-code location, as specified by a value of the program counter. Each is associated with a breakpoint action specified when it is planted; the breakpoint action is executed when the target hits the breakpoint. The object type used to represent user-level breakpoints is a subtype of the type of user-level events. Because the user-level breakpoint is a type of user-level event, a breakpoint action can deliver the breakpoint itself to the user interface. This section explains how user-level breakpoints are implemented by combining actions and low-level breakpoints; Section 7.5 explains the implementation of low-level breakpoints.

`ldb` offers two simple user-level breakpoints: one set at an object-code location and one set at a source location. A breakpoint set at an object-code location is associated with exactly one low-level breakpoint, whereas a breakpoint set at a source location may be associated with an arbitrary number of low-level breakpoints, because the C preprocessor may cause code from one source location to be compiled into many object-code locations.[1] These breakpoints specify the same action: the point,

[1] This duplication can occur if `#include` is used to include code. A programmer might do so as a way of instantiating generic code.

which is a type of user-level event, is delivered to the user interface. The two kinds of breakpoints have different `print` methods, which enable users to distinguish them.

```
ldb fib (stopped) > b fib:7 fib.c:13
          break  [1] at fib:7 (fib.c:6,14)
          break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
```

The object-code breakpoint's `print` method shows the object-code location and the corresponding source location: `fib:7` at line 6, column 14 of file `fib.c`. The source breakpoint's `print` method shows both the source location that was requested and the source location that corresponds to the nearest stopping point, where the breakpoint is actually planted: `fib:12` at line 13, column 2 of file `fib.c`.

The `finish` command halts the target when a particular procedure activation finishes executing. It is associated with a single low-level breakpoint, planted at the address to which control will return in the calling frame. This address is obtained by walking the stack, as described in Chapter 8. When the breakpoint is created, it saves the location of the calling frame in the Modula-3 object representing the breakpoint action; the procedure in question may be recursive, and `ldb` should not stop the target if some other activation executes the instruction at the return address. When taken, the breakpoint action verifies that the current frame matches the saved one, delivers the breakpoint to its continuation, and deactivates it—a `finish` breakpoint is a "temporary" breakpoint, taken only once. If the frame does not match, the action does nothing, waiting for the next time control reaches that address.

A `finish` breakpoint may never be taken if the target program does a `longjmp` that unwinds the stack past the activation being finished. Stack unwinding is not common in C programs, but it is used in languages with exceptions, like Modula-3. Debuggers for such languages could detect stack unwinding by setting breakpoints in the run-time exception mechanism.

`ldb` implements source-level single stepping by creating a user-level breakpoint that plants a low-level breakpoint at every stopping point in the current procedure (except the stopping point that is about to be executed). The action taken is the same action that is taken for a finish breakpoint, except the frame saved in the action object is the frame in which the single stepping takes place, not the calling frame.

Every user-level breakpoint is a member of some *breakpoint set*. A breakpoint set is associated with each target process. Breakpoint sets serve two purposes. They provide a way to enumerate all of the breakpoints associated with a target, for example for listing by the `b` command or undoing by the "`u *`" command. They hold a low-level *breakpoint implementation*, which contains the machine-dependent data and code needed to create low-level breakpoints. Simple breakpoints stay in their sets until removed by a user, but temporary breakpoints like `finish` and `next` remove themselves from their sets as part of their actions. Figure 26 shows the breakpoint set for program `fib` during

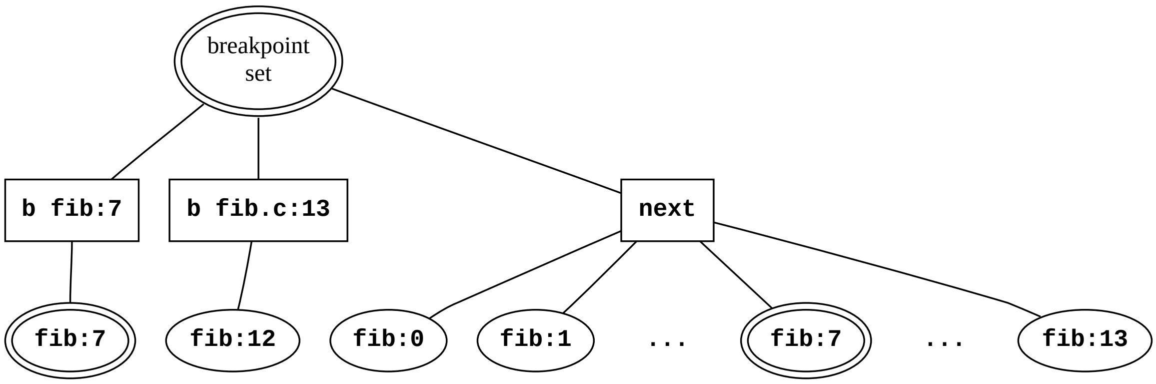


Figure 26: User-level breakpoint set with user-level and low-level breakpoints.

the single stepping. User-level breakpoints appear in boxes labeled by the commands used to create them. The ovals on the bottom row are low-level breakpoints labeled by their object-code locations.

A breakpoint set implements an event handler, which handles low-level events that are known to be breakpoint-trap events. When such an event arrives, the breakpoint-set handler distributes it to every low-level breakpoint that is associated with a user-level breakpoint in the set. This distribution makes sense because, by virtue of Modula-3 subtyping, every low-level breakpoint doubles as its own event handler (Section 7.5). Event handlers are shown as ovals in Figure 26; the handlers that recognize the trap at `fib:7` are drawn in double ovals. Their actions deliver the associated user-level breakpoints, which are also user-level events, to the event continuation. If no low-level handler recognizes a trap event delivered to the breakpoint set, the breakpoint set is inconsistent and there is a bug in `ldb`.

There are fields common to all user-level breakpoints. Every user-level breakpoint contains a condition that tells when to take the breakpoint and a counter that tells how many times the debugger should take the breakpoint before stopping. The default condition is true and the default counter is zero. Conditions and counters are changed by the `take` command, shown in Section 7.1 and in Chapter 2. `ldb`'s breakpoints also serve as trace points; the difference is that the debugger does not stop when it encounters a trace point. Every user-level breakpoint contains a Boolean that is true if the point is a breakpoint; the `print` methods print "`break`" or "`trace`" depending on its value. The action associated with simple breakpoints uses the condition, counter, and Boolean:

```
IF ⟨pt.condition is true⟩ THEN
  IF pt.skipcount > 0 THEN
    DEC(pt.skipcount);
  ELSE
    cont.throw(pt, pt.stop);
```

```
    END;
  END;
```

`cont.throw` delivers the user-level breakpoint or trace-point event to the user interface.

Every user-level breakpoint also has a Boolean `suspended` field, which is true if the breakpoint is inactive. Both user- and low-level breakpoints supply `undo` and `redo` methods, which change the state of the breakpoint from active to inactive and back. All user-level breakpoints share a single `undo` method, which in turn calls the `undo` methods of the associated low-level breakpoints, and likewise for the `redo` methods. `ldb`'s `u` and `r` commands call the `undo` and `redo` methods of the user-level breakpoints.

7.4 Traps

All low-level breakpoints are implemented by planting traps. "Suspending" a trap is the inverse of planting one; it removes the trap and restores the original instruction that was overwritten when the trap was planted. Traps may be planted and suspended by event handlers and breakpoint actions as well as by users' commands. There are no constraints on the order in which traps can be planted and suspended, except that a trap must be planted before it can be suspended. If there were constraints, each caller of the plant and suspend procedures would have to cooperate with all the other callers. Without constraints, each caller can act independently.

If plant and suspend were implemented naively, saving the overwritten instruction in local storage, the order of execution would be constrained. For example, the following sequence, which should leave memory unchanged, would leave a trap at I.

1. Plant trap 1 at I (save I in trap 1).
2. Plant trap 2 at I (save trap in trap 2).
3. Suspend trap 1 (restore I to memory).
4. Suspend trap 2 (restore trap to memory).

In other words, plant operations would not commute if they planted at the same location, nor would suspend operations. The solution is to centralize the state associated with planted traps, which `ldb` does using the *trap set* type. The implementation associates a count with each program-counter value; planting a trap increments the count, and suspending one decrements it. The underlying memory is manipulated only when the count changes from zero to nonzero or vice versa. As a result, plant operations commute regardless of location, and so do suspend operations.

As described in Section 3.2.1, it is not safe to store information about overwritten instructions in the debugger; if the debugger's machine should crash, the information needed to suspend the traps would be lost. The solution is to keep the information in the debug nub. The nub then implements a trapped memory, which has the `plant`, `suspend`, and `traps` methods shown in Section 3.2.1. To keep the nub simple, the debugger may ask the nub to plant a trap only at a location that does not already have one. To obey this restriction, the debugger must still associate a count with each program-counter value. The debugger keeps this information in a trap set. A trap set is associated with a target process, and it is stored in the user-level breakpoint set of that target.

When a debugger connects to a new target process, the `traps` method of the trapped memory is used to add traps to the trap set. The trap-set implementation makes upcalls into the user-level breakpoint implementation, creating a new object-code breakpoint to be associated with each previously unknown trap. By making these upcalls, `ldb` maintains the invariant that every trap known to the nub corresponds to some active user-level breakpoint known to the debugger.

An example shows the breakpoints that `ldb` creates when connecting to a process containing traps. Disconnecting from `fib` while two breakpoints are planted leaves traps in the process:

```
ldb fib (stopped) > b
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib.c:13 (fib.c:13,2) <fib:12>
ldb fib (stopped) > disc wait
ldb fib (disconnected) > quit
dynastar %
```

A fresh debugger for the same target program has an empty breakpoint set:

```
dynastar % ldb target fib
ldb fib (disconnected) > b
ldb fib (disconnected) >
```

Reconnecting to the original instance of `fib` shows the event that stopped it, a breakpoint at `fib:7`:

```
ldb fib (disconnected) > connect orchard 1316
=> old breakpoint at fib:7 <=
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
ldb fib (stopped) >
```

The reconnection mechanism has added two object-code breakpoints, which correspond to the traps present in the newly reconnected instance of `fib`:

```
ldb fib (stopped) > b
           break  [1] at fib:7 (fib.c:6,14)
           break  [2] at fib:12 (fib.c:13,2)
```

The reconnection mechanism does not distinguish different flavors of breakpoints; the second breakpoint is now an object-code breakpoint even though it was originally created as a source-code breakpoint. This loss of information could be inconvenient if `next` planted a large number of low-level breakpoints just before a loss of communication; upon reconnection, the breakpoints would have to be deleted individually. TTD has solved this problem by associating a set of flavors with each trap and enabling users to delete breakpoints by flavor (Redell 1989). Doing so in `ldb` would require extending the trapped-memory methods and the nub to keep track of the flavors of traps as well as their addresses.

The presence of trap instructions in target memory has no effect on the abstract memory exported by the nub. The nub's trap and abstract-memory implementations cooperate to ensure that fetches to trapped locations return the original instructions, not the traps. The debugger can therefore decode instructions without regard for the presence or absence of traps. Other debuggers solve this problem by undoing all breakpoints every time the target program stops (Linton 1990).

`ldb`'s implementation of traps uses no machine-dependent code, but it needs machine-dependent data: the sizes and bit-patterns of trap instructions on each target.

7.5 Low-level breakpoints

The problem of planting a low-level breakpoint at an instruction I can be divided into two parts. First, the debugger must get control when the target program attempts to execute I. Second, when execution is resumed, I must be executed once, but subsequent attempts to execute I must return control to the debugger. To get control at an instruction I, a debugger can overwrite I with a trap instruction, then handle the resulting trap (Caswell and Black 1990), or it can overwrite I with an instruction that branches to debugging code (Digital 1975). To resume execution, there is a wider range of choices. A debugger can return the overwritten instruction to memory, execute it by single stepping the target machine, and re-plant the breakpoint (Caswell and Black 1990). Single stepping can be avoided by transforming the overwritten instruction so that it can be correctly executed out of line (Digital 1975; Kessler 1990). A debugger could also simulate the effect of the overwritten instruction and resume execution at the succeeding instruction. Finally, some machines have special hardware that supports resumption after a break instruction (Bruegge 1985).

The single-stepping implementation of a breakpoint must handle unexpected events. When the machine is single stepped, the debugger expects the target to execute one instruction and notify the debugger of its execution. It is possible, however, that the instruction does not execute successfully, for example that the instruction being single stepped refers to an invalid address or divides by zero. The debugger must associate an appropriate event handler with each possible outcome.

`ldb` implements breakpoints by planting trap instructions. Under UNIX, a signal is delivered to a process when it attempts to execute a trap instruction. The debug nub's signal handler deals both with errors like dereferencing a nil pointer and with traps that implement breakpoints. To resume execution after a breakpoint, `ldb` uses either single stepping or a restricted form of simulation, depending on the target architecture.

`ldb` creates a low-level breakpoint at an instruction I by planting a trap at I and creating an event handler that makes it possible to resume execution without immediately trapping at I a second time. By virtue of Modula-3 subtyping, a single object acts both as the low-level breakpoint and as an event handler interested in trap events.

```
TYPE
  Breakpoint.T = Event.Handler OBJECT
    pc        :  INTEGER;
    action    :  Breakpoint.Action;
    suspended := TRUE;
    traps     :  Trap.Set;
  METHODS
    undo();
    redo();
  END;

  Breakpoint.Action = OBJECT METHODS
    take(e: Event.Low; cont: Event.Continuation);
  END;
```

`ldb` uses two subtypes of `Breakpoint.T` that provide two different implementations of breakpoints; the subtypes implement the `undo`, `redo`, and `matches` (event-handler) methods. The `undo` and `redo` methods make it possible for users to temporarily inactivate breakpoints; the `suspended` field tracks the state. The `matches` method takes the breakpoint action when it determines that the target has hit the breakpoint. The two implementations of `Breakpoint.T` offer different trade-offs between machine-independence and functionality. One is based on *follow sets*, the other on no-ops.

A follow-set breakpoint temporarily suspends the trap, single steps the target machine to execute I, then re-plants the trap. Single stepping means arranging that the target machine will trap again immediately after executing I. Some processors have a trace mode that causes the machine to trap after every instruction, but it is possible to implement single stepping without such a mode, provided one can tell what instructions might be executed immediately after I. Such instructions constitute I's follow set.

`ldb` computes follow sets using a simple model. Every instruction I has an *inline successor*, which is the next instruction in the instruction stream. If I is a branch instruction, it also has a *branch target*. Conditional branches have follow sets containing both these instructions; others have follow sets containing either one or the other.[2] If I is a computed branch, `ldb` needs an abstract memory with up-to-date register contents to compute its follow set.

A follow-set breakpoint plants a trap at I and creates a handler recognizing traps either at I or at I's follow set. The handler, when it sees a trap at I, plants traps at I's follow set, suspends the trap at I, and takes the action associated with the breakpoint. When it sees a trap at I's follow set, it suspends the traps at the follow set, re-plants the trap at I, and takes no action. When I is a

[2] Dave Redell pointed out that this model applies to machines with delayed branches, provided that the branch and the instruction in its delay slot are considered a single, two-word instruction.

computed branch, the handler recomputes I's follow set every time the trap at I is recognized. At that time, the register contents are guaranteed to be up to date.

Appendix A gives a formal model of the implementation of follow-set breakpoints. The model shows that the implementation described above is incorrect in the presence of multiple threads of execution. Although `ldb` does not support targets in which multiple threads are active concurrently, there are two ways in which `ldb` can change threads. When expression evaluation requires a procedure call, a new thread is created in the target address space, and the call executes in that new thread. The previous thread is suspended during the call. `ldb` can also change threads by changing targets. If `ldb` changes threads while traps are planted at I's follow set, `ldb` could miss a breakpoint; Appendix A shows how. Such a change occurs when a procedure is unwound after hitting a breakpoint, for example. The problem can be solved by delaying; instead of suspending the trap at I and planting traps at I's follow set when the event is handled, as described above, `ldb` waits until it is about to resume execution of the thread that hit the trap. This solution is not complete; `ldb` could still miss a breakpoint if the attempt to execute I fails and the user then changes threads. Such a situation is unlikely because the natural response to a failing instruction is to try to debug it, not to re-execute that instruction in another thread. If I fails and the user does change threads, the user must reset the breakpoint by undoing and redoing it.

Computing the follow set of an instruction I requires machine-dependent code that decodes I, determines whether it is a branch instruction, and computes the target of the branch if so. Decoding instructions is straightforward, but the code can be tedious, error-prone, and voluminous. To simplify the problem, I have designed a little language for describing instruction encodings. The language makes it possible to name different fields of an instruction and to specify patterns based on the values of different fields. To reduce the likelihood of errors, patterns can be specified in the forms of tables of opcodes, like those often found in architecture manuals. Patterns can be combined to avoid duplicating code that treats different instructions in similar ways. Appendix B describes the specification language in more detail and gives as an example the follow-set computation and complete instruction specification for the MIPS.

Even with the help of a special specification language, computing follow sets for `ldb`'s two RISC targets requires between one and two hundred lines of machine-dependent code per target. That code can be eliminated by using no-op breakpoints, which have `ldb` simulate the effect of the instruction at the breakpoint and resume execution at the following instruction. Simulation is made tractable by insisting that breakpoints be planted only at no-op instructions, which reduces simulation to advancing the program counter. This scheme relies on compiler support to be useful: `lcc` places a no-op instruction at each stopping point. Because `lcc` already places labels at stopping points, this support requires no extra implementation effort in the compiler. The no-ops increase the number of instructions by 16–19%, depending on the target. `ldb`'s implementation of no-op breakpoints is machine-independent, but it requires machine-dependent data: the size and bit pattern of a no-op instruction, which are used to prevent `ldb` from planting a trap at a location not containing a no-op.

`ldb` uses follow-set breakpoints on the MIPS and SPARC and no-op breakpoints on the VAX and 68020.

7.6 Procedure calls

Asking the nub to call a procedure presents the same problem as asking the nub to resume executing the target program: any of several different events can occur. If I adopted the same solution as for breakpoints, any PostScript code that needed to call a procedure would have to create event handlers for the completion or failure of the procedure and would have to associate actions with these handlers. Since the PostScript code that calls procedures is generated by the expression-evaluation server, such a solution is tantamount to requiring the expression server to use continuation-passing style (Appel 1992), which would complicate its implementation. `ldb` hides the necessary event handlers, making the PostScript procedure-call operator look like any other operator: it can complete execution normally by returning a value, or it can fail by raising an exception. It is appropriate to make the debugger more complex in order to simplify the expression server because to support multiple languages `ldb` might use multiple expression servers.

If the target process hits a breakpoint while executing a procedure call on behalf of `ldb`, the procedure call should not just fail; the user should be able to debug it, as shown on page 18. When the user wants to debug the procedure call, the PostScript interpreter is busy waiting for the result of the operator, and the user interface is busy waiting for the completion of the expression evaluation. A second user interface and PostScript interpreter must therefore be made available to handle the interrupted procedure call. The second user interface distinguishes itself by using `>>` in the prompt. While this prompt is used, the original user interface and interpreter are suspended awaiting the outcome of the call, which may be normal termination or unwinding. When the call completes, control reverts to the original user interface and interpreter.

`ldb` can handle an arbitrary number of suspended procedure calls. If one should fail with a fatal error, such as deferencing a nil pointer, the user can revert to the previous one (or to the original program) by using the `unwind` command. `gdb` and `dbx` either cannot debug failing procedure calls or cannot return to the previous context after debugging such a call.

Understanding the debugger's end of the procedure-call implementation requires understanding its use of concurrency. As described in Chapter 6, the debugger uses a user-interface thread to read commands and a listener thread to respond to low-level events from the target. When the target stops, the listener thread receives a message, reconstructs the event, delivers it to the appropriate event handlers, and blocks waiting for the target to start running again. The listener thread uses an event continuation that prints user-level events and changes the user interface's current focus.

A single lock is used to synchronize the two threads; the lock protects both the current focus and the state associated with the target, including the current user interface and PostScript interpreter. The user-interface thread goes to sleep during a call; when the call completes, the listener thread awakens the sleeping thread. If the call fails, the listener thread creates a new interpreter and a new user interface to debug the suspended call. The listener thread destroys the new interpreter and user interface when it receives an event indicating that the call has returned or unwound.

`ldb` represents its user interface as a Modula-3 object that can accept delivery of an event, can be asked to create a new user interface as a result of an unexpected event arriving during a procedure call, and can be asked to destroy itself when it is no longer needed. The creation and destruction methods hide the details of forking and destroying a user-interface thread to read commands.

Not all procedure calls are made by user threads; the listener thread can make a procedure call during event handling, e.g., if a procedure must be called to evaluate the condition associated with a conditional breakpoint. The listener thread cannot go to sleep after asking the nub to make the call, because there would be no thread listening for the reply. Instead, the listener thread asks the nub to make the call, then waits for the next event. If the next event is a return, the procedure-call operator returns normally, and the user thread continues to evaluate the condition. If some other event occurs, the listener thread does not attempt to create a user interface to handle the event; it is simpler to ask the nub to unwind the call. Once the call is unwound, the procedure-call operator raises an exception indicating that a failure occurred during event handling. When such a failure occurs during the evaluation of a breakpoint condition, the breakpoint is taken, and an additional user-level event is generated indicating what failure occurred.

7.7 Discussion

To create a new kind of user-level breakpoint, I have to write a procedure that determines where to plant low-level breakpoints and what actions should be associated with those breakpoints. There are many possibilities, and sophisticated users might want to program their own user-level breakpoints. Such a scheme would be most useful as part of a programmable user interface. PostScript could be used, but the operators would have to be at a higher level than the operators in `ldb`'s PostScript; for example, "plant a low-level breakpoint at line 100 of file `main.c`" is better than the current "store this value at this object-code location."

`ldb` uses only one event continuation, which accumulates events and delivers them to the user interface. `ldb` could support event-based debugging schemes by using different event continuations. For example, `ldb`'s user-level events could correspond to the "primitive events" input to a dataflow machine (Olsson, Crawford, and Ho 1991).

It might be useful to formalize the semantics of `ldb`'s breakpoint commands using a language designed to describe debugging (Crawford *et al.* 1992). A formal definition would provide a basis on which to judge the correctness of other implementations of the commands. For example, conditional breakpoints would be more efficient if the code to test the condition could run in the target process, not in the debugger. Formalization would also help clarify what the commands do in the presence of exceptions or non-local gotos.

`ldb`'s "`take` *b* `skipping` *k*" command provides a convenient way to measure the performance of `ldb` and the nub. I set a breakpoint at `i<n` in the loop "`for(i=0; i<n; i++);`" and measured how long it took to skip 1000 breakpoints. A breakpoint takes 13 msec when `ldb` uses pipes to debug a child process, 18 msec when it uses a network connection to its own machine, and 30 msec when it uses a network connection to a different machine. Experimental error is 0.2 msec. `gdb` debugging a child process using `ptrace` takes 17 msec. All measurements are elapsed time on a DEC 5000 model 240, which contains a MIPS processor.

A comparable measurement cannot be made for `dbx` because it does not provide a command that skips breakpoints, but there is another way to estimate the cost of interaction with a target program. Every time `ldb` handles the breakpoint, it goes through two cycles of planting a trap, suspending a trap, and resuming execution. `dbx` takes 8 seconds to step through 1000 machine instructions. The MIPS C compiler generates 7 ordinary instructions and one branch instruction for the loop above. On the branch instruction, `dbx` must plant and suspend two traps, not one, so its cost per cycle is between 7 and 8 msec. `ldb` takes 6.5, 9, or 15 msec per cycle.

Other debuggers' conditional breakpoints are faster than `ldb`'s; it takes `gdb` 18 msec to take a breakpoint and evaluate the condition `i % 1000 == 0`. `dbx` takes 19 msec. `ldb` takes 29, 41, or 62 msec depending on which connection mechanism is used. One reason `ldb`'s mechanism is slower is that it stores the result of the expression in the nub's global area, from which it must be fetched for the test. `ldb` must also fetch the address of the global area, so `ldb` makes two unnecessary fetches and one unnecessary store. To eliminate these unnecessary interactions with the nub, the expression server would have to use PostScript variables to hold temporaries whenever possible.

The follow-set implementation of low-level breakpoints is usually explained in terms of instruction-level single stepping. This explanation misleadingly suggests that breakpoints can be implemented using simple, sequential code. In fact, the implementation must be written in a kind of continuation-passing style, using event handlers to match continuations with events. Thinking in terms of traps at follow sets makes it easier to understand the real implementation. It also clarifies the relationship between an implementation that uses only traps and one that uses a hardware trace mode; planting or suspending traps in a follow set is equivalent to setting or clearing a trace bit in a program status word.

`ldb` does not use VAX trace mode to implement VAX breakpoints; I simply overlooked the possibility of being able to change the program status word in a non-privileged program. An implementation based on VAX trace mode would verify that the model presented in this chapter can handle machines with hardware support for single stepping.

Follow sets are useful for more than just implementing low-level breakpoints. For example, `ldb`'s `stepi` command uses follow sets to figure out where to plant low-level breakpoints. Follow sets would have to be used to implement a source-level single-stepping command that stepped into called procedures as `dbx`'s `step` command does. Follow sets could also be used to compute the control-flow graph of a procedure and to reduce the number of breakpoints planted by the source-level single-step command. The cost savings would not be measurable except when single stepping a heavily recursive procedure. Except when a statement contained a computed branch, `ldb` would be able to implement source-level single stepping by planting low-level breakpoints at only one or two stopping points, instead of planting one at every stopping point. The number of breakpoints hit by called activations would be reduced, reducing the message traffic between the nub and the debugger. I have chosen not to require follow sets in `ldb` because the payoff is not high enough to require the extra implementation effort for every target.

Chapter 8

Stack Walking

`ldb`'s users can debug any active procedure by selecting the appropriate stack frame as the current focus. A stack frame provides the context for commands, in particular, a scope for name resolution and an abstract memory against which to print variables and evaluate expressions. Much of the work of implementing stack frames is machine-independent, e.g., using the symbol table to build a scope for a stopping point. But to walk the stack and to find out where control has stopped, to restore registers from the stack, and to build an abstract memory, `ldb` needs to know the target's calling sequence; the location of saved registers within a frame and the relationships of frames to their caller's frames are machine-dependent. Modula-3 subtyping helps decompose the problem into machine-independent and machine-dependent parts, isolating the machine-dependent code in a few methods.

The stack-frame abstraction is implemented in six layers, shown in Figure 27. The layers are defined by subtyping; arrows point from subtypes to supertypes. The upper four layers are machine-independent. They use a model of frame linkage that is general enough to cover different calling sequences on different targets. The middle layer, `FrameClass.Class`, defines precisely what must be done by the machine-dependent layers. `Frame.T` is boxed because its data structures are hidden from all other layers. Only the bottom two layers are machine-dependent. The lowest contains machine-dependent code; the next contains machine-independent code parameterized by machine-dependent data. Modula-3 generics perform the parameterization, and the code is referred to as generic code throughout this chapter. All frame objects used in `ldb` have machine-dependent types from the bottom layer; by virtue of subtyping, their types also belong to each of the upper layers.

Not all of the layers shown in Figure 27 contribute to retargetability; some are used to restrict visibility. With the exception of `Frame.T`, every layer is visible to the layers below it and hidden from the layers above it; `Frame.T` is hidden from all other code. `Target.FrameP` and `Frame.Public` exist only to reveal limited information to the user interface and the rest of the debugger. The user interface sees only the top layer, and the rest of the debugger only the top two layers. Table 2 shows

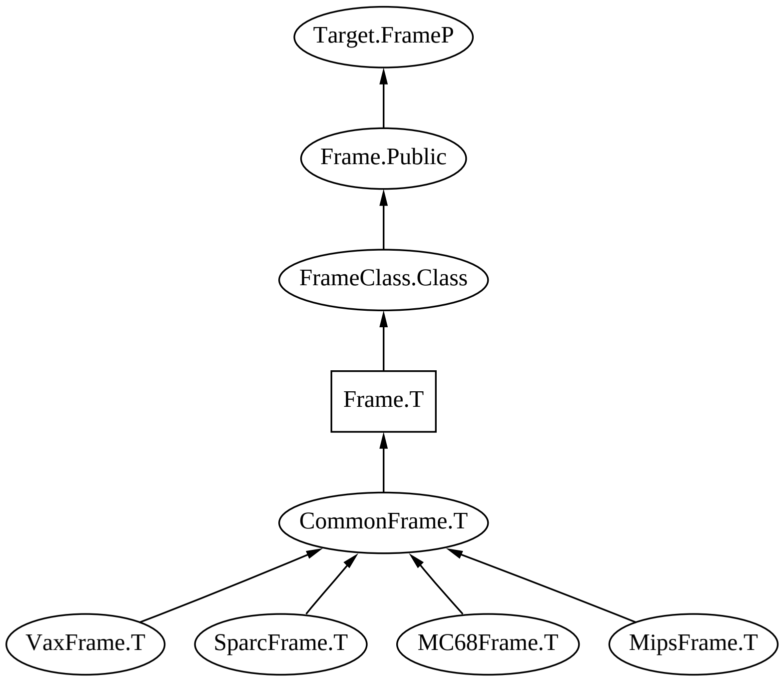


Figure 27: The layers of the stack-frame abstraction.

which parts of the debugger import each layer and what data fields are defined in each layer. A line separates the machine-independent and machine-dependent layers. Table 2 also shows where methods are defined and implemented. A Modula-3 method can be defined in an upper layer and implemented in a lower layer. A client may call a method if the method's definition is visible; its implementation need not be visible. For brevity, some of the machine-independent methods defined in `Target.FrameP` and implemented in `Frame.T` have been omitted. Because the frame objects that are actually used come from the bottom layer, they not only contain the method implementations defined in that layer; they also inherit the implementations supplied by the layers above them.

8.1 Machine-independent layers

The top layer of the stack-frame abstraction, `Target.FrameP`, is exposed to the user interface, which sees the methods shown in Table 2. The `caller` and `callee` methods provide access to adjacent frames. `visible` provides a scope, i.e. a context for name resolution, in which expressions can be compiled (see Section 5.1). `eval` evaluates compiled expressions against the abstract memory associated with the frame. `locus` returns the locus to which control will return when it re-enters the frame. The locus is often, but not always, a stopping point. For example, a procedure call can return to a locus in the middle of an expression, which is not a stopping point. In the example below, the

Layer	Imported by	Fields	Methods defined	Implemented
`Displayed.T`	user interface		`print`	

`Target.FrameP`	user interface		`caller`
			`callee`
			`visible`
			`eval`
			`locus`	

`Frame.Public`	rest of debugger	`hp`
		`ra`
		`proc`
			`m`	

`FrameClass.Class`	lower layers		`init`
			`abstractMemory`
			`callerFrame`	

`Frame.T`	—	not shown		`print`
				`caller`
				`callee`
				`visible`
				`eval`
				`locus`
				`m`
				`init`

`CommonFrame.T`	machine-dependent layer		`bindtop`
			`bindmiddle`
			`findRegSave`
				`abstractMemory`

machine-dependent	—			`callerFrame`
				`bindtop`
				`bindmiddle`
				`findRegSave`

Table 2: Locations of method definitions and implementations.

call to `fib` returns to a locus in `main` that is between two stopping points. The user-interface layer does not provide direct access to the abstract memory associated with the frame; this association is hidden from the user interface, which does not manipulate abstract memories directly.

Although it is not shown in Figure 27, `Target.FrameP` is a subtype of `Displayed.T`, as suggested in Table 2. The user interface can print a frame using its `print` method. A single `print` method is used on all machines; it is implemented in `Frame.T`. Any event that stops the target is delivered to the user interface with the frame on top of the stack. By repeated applications of the `print` and `caller` methods, the user interface can produce a stack trace like this one:

```
ldb fib (stopped) > t
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
  1 <main:2+0x28> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0x7fffba14)
  2 <mAiN:end> (MiniNub.c:8)
         int mAiN(int argc = 2, char **argv = 0x7fffba14,
                  char **envp = 0x7fffba24)
  3 <__start+0x38>  __start (no symbol table information)
ldb fib (stopped) >
```

The user interface supplies only the frame numbers and the star marking the current focus; the `print` method does the rest. A similar trace appears on page 15 in Chapter 2, but here `fib` is running on the MIPS, not on the SPARC as in Chapter 2. The source code for `fib` appears in Figure 3 on page 12.

The second layer, `Frame.Public`, represents frames to clients within the debugger below the level of the user interface. It exposes `ldb`'s model of frame linkage. This model assumes that each frame can be identified by a single pointer, called the *heavy pointer*, the value of which remains unchanged for the lifetime of the frame. All of `ldb`'s targets conform to the model, realizing the heavy pointer as either the frame pointer or the stack pointer. I use the term "heavy" pointer, not frame or stack pointer, because the frame and stack pointers are used in different ways on different architectures. A *stack frame* F is a pair (hp, ra), where hp is the heavy pointer and ra is the address to which control will return when the frame is re-activated, i.e., the locus of control described above. ra determines a procedure *proc*, that is, the procedure of which F is an activation.

`Frame.Public` has `ra`, `hp`, and `proc` fields that have the values described in the model. It also defines a method `m`, which provides an abstract memory for the frame. The registers in the abstract memory describe the values that the machine registers will have when control returns to the frame.

Implementations of commands like `finish`, which makes the target run until it returns from the current frame (Section 7.3), use the `hp` field to identify the frame. The Modula-3 object representing the frame cannot be used because `ldb` discards the object when continuing execution of the target and builds a new one when the target stops again.

`FrameClass.Class` defines the interface between machine-dependent and machine-independent layers. It is exposed only to the code that implements the machine-dependent layers, not to the whole debugger. Given a heavy pointer, return address, and abstract machine, the methods exposed to the user interface can be implemented without machine-dependent code, with the exception of the methods that find adjacent frames. Because every stack is traversed from the top down, computation is needed only to find caller frames; each frame keeps a pointer to its callee frame, if any. Therefore, machine-dependent code is needed for three tasks.

1. Create an object representing the top frame on the stack, e.g., `fib` in the example.[1] The object must contain a return address, heavy pointer, procedure, and abstract memory. The implementation is a procedure that creates an object of a machine-dependent subtype of `Frame.T`. That subtype supplies the `callerFrame` and `abstractMemory` methods, which perform the next two tasks.

2. Given a frame, create an object representing its calling frame, again containing a return address, heavy pointer, and procedure, but not necessarily an abstract memory. In the example, `fib`'s calling frame is `main`.

3. Construct the abstract memory for a frame whose abstract memory was not specified at the time it was created, e.g., `main`.

`ldb` uses different methods to compute calling frames and abstract memories because it is not always necessary to compute the abstract memory for a frame. Users can skip over several frames by specifying the number of the frame to become the current focus:

```
ldb fib (stopped) > f
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 10)
ldb fib (stopped) > 3
* 3 <__start+0x38>  __start (no symbol table information)
```

Unless the abstract memories for frames 1 and 2 contain registers needed in frame 3, `ldb` need not compute them.

The machine-dependent methods are defined by `FrameClass.Class`, which also defines a method used to initialize the hidden fields of a frame:

```
TYPE
  FrameClass.Class = Frame.Public OBJECT METHODS
    init(callee: Frame.T; absmem: Memory.T := NIL) : Frame.T;
    callerFrame()    : Frame.T;
    abstractMemory() : Memory.T;
  END;
```

The `init` method must have the callee frame unless none exists, but the abstract memory is optional. The `callerFrame` method raises an exception if there is no calling frame, i.e., at the bottom of the stack.

Except for the top frame, the problem of constructing an abstract memory can be separated from the problem of finding the caller frame. The standard reconstruction of the abstract memory for a

[1] Although the correct way to refer this frame is "the activation of `fib`", it is more convenient to refer to the frame by the procedure name alone. When each procedure has at most one activation, as in this example, there is no ambiguity.

	Preserved?	Saved?	Writable?	`ldb`'s action
saved by callee	yes	in callee's frame	yes	recovers from frame
not written	yes	no	no	re-uses callee's version
saved by caller	no	in caller's frame	yes	—
scratch	no	no	yes	unrecoverable
linkage	varies	varies	yes	computes

Table 3: Register classifications used by `ldb`.

frame, e.g., `main`, uses information in the callee frame, e.g., `fib`. The problems are combined for the top frame; when `init` is called to initialize the top frame, it must be given an abstract memory, lest the standard reconstruction try to get information from a nonexistent callee frame.

The fourth layer of the frame abstraction, `Frame.T`, implements the machine-independent methods, like `print`. This method uses the machine-dependent symbol-table information for the frame's procedure, plus the abstract memory for the frame, to print the names and values of the arguments to the procedure. Depending on the "flavor" requested by the user interface, `print` may also print the values of local variables or of registers.

The `print` method calls only other machine-independent methods. Other methods in the `Frame.T` layer, like `m` and `caller`, may call machine-dependent methods, like `abstractMemory` and `callerFrame`. Such machine-independent methods use lazy evaluation; no machine-dependent methods are called until their results are needed, and those results are saved so the machine-dependent methods are called at most once per frame. The definition of `Frame.T` includes fields that are used to save such results. The `init` method, also implemented in the `Frame.T` layer, initializes those fields. `Frame.T` is private to the `Frame` module; it is not exposed to any other part of the debugger.

8.2 Generic layer (`CommonFrame.T`)

The machine-dependent code for each target is one top-frame creation procedure and two methods, `abstractMemory` and `callerFrame`. The creation procedure and `abstractMemory` method both have the job of restoring registers, which can be implemented by generic, machine-independent code. `ldb` classifies registers by their treatment at calls. *Linkage* registers are used in stylized ways to enforce particular calling conventions; they typically include a stack pointer and a frame pointer. *Preserved* registers are general-purpose registers whose values are preserved across procedure calls. They subdivide into two groups depending on whether the callee saves and restores them or simply does not write them. The preserved registers are often called "callee-save" registers; I use a different term because `ldb` must distinguish those registers that are saved and restored from those that are not written. A general-purpose register that is not preserved is either saved and restored by the caller or is a scratch register. Table 3 shows each register class and states how `ldb` recovers registers from that class. `ldb` recovers registers when creating an abstract memory.

On all of `ldb`'s targets, the calling sequence determines which registers are linkage registers, which are preserved, and which are not preserved; these choices are the same for every procedure.[2] For example, on the MIPS, integer registers 16–23, 26–28, and 30–31 are preserved, as are floating-point registers 20–31. For each procedure, the compiler may make preserved registers either saved by callee or not written. For example, on the MIPS, `lcc` compiles `fib` to save integer registers 30 and 31. Register 30 holds local variables, either `i` or `j` depending on which is live. Register 31 holds the return address. `fib` leaves other preserved registers untouched. For each call site, the compiler may make non-preserved registers either saved by caller or scratch. The official calling sequences for `ldb`'s four targets do not prescribe treatment for registers saved by callers. `lcc` may spill temporary registers across calls (Fraser and Hanson 1992), but it never places source-level variables in such registers, so `ldb` can present a complete source-level view of a stack frame without restoring them.

The register classifications are used to subdivide the machine-dependent parts of stack walking. Creating the top frame is subdivided into four problems: get the return address and heavy pointer from the process context, combine abstract memories as shown in Section 3.2.2, create aliases for registers in the process context, and create aliases for linkage registers. Only the last subproblem requires machine-dependent code. Creating a caller frame uses the current frame to compute the return address and heavy pointer, and it requires machine-dependent code. Creating a full abstract memory for a frame also subdivides into four problems: combine smaller abstract memories, find information describing registers saved by the callee, restore the preserved registers, and restore the linkage registers. Only the second and fourth subproblems require machine-dependent code.

Some subproblems are solved by generic, machine-independent code; others require machine-dependent code. `ldb` uses generic code to recover preserved registers. Linkage registers, and other registers that require special treatment, are recovered by one of two machine-dependent methods, `bindtop` and `bindmiddle`. `CommonFrame.T` shows how the work is divided between generic and machine-dependent code.

```
TYPE
  CommonFrame.T = Frame.T OBJECT
        stackmem : Memory.T;
        aliases  : AliasedMemory.T;
      METHODS
        bindtop    (contextoffset: INTEGER     ):T := BindNothing;
        bindmiddle (contextoffset: INTEGER := 0):T := BindNothing;
        findRegSave()                              := FindNothing;
      END;
```

`stackmem` represents the underlying memory containing the frame; it is a connection to the debug nub, and it is the same in each frame. Each frame has a distinct `aliases` memory, which holds

[2] This statement does not quite apply to the MIPS, which in principle can select any register as the frame pointer, as well as using a virtual frame pointer. In practice, MIPS frames are fixed in size, and procedures use a virtual frame pointer equal to the frame size plus the value of the stack pointer.

the register aliases for that frame. `bindtop` and `bindmiddle` are used to create aliases for linkage registers and for registers that cannot be restored by generic code. `bindtop` is called by the generic procedure that creates a frame at the top of the stack; `bindmiddle` by the generic `abstractMemory` method that restores registers for frames in the middle of the stack. `bindtop` needs the address of the process context because it may need to restore registers from that context. `bindmiddle` does not need that address, but because some targets can use a single procedure to implement both methods, the signatures are made compatible. `findRegSave` is a machine-dependent method that may recover register-save information for procedures not compiled with `lcc`. For example, the VAX stores a register-save mask in memory, and the MIPS stores one in the run-time procedure table[3] (MIPS 1989, page 9-25). All three machine-dependent methods default to procedures that do nothing.

It can be useful to leave the default methods in place until the last stages of retargeting `ldb`. With the defaults in place, `ldb` still works, but it does not restore as many registers as possible. If no machine-dependent `findRegSave` is implemented, `ldb` might not be able to recover all local variables of a procedure that calls a procedure not compiled with `lcc`. For example, if the program `fib` were interrupted during `fib`'s call to `printf`, `ldb` would be unable to recover the value of the local variable `j`. `j` is in register 30, and without register-save information for `printf` there is no way to tell whether register 30 is saved on the stack or left untouched. If no machine-dependent `bindmiddle` is implemented, the linkage registers do not appear in the abstract memory. Since local, automatic variables are addressed by indirection with respect to one of the linkage registers, omitting `bindmiddle` makes it impossible for PostScript code to fetch the values of such variables. `ldb` implements `findRegSave`, `bindmiddle`, and `bindtop` for all targets except the 68020, for which it does not implement `findRegSave`. Finding register-save information for the 68020 would entail searching the instruction stream for move-multiple instructions.

`CommonFrame.T` contributes to retargetability by making it possible to do most stack walking using generic, machine-independent code, which requires as a parameter a machine-dependent *configuration interface* containing only one or two dozen lines of machine-dependent data. The primary task of the generic code is to build abstract memories that contain the proper aliases for registers. For frames in the middle of the stack, e.g., `main`, registers are recovered from the callee's frame, e.g., `fib`. For the frame on the top of the stack, e.g., `fib`, registers are recovered from the process context. The rest of this section explains in detail how the generic code handles these two cases. When the generic code uses machine-dependent data, examples of that data are shown.

[3]The MIPS run-time procedure table is a data structure in target memory, created by the linker, that contains machine-dependent data describing every procedure in the program, including register-save information, frame sizes, information about which registers hold return addresses, and more. The run-time procedure table is intended to enable language run-time systems to walk the stack, e.g., to implement exceptions.

8.2.1 Preserved registers

`lcc` divides registers into *register sets*. This division is orthogonal to the division into linkage, preserved, and not preserved; it reflects the structure of the underlying hardware. The VAX has a single register set, and the SPARC and MIPS each have two: integer and floating-point registers. The 68020 has three sets: data, address, and floating-point registers. Each register set corresponds to a space in an abstract memory. A list of such spaces is one of the specifications used to construct abstract memories, as shown in Section 3.2.2. It is possible for the debugger to add spaces that do not correspond to register sets known to the compiler. The specification for the MIPS is

```
  AliasSpaces = ARRAY OF ['a'..'z'] { 'r', 'f', 'x' };
```

Spaces `r` and `f` correspond to the two register sets known to `lcc`; space `x` provides access to the program counter and virtual frame pointer from PostScript.

To restore preserved registers, `ldb` must know which registers were saved. In PostScript, `ldb` uses masks to represent sets of registers. In Modula-3, it uses type `RegSet.T`, a set of integers between 0 and 31. `ldb` also must know where registers were saved. Its generic code assumes that a single, contiguous part of the stack frame is reserved for registers saved by the callee, and that registers are saved in order, either increasing or decreasing, with no gaps for registers not saved. All of `ldb`'s targets use calling sequences that satisfy this assumption.[4] The amount of space needed per register and the order used are specified by the constant `EeSizes` in the configuration interface:

```
CONST
  EeSizes     = ARRAY OF INTEGER    {  4,    4,  0  };
  Preserved   = ARRAY OF RegSet.T   { RegSet.T { 16..23, 26..28, 30..31 },
                                      RegSet.T { 20..31 }, RegSet.T {} };
```

Each element in `EeSizes` applies to the corresponding space in `AliasSpaces`, and similarly for the other constant arrays in the configuration interface. The two `4`s specify that the callee saves integer and floating-point registers four bytes apart, with higher-numbered registers at larger addresses. The opposite order would be specified by `-4`. The constant `0` indicates that `x` registers are never saved in the frame; they are recovered by machine-dependent means, i.e., the `bindmiddle` method. `Preserved` shows which registers in each set are preserved.

The generic implementation of `abstractMemory` does the actual restoring. Section 3.2.2 shows the part of the implementation that creates the abstract memory; Figure 28 shows the full implementation, including the code that creates aliases for the abstract memory. The generic `abstractMemory` is used only for frames in the middle of the stack, so there is always a callee procedure, computed on line 3. When `AbstractMemory` is called to create an abstract memory for `main`, the callee frame is

[4] `ldb` could support machines that did not satisfy this assumption, but the `abstractMemory` methods for such machines would not be able to use the current generic, machine-independent code.

```
1  PROCEDURE AbstractMemory(frame: CommonFrame.T) : Memory.T =
2  VAR this     := AliasedMemory.New(frame.stackmem, AliasSpaces, AliasSizes);
3      callee   := NARROW(frame.callee(), CommonFrame.T);
4      prevbind : AliasedMemory.T;
5  BEGIN
6    ⟨If necessary, find register-save info and set callee.proc.regs⟩
7    FOR i := 0 TO LAST(callee.proc.regs^) DO
8      WITH regs      = callee.proc.regs[i],
9           preserved = Preserved[i],
10          space     = AliasSpaces[i]
11     DO
12       ⟨If not all preserved regs. are saved by callee, make prevbind hold bindings of callee⟩
13       FOR reg := 0 TO AliasSizes[i] - 1 DO
14         IF reg IN regs.saved THEN
15             this.bind(reg, space, Location.Absolute(callee.hp + regs.base
16                       + EeSizes[i]*RegSet.Count(regs.saved, reg)), 'd'));
17         ELSIF reg IN preserved AND prevbind.bound(reg, space) THEN
18           this.bind(reg, space, prevbind.binding(reg, space));
19         END;
20       END;
21     END;
22   END;
23   frame.aliases := this;
24   EVAL frame.bindmiddle();
25   RETURN Memory.Join(RegisterMemory.New(this, RegSpaces, RegSpec),
26                      frame.stackmem);
27 END AbstractMemory;
```

Figure 28: The generic implementation of the `abstractMemory` method.

`fib`. `lcc` supplies with the callee procedure both a register-save mask and the offset from the heavy pointer at which the lowest-numbered register is saved. This information appears in Figure 28 as the set `regs.saved` and the integer `regs.base`, defined on line 8 and used on lines 14–16. Combined with the size information from the configuration interface, it enables `ldb` to compute the location of every saved register.

Registers for `main` are restored from `fib`'s frame, which is shown in Figure 29. `fib`'s register-save mask marks integer registers 30 and 31, and its base for integer registers is 16, showing the generic code where to find those registers with respect to `fib`'s heavy pointer. `fib`'s frame also contains space for an argument-build area, used when `fib` calls `printf`.

Lines 7–22 of Figure 28 loop through register sets; lines 13–20 loop through registers in a set. Line 7 loops not through all known register sets, but only through those for which register-save

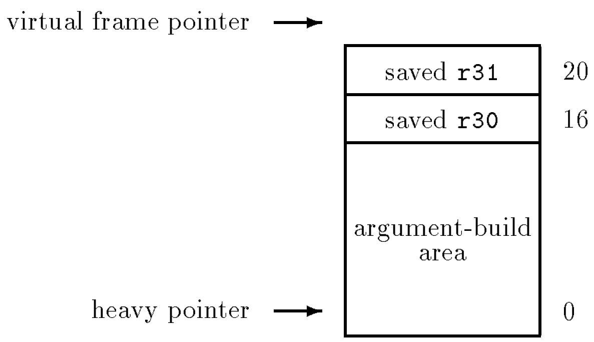


Figure 29: `fib`'s stack frame on the MIPS.

information is available. If no register-save information is available, `callee.proc.regs^` is empty, and none of the code on lines 7–22 is executed.

If a register is saved by the callee (Figure 28, line 14), lines 15–16 create an alias to the appropriate location in memory. `regs.base` is added to the heavy pointer to find the location of the lowest-numbered register in the register-save area.[5] `RegSet.Count` returns the number of registers numbered lower than `reg` that have also been saved. This number, multiplied by `EeSizes[i]`, yields the offset of `reg` within the register-save area. The code on lines 15–16 may restore some non-preserved registers; some VAX procedures save some non-preserved registers.

If a register is not saved, but is a preserved register, the callee may not change its value, so the alias from the callee frame can be re-used, provided it exists. `prevbind`, set on line 12 of Figure 28, holds the alias memory of the callee frame. If a register is preserved, line 17 tests `prevbind` for the existence of an alias; if one exists, a copy is created on line 18. Lines 15, 17, and 18 use the aliased-memory methods whose definitions are shown in Section 3.2. `bind` creates a new alias, `bound` tests to see whether an alias exists, and `binding` returns the underlying location of an existing alias.

The loop on lines 7–22 creates aliases only for saved and preserved registers. To create aliases for linkage registers, `abstractMemory` calls the machine-dependent `bindmiddle` method (line 24). This method finds the alias memory in the `aliases` field of the frame (line 23). When all aliases are created, the generic method assembles and returns the full abstract memory (lines 25–26), just as in Section 3.2.2.

In the example, the branch on lines 15–16 is taken for integer registers 30 and 31; aliases are created for those registers that refer to locations in `fib`'s frame. For example, register 30 is an alias for offset `7fffb9c0` in the data space, which is in `fib`'s frame. The branch on line 18 is taken for the other preserved registers; the aliases created for those registers are copies of the aliases used in `fib`'s frame, which refer to locations in the process context. For example, register 23 is an alias for

[5] In this and later examples, signed arithmetic is used for clarity. Actual arithmetic involving return addresses or heavy pointers must be unsigned.

offset `7fffb8ec` in the data space, which is in the process context. Non-preserved registers are not restored.

On the MIPS, `bindmiddle` creates aliases for two linkage registers, the return address and heavy pointer (stack pointer), and two special registers, the virtual frame pointer and register zero. Register zero is always zero. This MIPS-dependent `BindLinkage` implements both the `bindmiddle` and `bindtop` methods:

```
PROCEDURE BindLinkage(frame: MipsFrame.T; ...) : CommonFrame.T =
BEGIN
  frame.aliases.bind( 1, 'x', Location.Immediate(Memory.Value { n := frame.ra }));
  frame.aliases.bind(29, 'r', Location.Immediate(Memory.Value { n := frame.hp }));
  frame.aliases.bind( 0, 'x', Location.Immediate(Memory.Value { n := frame.hp +
                          GetInteger(RPTProc(frame).attributes, "framesize")}));
  frame.aliases.bind( 0, 'r', Location.Immediate(Memory.Value { n := 0 }));
  RETURN frame;
END BindLinkage;
```

The locations created by `Location.Immediate` hold the values given as arguments; they do not correspond to real locations in memory. The `bind` method of `frame.aliases` makes the registers aliases for these immediate locations, so that, for example, register 29 is an alias for the heavy pointer. `RPTProc(frame)` uses the MIPS run-time procedure table, if necessary, to add machine-dependent data to the PostScript dictionary associated with the current procedure. That data includes the size of the frame, which is used to compute the virtual frame pointer.

`ldb`'s "`F Frame.Registers`" command prints a frame and includes the values of registers. Registers that it cannot restore are printed as double dashes:

```
ldb fib (stopped) > F Frame.Registers
* 1 <main:2+0x28> (fib.c:17,4)
         int main(int argc = 2, char **argv = 0x7fffba14)
    $pc = 100001a4   $vfp = 7fffb9e0    $sp = 7fffb9c8
    $r0 = 00000000    $r8 =    --      $r16 = 00000000   $r24 =    --
    $r1 =    --       $r9 =    --      $r17 = 100342c4   $r25 =    --
    $r2 =    --      $r10 =    --      $r18 = 1000b730   $r26 = 00000000
    $r3 =    --      $r11 =    --      $r19 = 100342a4   $r27 = 00000000
    $r4 =    --      $r12 =    --      $r20 = 10034d54   $r28 = 10012400
    $r5 =    --      $r13 =    --      $r21 = 1001f544   $r29 = 7fffb9c8
    $r6 =    --      $r14 =    --      $r22 = 00000000   $r30 = 7fffbb34
    $r7 =    --      $r15 =    --      $r23 = 00000b63   $r31 = 100001a4
```

The first line of registers shows the two `x` registers, the program counter and the virtual frame pointer, as well as `$sp`, the heavy pointer. Registers 0 and 29 (zero and the stack pointer) are restored not by the generic code, but by the `bindmiddle` method. I have omitted the display of the floating-point registers.

8.2.2 The frame on top of the stack

`ldb` also uses generic code to implement the procedure that creates the top frame on the stack, e.g., `fib`. The frame-creation procedure must restore registers from and find the heavy pointer and program counter in the process context. A description of the process context, identifying the names of the fields and their offsets from the beginning of the context, appears in the configuration interface. On the MIPS, the process context is the system's `struct sigcontext`, described by

```
TYPE  Context = {onstack, mask, pc, regs, mdlo, mdhi, ownedfp,
                 fpregs, fpc_csr, fpc_eir, cause, badvaddr, badpaddr};
CONST Offsets = ARRAY Context OF INTEGER
                {0, 4, 8, 12, 140, 144, 148, 152, 280, 284, 288, 292, 296};
```

The description of the context for the 68020, which is shown on page 89 in Section 6.4.2, is

```
TYPE  Context = {data, address, pc, floatx};
CONST Offsets = ARRAY Context OF INTEGER {0, 32, 64, 68};
```

These descriptions are generated automatically by a PostScript program that reads the symbol-table entry of the context defined in the debug nub.

The process context is located in the target data space. If its address is known, the locations of the various fields can be calculated. The `CommonFrame` interface defines an object that represents an integer stored in the process context. The address of the context must be specified in order to read or write such an integer:

```
TYPE
  CommonFrame.Integer = OBJECT METHODS
    get(m: Memory.T; contextoffset: INTEGER) : INTEGER;
    put(m: Memory.T; contextoffset: INTEGER; val: INTEGER);
  END;
```

The configuration interface defines two variables, `pc` and `hp`, that enable the generic frame-creation procedure to find the program counter and heavy pointer in the process context.

```
VAR pc, hp: CommonFrame.Integer;
```

The generic code provides a subtype used to denote an integer of a known type at a known offset in a known field. The field is a value of type `Context` from the configuration interface, `Config`:

```
TYPE
  GCommonFrame.Integer = CommonFrame.Integer OBJECT
    type   : Memory.Type;
    offset : INTEGER := 0;
    field  : Config.Context;
  END;
```

```
1  PROCEDURE New(target: Target.T; contextoffset: INTEGER) : Frame.T =
2  VAR m    := target.m;
3      this := AliasedMemory.New(m,AliasSpaces,AliasSizes);
4  BEGIN
5    ⟨use process context to create register aliases in this⟩
6    WITH pc = Config.pc.get(m, contextoffset),
7         hp = Config.hp.get(m, contextoffset) DO
8      RETURN NEW(FrameType, ra := pc, hp := hp, stackmem := m, aliases := this,
9                 proc := TargetState.ProcedureAt(target, pc), target := target)
10          .bindtop(contextoffset)
11          .init(NIL,Memory.Join(RegisterMemory.New(this,RegSpaces,RegSpec),m));
12   END;
13 END New;
```

Figure 30: Generic procedure for creating the top frame on the stack.

When the generic code is instantiated, objects of the instantiated type are used to represent `pc` and `hp`. On the MIPS, the program counter is in the `pc` field, and the heavy pointer is register 29, located in the `regs` field:

```
pc := NEW(MipsGCF.Integer, field  := Context.pc,   type := Memory.Type.I32);
hp := NEW(MipsGCF.Integer, field  := Context.regs, type := Memory.Type.I32,
                           offset := 29*4);
```

`MipsGCF` is the generic instantiation, for the MIPS, of `GCommonFrame`.

Figure 30 shows the generic frame-creation procedure. The argument of type `Target.T` contains the global state associated with the target, including a connection to the debug nub, in `target.m`, and the target's linker table and top-level dictionary. The abstract memory is created using the same technique shown above and in Section 3.2.2; the code appears on lines 3 and 11 of Figure 30. Line 11 shows how the `init` method implemented of the `Frame.T` type is used; the first argument of `NIL` indicates that there is no callee frame, and the second argument is the abstract memory of the newly created frame. Lines 6 and 7 show how `pc` and `hp` from the configuration interface are used to fetch the program counter and heavy pointer from the process context. Lines 8 and 9 create the frame itself. The type of the frame is specified in the configuration interface:

```
TYPE FrameType = MipsFrame.T;
```

When creating the frame, it is not enough just to compute the return address, heavy pointer, and procedure required by the `Frame.Public` definition. The creating procedure must initialize data defined by other layers of the frame abstraction, including `stackmem` and `aliases`, defined in the generic layer, `target`, defined in the top layer, and the private data of the `Frame.T` layer, initialized by the `init` method. `TargetState.ProcedureAt` uses the linker table to find the procedure containing the program counter `pc`, as described in Section 4.2.

The registers in the top frame are restored differently from registers in frames in the middle of the stack. There is no callee frame holding saved registers; registers are saved in the process context. No register-save masks are used, because register sets are assumed to be saved wholesale; either all of the registers are in the process context, or none are. This assumption holds on every machine but the SPARC; on the SPARC, the machine-dependent `bindtop` method is used to restore ordinary registers as well as linkage and special registers. Two constants from the configuration interface specify where in the context register sets are saved and how they are arranged. On the MIPS, integer and floating-point registers are stored in the `regs` and `fpregs` fields of the process context. They are stored four bytes apart, with higher-numbered registers at higher addresses:

```
AliasSpaces = ARRAY OF ['a'..'z'] { 'r', 'f', 'x' };
RegBases    = ARRAY OF Context    { Context.regs, Context.fpregs, Context.pc };
RegShifts   = ARRAY OF INTEGER    {  4,     4,  0  };
```

I have repeated the definition of `AliasSpaces` to make it clear what information applies to which register set. The shift of `0` indicates that the `x` registers are not saved in the context; they are created by the machine-dependent `bindtop` method, called on line 10 of Figure 30. On the MIPS and 68020, a single procedure serves as both `bindtop` and `bindmiddle` methods; the MIPS implementation, `BindLinkage`, is shown above.

The generic code that uses `AliasSpaces`, `RegBases`, and `RegShifts` to restore the registers for the top frame is simpler than the analogous code in the `abstractMemory` method:

⟨*use process context to create register aliases in* `this`⟩≡
```
FOR i := FIRST(AliasSpaces) TO LAST(AliasSpaces) DO
  IF RegShifts[i] # 0 THEN
    WITH base = ContextLocation(contextoffset, RegBases[i]) DO
      FOR j := 0 TO AliasSizes[i] - 1 DO
        this.bind(j, AliasSpaces[i], Location.Shifted(RegShifts[i]*j, base));
      END;
    END;
  END;
END;
```

`ContextLocation` produces the location of a field within the process context, given the address of that context.

For `fib`'s frame, this generic code restores all the integer registers, not just the preserved registers. The `bindtop` method creates aliases for the `x` registers, so all the register values are known:

```
ldb fib (stopped) > F Frame.Registers
* 0 <fib:7> (fib.c:6,14)  void fib(short n = 12)
    $pc = 100000f0   $vfp = 7fffb9c8    $sp = 7fffb9b0
    $r0 = 00000000    $r8 = 7fffffff   $r16 = 00000000   $r24 = 00000002
    $r1 = 10010008    $r9 = 7ffffff5   $r17 = 100342c4   $r25 = 00000008
    $r2 = 0000000c   $r10 = 00000032   $r18 = 1000b730   $r26 = 00000000
    $r3 = 00001000   $r11 = 00000002   $r19 = 100342a4   $r27 = 00000000
    $r4 = 0000000c   $r12 = 00000032   $r20 = 10034d54   $r28 = 10012400
    $r5 = 0000000c   $r13 = 100025e4   $r21 = 1001f544   $r29 = 7fffb9b0
    $r6 = 00000000   $r14 = 0ccccccc   $r22 = 00000000   $r30 = 00000003
    $r7 = 0000000c   $r15 = 00000001   $r23 = 00000b63   $r31 = 100001a4
```

Except for registers 30 and 31, the preserved registers have the same values in `fib`'s frame that they have in `main`'s frame.

8.3 Machine-dependent layer

The lowest layer of the frame abstraction is the machine-dependent layer, in which there is a different subtype for each target, e.g., `MipsFrame.T`. The subtype specifies the implementations of five methods: `abstractMemory`, `bindtop`, `bindmiddle`, `findRegSave`, and `callerFrame`. As described above, `abstractMemory` is implemented by generic, machine-independent code, parameterized by the configuration interface; the others are implemented by machine-dependent procedures.

The MIPS implementation of `bindtop` and `bindmiddle` appears above; the 68020 implementation is even simpler, binding only the two linkage registers:

```
PROCEDURE BindLinkage(frame: T; ...) : CommonFrame.T =
BEGIN
  frame.aliases.bind(0, 'x', Location.Immediate(Memory.Value { n := frame.ra }));
  frame.aliases.bind(6, 'a', Location.Immediate(Memory.Value { n := frame.hp }));
  RETURN frame;
END BindLinkage;
```

If the callee procedure is not compiled with `lcc`, no register-save information is provided by the compiler. It may still be possible to get the information by calling the machine-dependent `findRegSave` method:

⟨*If necessary, find register-save info and set* `callee.proc.regs`⟩≡

```
  IF callee.proc.regs = NIL THEN
    callee.findRegSave();
  END;
```

This machine-dependent method is called at most once per procedure, and only when no information is supplied by the compiler. The implementation for the VAX is the simplest; the low-order 12 bits of the first word of the procedure are a register-save mask. The code uses `Memory.FetchAbs` to specify offset and space directly, avoiding the allocation of a Modula-3 object implied by a call to `Location.Absolute`.

```
PROCEDURE FindRegSave(frame: T) =
VAR proc      := frame.proc;
    entryword := Memory.FetchAbs(frame.stackmem, proc.entry, 'd',
                                                   Memory.Type.I16).n;
    saved     := RegSet.FromMask(Word.Extract(entryword, 0, 12));
BEGIN
  proc.regs := NEW(REF ARRAY OF RegInfo, 1);
  proc.regs[0].saved := saved;
  proc.regs[0].base  := 20;
END FindRegSave;
```

The implementation for the MIPS is slightly more complicated; it must get the information from the run-time procedure table. On the SPARC, the register windows keep the implementation simple; optimized leaf routines save no registers, and others save registers 16–31 starting at the frame pointer. The SPARC operating system flushes register windows to the stack before it delivers a signal to the nub. `ldb` does not provide a machine-dependent `findRegSave` method for the 68020 because, as noted above, doing so would entail searching the instruction stream for move-multiple instructions. The default method, used on the 68020, returns an empty `regs` array; none of the code on lines 7–22 of Figure 28 is executed, and no aliases for registers are created.

The last part of the stack-walking problem is finding caller frames: given a frame $F = (hp, ra)$, compute $\overline{F} = (\overline{hp}, \overline{ra})$. For example, given `fib`'s frame, compute `main`'s frame. I have not identified a common, machine-independent part of this job, but the cost is low; the machine-dependent implementations of the `callerFrame` methods are 15–31 lines of Modula-3.

On the MIPS, the heavy pointer is the stack pointer, except in procedures with varying-size frames. In those procedures, the heavy pointer is the "frame register." The "pc register," which holds the return address, can also vary. Most procedures use register 31, but the procedure on the bottom of the stack uses register 0. The run-time procedure table identifies the pc register, frame register, and frame size of each procedure. Although the pc register is a linkage register, it is treated as a preserved register; leaf routines do not write it and other routines save its initial value in their frames. The information from the run-time procedure table is needed to compute the caller frame:

$$\begin{array}{lll} \overline{ra} & := & \text{"pc register" (may be saved in frame)} \\ \overline{hp} & := & hp + proc.framesize \\ \text{fail} & & \text{if } \overline{ra} = 0. \end{array}$$

An attempt to compute the caller frame fails when `ldb` reaches the bottom of the stack; there is no caller frame, and the `callerFrame` method raises an exception. In the example, an exception is raised when the machine-independent code calls the `callerFrame` method of the object representing `start`'s frame.

On the SPARC, the choice of heavy pointer is arbitrary; I use the stack pointer. For non-leaf routines,

$$\begin{array}{lll} \overline{hp} & := & M[hp + (30-16)*4] \\ \overline{ra} & := & M[hp + (31-16)*4] \\ \text{fail} & & \text{if } \overline{ra} = 0. \end{array}$$

$M[a]$ indicates the value of the word stored in the data space at offset a. The register-window mechanism saves registers 16–31; the heavy pointer and return address are registers 30 and 31.

SPARC leaf routines use their caller's frame:

$$\begin{array}{lll} \overline{hp} & := & hp \\ \overline{ra} & := & \text{register } \mathtt{o7} \\ \text{fail} & & \text{if } \overline{ra} = 0. \end{array}$$

On the 68020, the heavy pointer is the frame pointer:

$$\begin{array}{lll} \overline{ra} & := & M[hp + 4] \\ \overline{hp} & := & M[hp] \\ \text{fail} & & \text{if } hp = 0 \text{ or } \overline{ra} = 0. \end{array}$$

On the VAX, the heavy pointer is the frame pointer:

$$\begin{array}{lll} \overline{ra} & := & M[hp + 16] \\ \overline{hp} & := & M[hp + 12] \\ \text{fail} & & \text{if } ra = 0. \end{array}$$

The mapping $ra \rightarrow proc$ is machine-independent, computed by `TargetState.ProcedureAt`, as shown Figure 30.

Like the top-frame procedure, the `callerFrame` method must initialize the frame it creates. The complexity of `callerFrame` varies with the target. The implementation for the 68020 is the simplest.

```
PROCEDURE CallerFrame(frame: MC68Frame.T) : Frame.T =
VAR ra, fp : INTEGER;
BEGIN
  IF frame.hp # 0 THEN
    ra := Memory.FetchAbs(frame.stackmem, frame.hp+4, 'd', Memory.Type.I32).n;
    fp := Memory.FetchAbs(frame.stackmem, frame.hp,   'd', Memory.Type.I32).n;
    IF ra # 0 THEN
      RETURN NEW(MC68Frame.T, ra := ra, hp := fp, stackmem := frame.stackmem,
                 proc := TargetState.ProcedureAt(ra)).init(frame);
    END;
  END;
  RAISE Error.UserE(Error.UC.StackBottom);
END CallerFrame;
```

The new frame is created much as the top frame is (Figure 30, lines 8–9), but it is initialized differently; the callee frame is supplied, but not an abstract memory. Abstract memories for frames other than the top are computed only on demand; as shown in Figure 28, this computation initializes the `aliases` field, which need not be initialized by the `callerFrame` method.

8.4 Discussion

`ldb` does not handle caller-save registers. This restriction does not prevent it from working with `lcc`, because `lcc` puts no source-level variables in caller-save registers. `ldb`'s current approach could be generalized to handle caller-save registers. Such generalization would require changes not just to the generic code, but to the PostScript symbol tables, because the compiler would have to provide register-save information on a per-call-site basis. The current interface between the compiler and debugger cannot refer to call sites, only to stopping points, and there can be more than one call site per stopping point. This problem could be handled by having the compiler make every call site a stopping point.

Generalizing `ldb`'s current approach would be a mistake because the current approach assumes that registers are saved and restored at every call. Caller-save registers are more effective when some saves and restores can be eliminated, e.g., by splitting the live ranges of variables (Chow and Hennessy 1990). For example, a variable may be moved from a register to memory and stay in memory over a span of several calls. When the variable is restored, it need not be restored to the register it occupied before the call. Taking this view makes it clear that the problem is not one of restoring caller-save registers; it is one of finding the values of variables. The problem should be solved not by having the debugger restore registers but by having the compiler tell the debugger how the locations of variables change over time, as described in Section 4.7. This solution has the advantage of being machine-independent, and it does not require any change to the stack-walking code.

The design of the frame abstraction illustrates an engineering trade-off between simplicity of specification and simplicity of implementation. In an earlier version of `ldb`, each machine-dependent frame type implemented `FrameClass.Class` directly, omitting the `CommonFrame.T` layer. The implementation task was more simply specified, because the machine-dependent code had only to implement the `abstractMemory` and `callerFrame` methods and a creation procedure, each of which does an easily described job. The implementation itself was more complex. Because the restoration of registers was done in machine-dependent code, its implementation was combined with the reconstruction of register-save information. Similarly, the creation of the topmost stack frame was combined with the restoration of registers from the process context. In both cases, the combined problems were solved somewhat differently on all four targets.

The current model, with the additional `CommonFrame.T` layer, is more complex because it identifies more subproblems, but there is less machine-dependent code, because many of the subproblems can be solved using generic code. As noted above, creating the top frame is subdivided into four problems: get the return address and heavy pointer from the process context, build the framework of abstract memories, create aliases for registers in the process context, and create aliases for linkage registers. Only the last subproblem requires machine-dependent code. Similarly, the problem of restoring the registers for a frame is subdivided into three problems: find information describing registers saved by the callee, restore the preserved registers, and restore the linkage registers. Only the first and third subproblems require machine-dependent code. This decomposition also provides a better path for retargeting because the first subproblem must be solved only when no information is provided by the compiler, i.e., when a function is not compiled with `lcc`.

The model of frame linkage presented in this chapter assumes that procedures are called atomically. In fact, on all targets except the VAX, the process is not atomic. First the program counter changes, then a new frame is allocated, and then the registers are saved. On the SPARC, this sequence takes only two instructions, but it can take many on the MIPS, because only one register can be saved at a time. When a target is stopped in an intermediate state, after the change of program counter but before registers are saved, `ldb`'s aliases for registers do not reflect the actual locations of those registers. If the target is stopped before the stack frame is allocated, `ldb` walks the stack incorrectly. The problem is not severe because `ldb` normally does not see targets in such an intermediate state. It can happen if the target is interrupted in such a state, if a user makes injudicious use of `ldb`'s ability to step forward one instruction, or if a faulty program branches to an invalid address. Similar intermediate states exist because exit sequences are not atomic either, and they cause similar problems. A better, but still retargetable, implementation of stack walking could be made by having the compiler identify the program-counter values that correspond to these intermediate states. Because `lcc`'s front end also assumes that procedure call is an atomic operation, the changes would have to be made to the back end. Also, because the topmost procedure on the stack might not have been compiled with `lcc`, for full generality it would be necessary to write machine-dependent code to examine the instruction stream and determine whether the target was in an intermediate state.

The error that most commonly puts a target program in an intermediate state is an indirect call through a function pointer that is zero; the program counter changes, but no frame is allocated and no registers are saved. `ldb` detects this case and correctly recovers the top frame from the process context. It refuses requests to walk the stack because the program counter is not in a valid procedure. `ldb` should be extended to handle this common case, which can be handled correctly without help from the compiler. Still, `ldb` is no worse than other debuggers. On the SPARC, `gdb` fails to walk the stack. Sun `dbx` walks the stack correctly if the program is compiled with `dbx`-style debugging symbols, but it crashes otherwise. On the MIPS, neither `gdb` nor DEC `dbx` can walk the stack, even if their debugging symbols are present.

The intermediate-state problem affects all debuggers, not just `ldb`. A proper solution requires restrictions on entry and exit sequences that make it easy to identify the transitions between states. Such restrictions must apply not only to ordinary leaf and non-leaf procedures, but also to other frames that might appear on the stack, like frames used at the bottom of the stack or to support signal handlers. Restricted calling sequences need not degrade performance; DEC has a calling sequence that meets these requirements without affecting performance (Digital 1992, Section 3).

The SPARC keeps information about saved registers both on the stack and in register windows. In general, the SPARC process context includes the state of some number of register windows, a number given by the operating system when it delivers a signal. In practice, the operating system flushes the register windows to the stack when an interrupt occurs, and the process context contains no register windows. The exception occurs when the operating system is unable to write the stack, a problem typically caused by stack overflow. In that case, the nub's signal handler cannot go on the stack, and it is impossible to debug anyway. The nub could arrange for its handler to be executed on a "signal stack," but that scheme creates new problems; the operating system does not detect overflow on the signal stack, so it would be impossible to call procedures reliably on that stack. `ldb` is no worse off than Sun `dbx`, which also cannot debug a process whose stack has overflowed. Rather than handle the general case, `ldb`'s SPARC-dependent stack-walking code simply verifies that the register windows have been flushed to the stack.

Chapter 9

`ldb`'s PostScript

"PostScript" does not necessarily mean a language for printing and graphics. PostScript is a simple, strongly typed, postfix language that is usually delivered with a large subroutine library, or "set of operators," for printing. `ldb` retains the core language, but it uses a different subroutine library more suited to debugging.

`ldb` uses its PostScript interpreter to do four jobs: reading symbol tables, transferring information from the compiler to the expression-evaluation server, evaluating expressions, and printing values. These jobs require about 1000 lines of supporting PostScript code in addition to the PostScript symbol tables generated by the compiler, The compiler and expression server rely on about 600 lines, divided equally among three tasks: printing values (Section 4.4), transmitting symbols and types to the expression server (Section 5.3.1), and implementing `lcc`'s intermediate language (Section 5.3.3). This code depends not just on C but also on `lcc`; all 600 lines would have to be rewritten to support another C compiler, although the structure of the value-printing procedures would be similar. The remaining PostScript code manipulates symbol tables (Section 4.5) or provides machine-dependent code and data (Section 10.2); it is language- and compiler-independent.

About 20% of the debugger source code is devoted to implementing PostScript. This implementation effort is justified because using PostScript confers benefits not conferred by Modula-3, including support for expression evaluation, a readable representation of symbol tables, the ability to change printing procedures at debug time, and fast turnaround for debugging the expression server and compiler. I explain below why an interpreted language is preferable to a simple interpreter for intermediate code or abstract syntax trees, and why PostScript is preferable to other languages.

`ldb`'s implementation of PostScript uses two techniques that may be of interest to other implementors: safe, fast lexical analysis in Modula-3 and compact specifications of operator implementations. These techniques are described below, as are the measures I took to improve the performance of the PostScript interpreter.

`ldb` deviates from the published semantics of PostScript (Adobe 1985, Chapter 3) in three ways. Some features were changed to make the language work better with Modula-3. Some features were omitted because they are expensive to implement and not useful for debugging. Some features were added to support debugging and expression evaluation.

Modula-3 uses the type `TEXT` to represent strings, and these strings are immutable. Modula-3 supports garbage-collected memory, uses exceptions to indicate errors, and uses different types to represent files used for reading and for writing. All of these features are reflected in `ldb`'s PostScript.

`ldb`'s PostScript omits substrings and subarrays. Because strings are immutable, substrings would not be useful; they would be indistinguishable from copies. Subarrays are not worth the implementation cost. Subarrays would be more expensive in Modula-3 than in C; in the safe subset of Modula-3 one can use `SUBARRAY` to construct subarrays of dynamically-allocated arrays, but one cannot create pointers to such subarrays, nor store them in objects. In standard PostScript, both substrings and subarrays are used primarily as part of an idiomatic storage-allocation strategy. The standard string and array operators do not allocate space for their results; a string or array must be provided to hold the results. Because the space provided may be more than is needed, the operators return substrings or subarrays. In `ldb`'s PostScript, similar operators do allocate space for their results, and the interpreter relies on garbage collection to reclaim such space when it is no longer needed.

`ldb`'s PostScript does not use access control. Access control is necessary when the state of the PostScript interpreter resides in a printer, which is shared among several users, but not when the PostScript interpreter is used by only one user, as with `ldb`. `ldb`'s PostScript also omits the imaging types and operators.

A dozen types and 21 operators were added to support debugging, and another 18 operators to support expression evaluation. The new types include abstract memories, locations, symbols, and so on; some are shown in Figure 31. `ldb`'s PostScript also has a character type and literal syntax used to refer to spaces in abstract memories.

9.1 Implementation

`ldb`'s PostScript interpreter has three parts. About 45% of the code—1200 lines of Modula-3—implements the basic abstractions of PostScript. These include the hierarchy of object types recognized by the interpreter, the structure of the interpreter itself, and the code needed to make the interpreter execute PostScript objects. Another 30%—850 lines—implements lexical analysis, and operator implementations account for the remaining 35%—750 lines.

`ldb` implements all PostScript object types except save and fontID. All these types are Modula-3 subtypes of `InterpTypes.Object`. Figure 31 shows part of the subtype hierarchy that describes interpreter types; the unqualified names all appear in the `InterpTypes` interface. Some operations,

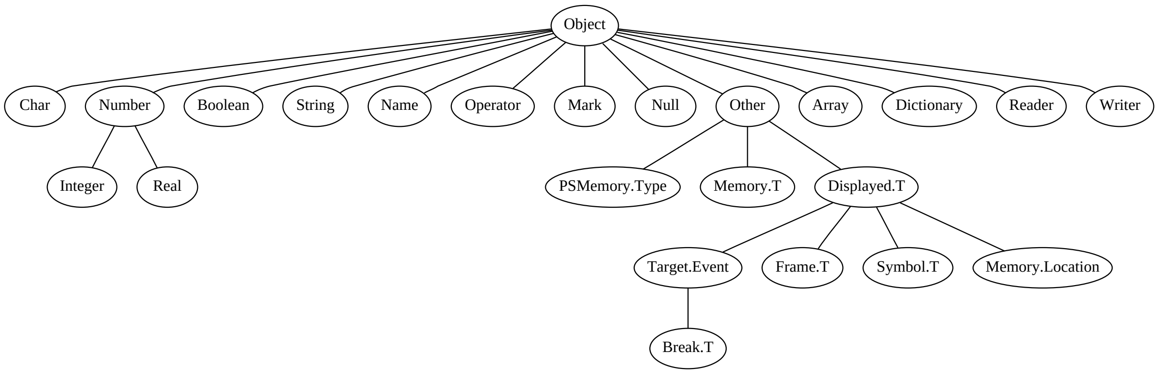


Figure 31: The hierarchy of interpreter types. Not all subtypes of `Other` are shown.

like storing a value in a dictionary, in an array, or on the operand stack, work on the type `Object`, and therefore on all of its subtypes. The interpreter distinguishes among the subtypes of `Object`, and also between `Integer` and `Real`; for example, objects of each of those subtypes are printed differently. The type `Other` provides a means for introducing new types to support debugging. A type declared as a subtype of `Other` inherits the ability to be manipulated by the interpreter; no code has to be written.

Some of the new types are stored in PostScript data structures but never otherwise manipulated by PostScript code. Others, like `Memory.T`, can be manipulated by PostScript operators like `Memory.Fetch`. To implement such an operator requires using the standard Modula-3 mechanism for run-time type checking, `TYPECASE`, but does not require any extra support in the interpreter. The type-checking code can be generated automatically, so adding a new operator requires only a short specification, as shown in Section 9.1.2. These techniques make the PostScript interpreter more useful by minimizing the marginal effort required to add new types and operators.

`ldb` represents a PostScript interpreter as a Modula-3 object having dictionary, operand, and execution stacks, as well as the three distinguished dictionaries `systemdict`, `userdict`, and `errordict`. Because an interpreter is an object, it is possible to have more than one active simultaneously. It is also possible to "clone" an active interpreter, making a new interpreter with the same operand and dictionary stacks, but with an empty execution stack. `ldb` uses this feature to create a new interpreter when an error in a procedure call suspends the current interpreter in the middle of evaluating an expression (Section 7.6). New interpreters are created by cloning from a master interpreter whose `systemdict` contains bindings for the predefined operators.

The rest of the debugger interacts with interpreters by using a few Modula-3 procedures to push and pop the operand stack and to interpret PostScript objects. There are convenience procedures that interpret text and files; each implementation converts a Modula-3 object into the corresponding PostScript object, then interpret the PostScript object. A PostScript object is interpreted by placing it on either the execution stack or the operand stack, depending on its type and its executability attribute. The interpreter then executes the object on the top of the execution stack until the execution stack is emptied. The semantics of executing an individual object depends on its type (Adobe 1985, Section 3.6).

The rest of the basic code implements PostScript dictionaries, including looking up a name in the dictionaries on the dictionary stack.

9.1.1 Safe, fast lexical analysis

In `ldb`, lexical analysis is a bottleneck accounting for more than 30% of the time required to read a large PostScript file. Standard techniques exist for implementing fast lexical analysis, but those techniques use unsafe language features (Waite 1986). Modula-3 offers a buffered-stream abstraction which is safe, but not fast enough. This abstraction has several implementations, which provide input from different sources, including files and strings. I have developed a related abstraction with three useful properties: it supports the implementation of fast lexical analyzers, it isolates the use of unsafe features, and it works with implementations of the existing input-stream abstraction.

Modula-3's "reader," or input stream, abstraction models a sequence of characters with a "current position" pointer at which characters can be read (Nelson 1991, Chapter 6). `ldb`'s `TokenStream.T` adds a "remembered position," which may or may not be set. The characters between the current position and the remembered position, if any, are guaranteed to be buffered, and they can be recoved by a single procedure call. The intended use is for the lexical analyzer to set the remembered position at the beginning of a token and to scan from the current position forward until the end of the token is found. The procedure `TokenStream.Search` provides fast scanning like that shown in the inner loop of the analyzer described by Waite (1986); for adequate speed, its implementation uses unsafe features of Modula-3, like pointer arithmetic.

The `TokenStream` interface improves speed by using other techniques besides fast search. It avoids making a procedure call for every character, eliminates unnecessary allocations, and avoids copying characters except when necessary (Waite 1986). It also reduces synchronization overhead; clients must explicitly acquire a lock before calling procedures in the `TokenStream` interface, instead of having each procedure acquire and release the lock, as with readers. This feature makes `TokenStream` technically unsafe; simultaneous, unsynchronized calls to `TokenStream.Search` could lead to an unchecked run-time error. The risk is acceptable because it improves performance; a lock is acquired and released once per token instead of once per character. Synchronization is still costly; when reading large PostScript files, `ldb` spends 8% of its time acquiring and releasing locks in addition to the 30% spent in lexical analysis. A lock is required by the Modula-3 reader abstraction; eliminating it would mean using system calls directly to read from files and writing special code to read from strings.

9.1.2 Compact specifications for operator implementations

Most of the code that implements a PostScript operator is used to find operands on the stack, to check their types, and to convert from PostScript types to Modula-3 types, for example, from `InterpTypes.Integer` to `INTEGER`. This code is generated automatically from compact specifications much like the notations used in Section 6.2 of the PostScript language reference manual (Adobe 1985). The generated code is about 4.5 times larger than the specifications. Here, for example, are the specifications of the first three operators in the language manual, which pop the stack, exchange the top two elements, and copy the top element.

```
     _ pop -->
x1 x2 exch --> y1 y2 = y1 := x2; y2 := x1;
     x dup --> _ y   = y := x;
```

To the left of the arrow, names are associated with the values of the operands on the stack. To the right, up to the equals sign, names are associated with the positions on the stack of the results; operator implementations can assign to those names. Underscores refer to unnamed operands or results. To the right of the equals sign is the Modula-3 code that implements the operator. The code generated from these specifications checks for stack underflow and overflow, binds names to values or designators, and adjusts the stack pointer after the implementation is executed. No Modula-3 implementation is needed for `pop` because the stack manipulation implements `pop`'s full semantics by itself.

The generator infers the types of operands from their names. `x` and `y`, used above, denote operands of any type. `int` denotes an integer operand, and can be treated as an integer in Modula-3 code, as in the implementation of `index`:

```
int index --> x
  IF int < 0 OR int > sp - 2 THEN Interp.Error(interp, "rangecheck");
  ELSE x := stack[sp-2-int];
  END;
```

If the operand to `index` is not an integer, the generated code indicates a `typecheck` error. As shown, the Modula-3 code can follow on separate lines, with no equals sign. `sp` and `stack` are predefined identifiers that refer to the interpreter's stack pointer and operand stack, respectively. `interp` refers to the interpreter itself.

Some operators are polymorphic. For such operators, one specifies a different implementation for each valid combination of operand types. The generated code places the appropriate implementations at the leaves of a nest of `TYPECASE` statements. The specification for the `add` operator is

```
int1  int2  add --> sum = sum := NewInteger(int1+int2);
int1  real2 add --> sum = sum := NEW(Real, x := FLOAT(int1, LONGREAL) + real2);
real1 int2  add --> sum = sum := NEW(Real, x := real1 + FLOAT(int2, LONGREAL));
real1 real2 add --> sum = sum := NEW(Real, x := real1 + real2);
```

It is not always possible to infer the type of an operand from its name. For example, it is *not* possible to specify the abstract-memory operators as follows:

```
m where val type Memory.Store -->     = ...
m where     type Memory.Fetch --> val = ...
```

The types of these operands must be given explicitly using the notation "name:type". The memory and location types can be specified by name because they are subtypes of `InterpTypes.Other`.

```
m:Memory.T where:Memory.Location val:... type:... Memory.Store -->     = ...
m:Memory.T where:Memory.Location         type:... Memory.Fetch --> val = ...
```

The types `Memory.Value` and `Memory.Type` are not subtypes of `InterpTypes.Other`, so they cannot be referred to by name, e.g., after `val` and `type`. `val` can be handled by using either an integer or a real instead of a `Memory.Value`. A new subtype of `InterpTypes.Other` is needed to represent `type`. The notation "`TYPE Memory.Type type`" creates such a subtype with one field, `type`, containing a `Memory.Type`. This new subtype is private to this module and is referred to as `Type`.

The complete, correct specification for the abstract-memory operators is

```
TYPE Memory.Type type
m:Memory.T where:Memory.Location val:Integer type:Type Memory.Store -->
  Memory.Store(m, where, Memory.Value { integer := TRUE,  n := val }, type);
m:Memory.T where:Memory.Location val:Real    type:Type Memory.Store -->
  Memory.Store(m, where, Memory.Value { integer := FALSE, x := val }, type);
m:Memory.T where:Memory.Location             type:Type Memory.Fetch --> val
  IF Memory.IsInteger[type] THEN
    val := NewInteger(Memory.Fetch(m, where, type).n);
  ELSE
    val := NEW(Real, x := Memory.Fetch(m, where, type).x);
  END;
```

Types are also named explicitly to specify the PostScript operators that create locations, which are based on the procedures described in Section 3.2.

```
offset:Integer space:Char Absolute --> l
  Validate(space);
  l := Location.Absolute(offset, space);
base:Memory.Location offset:Integer Shifted --> l
  l := Location.Shifted(offset, base);
base:Memory.Location offset:Integer space:Char Indirect --> l
  Validate(space);
  l := Location.Indirect(offset, space, base);
int  Immediate --> l
  l := Location.Immediate(Memory.Value{ n := int });
real Immediate --> l
  l := Location.Immediate(Memory.Value{ integer := FALSE, x := real });
```

`Validate(space)` ensures that `space` is a lower-case letter, not just any character. Example uses of two of these operators, `Absolute` and `Shifted`, appear in Chapter 4.

9.1.3 Performance tuning

The original implementation of `ldb`'s PostScript interpreter spent more than 50% of its time garbage collecting and more than 25% in lexical analysis. Several techniques were used to improve its performance. Lexical analysis uses token streams instead of character streams, as described above. Lazy transformation of symbol and type dictionaries defers lexical analysis, reducing the time required to read large tables by 35% (Section 4.5). A unique-string table ensures that no string is allocated more than once, reducing execution time by 25%. The lexical analyzer avoids allocating Modula-3 `TEXT`s as an intermediate stage between tokens and unique strings, reducing execution time by another 17%. Small integers are preallocated; 90–95% of integers used by `ldb`'s PostScript fall in the range $-16 \leq i < 512$, with larger symbol tables hitting slightly higher percentages. Earlier versions of `lcc` packed line and column numbers in a single integer, but `lcc` now uses two separate integers because both integers are likely to be preallocated, whereas the combined integer was not. I reduced garbage-collection time by altering the garbage collector distributed with SRC Modula-3 to handle rapid heap growth. The new version increases the heap in proportion to the amount of live data; previously the increase was of fixed size. The proportion can be adjusted dynamically to trade off speed and space usage (Appel 1989), reducing execution time by 37–50%.

These adjustments combine to make `ldb`'s current interpreter read large symbol tables six times faster than the original. Measurements taken by program-counter sampling indicate that it still spends about 30% of its time in garbage collection and 30% in lexical analysis. It spends about another 8% acquiring and releasing locks, almost all of which should be charged to lexical analysis, since a lock is acquired and released for every token scanned. Almost 20%, or two-thirds of the lexical-analysis time, is spent in the token-stream implementation; almost 10% in `TokenStream.Search` alone. The scanner is called too many times to make elapsed-time measurements practical; attempts to make such measurements increased the running time of the program by 30%.

9.2 Discussion

As noted above, `ldb` uses its PostScript interpreter to do four jobs: reading symbol tables, transferring information from the compiler to the expression-evaluation server, evaluating expressions, and printing values. These jobs could be done by Modula-3 code, but in each case there are advantages to using an interpreted language.

Other debugging formats predefine a fixed set of source-language types, and the debugger implements a printing procedure for each type. The set of types and the code in the debugger have to be extended to support new languages. Users cannot control how values are printed, except perhaps by adjusting a couple of parameters, such as whether integers are printed in decimal or hexadecimal notation. Using an interpreted language means that a compiler writer can create new types and new printing procedures without touching the debugger proper. Users can create new printing procedures at debug time, for example, specifying that a certain pointer should be treated as a pointer to a null-terminated array.[1]

Interpretation provides fast turnaround, which I found useful in writing both the expression-server and value-printing code. The expression-server code converts types and symbols to a representation as a stream of C tokens. I used `ldb`'s PostScript interpreter interactively to debug the code, experimenting until it was producing the stream of tokens I expected. `ldb` uses a prettyprinter to print values (Oppen 1980). Because the prettyprinter uses a backtracking algorithm to produce output, it is difficult to predict the output's appearance. Recompiling, re-linking, and restarting the debugger after each change in the prettyprinting procedures would have been unacceptably slow. Interpretation even makes it possible to change printing procedures while `ldb` is connected to a target, then to re-print an offending value. Such a facility would be helpful to a user trying to write a specialized printing procedure for a complex linked data structure, for example.

The fast turnaround provided by interpretation would be equally helpful if data were displayed using another method, for example by drawing pictures on the screen (Myers 1980). Picture-drawing procedures could be put into PostScript using the compact operator specifications shown above, and the standalone PostScript interpreter could be used interactively to experiment with ways of displaying data. When suitable methods were developed, they could be used to replace the existing printing procedures, providing an alternative way of displaying data.

Evaluating expressions requires either that the debugger be capable of generating machine code in the target address space or that it use some form of interpretation. There are several alternatives to compiling expressions into a general-purpose language; interpreting abstract syntax, as `gdb` does, compiling to a special intermediate code, as `ups` (Russell 1992) does, or interpreting an existing compiler's intermediate code. Interpreting `lcc`'s intermediate code is probably the simplest alternative, using PostScript the next simplest. `lcc`'s dag language has 9 "type suffixes" and 111 type-specific operators, each a variant of one of 36 generic operators. `ldb`'s PostScript has 11 standard data types, plus `Other`, which has more than a dozen subtypes. It has 151 operators, including the special operators that support debugging. The examples in Section 9.1.2 show that the cost of adding types and operators is small; new types are made subtypes of `Other`, and operator implementations require only a few lines of code each. PostScript's control operators and its structured data types, array and dictionary, make it more difficult to implement than the dag language, which has only goto and atomic data types. It also costs more to implement and execute PostScript's lexical analysis than it would cost to implement and execute code to recover `lcc`'s dags from a byte stream, but if not chosen carefully a direct representation of dags could be much harder to debug. The intermediate code used by `ups` compares unfavorably with `lcc`'s intermediate code, having 220 operators and both signed and unsigned variants of the basic types. `ups`'s ANSI C interpreter, including both front and

[1] `ldb`'s current user interface does not support creating new printing procedures; to do so one must use the PostScript interpreter directly and understand `ldb`'s conventions.

back ends, is over 16,000 lines of C (Russell 1992). PostScript is probably simpler to implement than an interpreter for abstract syntax, which would have to handle the control operators and data types of the source language.

No really convincing demonstration of PostScript's utility in expression evaluation can be made until it is used with a different language and compiler; one motivation for using a general-purpose language, and for using PostScript in particular, is to make it easy to work with different languages while minimizing the changes needed in the compilers.

Having symbol-table information in a form easily manipulated by a general-purpose interpreted language has possibilities beyond debugging. `ldb`'s machine-dependent description of the structure of the context used in the debug nub is generated automatically from the nub's symbol table. I have written a set of PostScript printing procedures that, instead of printing values, print Modula-3 declarations of the corresponding types. These procedures could become part of a tool to translate C header files into Modula-3 interfaces. Again, fast turnaround helped me develop these procedures quickly and easily.

Several factors influenced the choice of PostScript. One advantage of using an interpreted language is that it provides a representation that is easy for people to read and for tools to manipulate; bytecode and machine code are inappropriate. Designing a new language is best avoided, so I considered PostScript, Scheme (Clinger and Rees 1991), FORTH (Moore 1974), and Tcl (Ousterhout 1990). The latter two offer too few data types. Although most of the benefits of using PostScript would also be obtained with Scheme, there are a number of reasons to prefer PostScript.

Both the compiler and the expression server have to generate code for the debugger, and PostScript was designed to be generated by other programs (Adobe 1985, p. 2). Postfix notation is easy to generate as well as to interpret, as my experience confirms. For example, the expression-server code that rewrites a dag node into PostScript is less than 100 lines of C, even though there are 111 kinds of dag nodes. A postfix language is also easy to implement; Scheme requires higher-order functions and continuations.

PostScript dictionaries provide a convenient notation for symbol-table entries. Dictionaries are easily extended with machine-dependent data by adding entries; `ldb` does so for the MIPS, adding such information as the frame size and return register. Most of `ldb` ignores this information, but it is used by the machine-dependent stack-walking code.

Every PostScript object has an attribute that tells explicitly whether the object is literal or executable; the distinction need not be inferred from context. Because attempts to execute a literal object put that object on the stack, procedures that are interpreted at most once can be replaced with their results. `ldb` uses this technique when fetching addresses relative to anchor symbols (see Section 4.1).

The PostScript syntax makes it possible to defer not only interpretation but also lexical analysis, by quoting code with parentheses. As described above, and in Section 4.5, this technique reduces by 35% the time required to read a large symbol table. 15% of the savings is in lexical analysis, 20% in interpretation.

Chapter 10

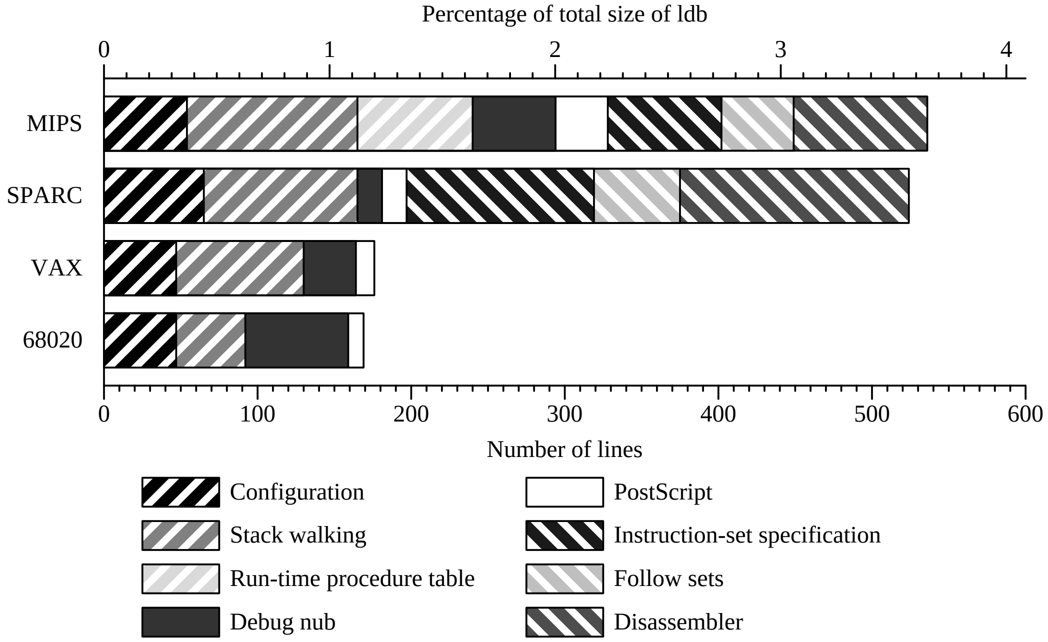
Retargeting `ldb`

Chapters 6, 7, and 8 describe `ldb`'s three major machine-dependent parts: the debug nub, breakpoints, and stack walking. Each part is factored into machine-independent and small machine-dependent subparts. Figure 32 shows how much code is needed to implement each subpart on each target; the four bars represent the four targets. The striped parts of the bars represent Modula-3 code in the debugger. The parts representing configuration and stack walking, on the left, are separated from those representing instruction decoding, on the right, because instruction decoding is implemented on only two targets and the other code is implemented on all four. The dark solid parts of the bars represent C or assembly code in the nub, and the white parts represent PostScript. The table at the bottom of Figure 32 shows the sizes of the major parts, including machine-dependent and machine-independent code.

`ldb` generates instruction decoding and the implementations of PostScript operators from compact specifications; the table in Figure 32 measures the size of the specifications, not of the generated code. The bottom of the table shows the sizes of the generators used to transform instruction specifications and PostScript operator specifications into Modula-3 code, as described in Appendix B and Section 9.1.2. The generators are written in Icon (Griswold and Griswold 1990).

Figure 32 reveals the different costs of debugging on different targets. Stack walking is most expensive on the MIPS. The MIPS calling sequence uses no frame pointer, so it impossible to walk the stack unless the frame size of each procedure is known. Because a program may contain procedures not compiled with `lcc`, that information may not be in the PostScript symbol table. The MIPS uses a 75-line module to get the information from the run-time procedure table, a table placed in target memory by the MIPS linker. The MIPS stack-walking code contains 21 lines that call this module. The run-time procedure table also supplies register-save information for procedures not compiled with `lcc`.

The MIPS nub and PostScript are both larger than those of the SPARC, a comparable machine, largely because the MIPS rules for passing arguments to procedures are more complex, requiring



	MIPS	SPARC	VAX	68020	shared	total
Debugger, less instruction decoding (M3)	240	165	130	92	11,509	12,136
Instruction decoding (M3, patterns)	208	327	—	—	—	535
Debug nub (C, assembly)	54	16	34	67	777	948
PostScript	34	16	12	10	996	1,068
Pattern transformer (Icon)					1,273	1,273
Operator transformer (Icon)					594	594
Total	536	524	176	169	15,149	16,554

Figure 32: The machine-dependent parts of `ldb`.

the use of either integer or floating-point registers depending on the types of the arguments. None of the other calling sequences exhibit comparable complexity. The 68020 has a large nub because it uses assembly language both to build process contexts and to manipulate 80-bit floating-point values; the VAX uses assembly language only to build process contexts.

`ldb` uses instruction-set specifications to implement follow sets and symbolic disassembly for the MIPS and SPARC, as shown by the right halves of the top two bars in Figure 32. Follow sets are used to implement low-level breakpoints and instruction single stepping, as described in Chapter 7. Disassembly is a frill, used only to show machine code to users. There are many instructions, including "synthetic" instructions, so the disassembly code is long, but it is not difficult to write because it parallels the instruction-set specification. The instruction-set specification of the SPARC is more complex than that of the MIPS, reflecting the greater complexity of the SPARC instruction encoding; the SPARC has six instruction formats, the MIPS three.

The effort required to retarget `ldb` is minimal. Ports to the 68020 and VAX architectures each took less than a week. A colleague independently retargeted a much earlier version of the SPARC stack-walking and configuration code; he spent about two weeks working half-time.

Detailed examples of some of `ldb`'s machine-dependent code appear in Chapters 6 and 8 and in Appendix B. This chapter provides a less detailed, but more comprehensive overview of the work needed to move each part of the system to a new machine. The nub, the PostScript, the debugger proper, and the compiler are discussed separately.

10.1 The nub

Although the nub does not require many lines of machine-dependent code, it is full of machine-dependent details. Most require only one or two lines of code. The details include defining signal handlers of the proper type, getting the nub to pause on startup, possibly invalidating the instruction cache, possibly supporting special data formats, and coping with machine-dependent idiosyncrasies. To isolate such details, the nub uses the C preprocessor; machine-independent code invokes machine-dependent macros. Using macros instead of conditional compilation does not always make the nub smaller, but it makes the code easier to understand. Control flow is not obscured by conditional-compilation directives, and it is usually possible to assign a machine-independent meaning, e.g., "flush the instruction cache," to a macro whose definition is machine-dependent.

Most of the effort of retargeting the nub is devoted not to the details but to capturing the process context and supporting procedure call. At minimum, the process context must include the values of registers and of the program counter. Changes to the values stored in the context must affect the actual machine state when execution is resumed. On the MIPS, the `struct sigcontext` has these properties, and it serves as the process context. `ldb` uses the `struct sigcontext` on the SPARC too, even though SunOS does not supply the values of all of the registers, because it does supply enough to build a stack frame. It does not supply the register used by optimized leaf routines to hold the return address, so when `ldb` encounters such a routine the user cannot examine the shared stack frame from the perspective of the routine's caller. On the 68020 and VAX, the `struct sigcontext` does not supply enough information, and the registers have to be acquired using assembly code, an example of which is shown in Section 6.4.2. This code, plus the definition of the process context, is 20 lines of C and assembly code on the 68020 and 16 lines on the VAX.

Section 6.4.1 explains how the nub implements procedure calls using a machine-independent template that invokes machine-dependent macros to handle argument passing. Writing the macros requires a thorough understanding of the argument-passing rules, but the macros themselves are not large: 3 lines each on 68020 and VAX, 8 on the SPARC, and 19 on the MIPS.

All the targets support three sizes of integers, as well as `float` and `double`, but only the 68020 supports 80-bit (extended) floating-point values. To do so, it defines the macro `HASF80`, plus the two functions `DebugNub_fetchF80` and `DebugNub_storeF80`, which the nub calls to fetch and store 80-bit floating-point values. If `HASF80` is undefined, attempts to fetch and store 80-bit floating-point values fail. These functions must be written in assembly language because C cannot manipulate 80-bit floating-point values; their implementations take 27 lines. They manipulate 80-bit values only in memory; their arguments and results are 64-bit `double`s. The extra bits are discarded. Similar losses of precision may occur in the target program itself because floating-point values occupy 80 bits when stored in registers, but 64 or 32 bits when stored in memory. The debugger fetches and stores 80-bit floating-point values only when the 68020's floating-point registers are saved on the stack or in the process context. If the extra bits were needed for an application, different machine-dependent code would have to written, and `ldb`'s debug-nub protocol would have to be changed to handle floating-point values larger than two long words. If future targets required the introduction of more data types not accessible from C, they could be handled similarly.

In addition to the process context, procedure call, and extra data formats, the nub must handle the idiosyncrasies of individual machines. The MIPS is the most idiosyncratic, needing about 20 lines of C and assembly code to handle problems that have no analogs on other machines. For example, on a big-endian MIPS, doubleword floating-point values are stored with the most significant word first, except that when the kernel saves floating-point registers in a `struct sigcontext`, it stores the least significant word first. To recover such registers correctly, the nub must save the location of the floating-point registers within the process context, and its code for doubleword fetches and stores of saved floating-point registers must swap the words when fetching from or storing to those locations. The code uses the machine-dependent macro `MIPS_SWAP`, which does the swap on the MIPS and is a no-op on the other targets.

There is one more retargeting task associated with the nub. To give the nub initial control, the system-dependent startup code must be modified to call the nub instead of `main`. It is most easily modified by editing the object code, replacing `main` with `mAiN`. This method is dirty, but it is easier than modifying the assembly code to call `DebugNub_Init`, even on targets for which assembly code is available.

10.2 PostScript

`ldb` uses machine-dependent PostScript to address automatic variables, to describe a target's registers, and to help support procedure calls. `lcc` assigns an offset to each automatic variable. On most targets, all such variables are addressed indirect with respect to the frame pointer. On the VAX, those with positive offsets are arguments, indirect with respect to the argument pointer; those with negative offsets are locals, indirect with respect to the frame pointer. The VAX uses a separate argument pointer because arguments need not be passed on the stack. `lcc` emits PostScript code that computes the location of an automatic variable by applying the `Local` procedure to its offset.

```
/Local { dup 0 ge { ap } { fp } ifelse exch 'd' Indirect } def   % VAX
/Local { fp exch 'd' Indirect } def                              % others
```

`fp`, the location of the frame pointer, is defined differently on each machine. `ap`, the location of the argument pointer, is defined only on the VAX.

`ldb` defines names like `Local` and `fp` by using one PostScript dictionary for each target architecture. The dictionary associates machine-dependent names with their definitions. An architecture is selected by the PostScript procedure `Architecture.Set`, a call to which appears in each of the symbol tables emitted by the compiler. When first called, `Architecture.Set` places the appropriate machine-dependent PostScript dictionary on the dictionary stack, making the definitions visible. Unless the architecture changes, `Architecture.Set` does nothing on subsequent calls.

`ldb` uses machine-dependent PostScript code to print the register values associated with a stack frame, because it is more easily done in PostScript than in Modula-3. Each machine-dependent dictionary defines names, locations, and types for the registers in the machine. Register names beginning with `$` can be used in expressions. For example, one can discover the size of `fib`'s frame by subtracting the frame pointer from the stack pointer. On the SPARC, it is 24 words—hex 60 bytes:

```
ldb fib (stopped) > p $fp - $sp
unsigned int  $fp - $sp = 0x60
```

The MIPS nub, as mentioned above, must know the types of the first two arguments of a procedure in order to put those arguments into the proper registers. The MIPS dictionary redefines the PostScript procedures that store arguments in the argument-build area. The new versions, in addition to storing the arguments, encode the types of the first two arguments in a word of machine-dependent data. That word is eventually passed to the `PREPARE_CALL` macro described in Section 6.4.1.

10.3 Debugger code: breakpoints and stack walking

Within the debugger proper, `ldb` uses an object of type `Architecture.T` to hold all the machine-dependent code and data for a particular target architecture. This code and data describe breakpoints, instructions, and stack walking. `Architecture.T` has seven fields. `name` is the name of the architecture, which `ldb` associates with the particular object of type `Architecture.T`. When an architecture name is used in PostScript, `ldb` uses the machine-dependent code and data in the associated object. `break` is a low-level breakpoint implementation, and `trapsig`, which is usually defaulted, is a specification used to recognize breakpoint-trap events. `pc` is an object that provides access to the program counter in a process context. `follow` is a procedure that computes

```
1  pc := NEW(MipsGCF.Integer,
2            field := Context.pc,   type := Memory.Type.I32);
3  hp := NEW(MipsGCF.Integer,
4            field := Context.regs, type := Memory.Type.I32, offset := 29*4);
5  Architecture.Add(NEW(Architecture.T,
6        name        := "mips",
7        break       := FollowBreakpoint.Implementation(MipsDecoding.Follow,
8                                  Trap.Specification{Memory.Type.I32,16_0d}),
9        pc          := pc,
10       follow      := MipsDecoding.Follow,
11       disassemble := MipsDecoding.Disassemble,
12       topFrame    := MipsGCF.New));
```

Figure 33: The MIPS configuration module.

the follow set of an instruction (Section 7.5); `disassemble` is a procedure that computes a textual representation of a machine instruction. `topFrame` is a procedure that, given a process context, returns an object representing the frame on top of that process's stack. On each architecture, the `Architecture.T` is created by the configuration module, as shown in Figure 33.

A low-level breakpoint implementation encapsulates one of the two strategies `ldb` uses to implement low-level breakpoints, hiding both the strategy and the machine-dependent data or code needed to implement it (Section 7.5). The breakpoint implementation is an object with a `plant` method, which user-level breakpoints use to create low-level breakpoints, plus a trap specification (Section 7.4). Two different `plant` methods are available; to use one the configuration code must supply either a specification describing no-op instructions or a procedure that computes follow sets. For the MIPS, `ldb` uses the follow-set procedure `MipsDecoding.Follow`, as shown on line 7 of Figure 33.

`follow` and `disassemble` rely on an instruction-set specification like that shown in Appendix B. The sizes of the three parts are shown on the right of Figure 32. `follow`, of course, can be used to specify follow-set breakpoints. `ldb` also uses `follow` to implement its `stepi` command, and it uses both procedures to implement its `assembly` command, shown in Chapter 2. The default versions of `follow` and `disassemble`, which are used on the 68020 and VAX, always raise exceptions, making `ldb` print an error message when a user attempts a command that uses them.

The `topFrame` procedure hides the details of getting information from the process context, building abstract memories, and walking the stack. It creates the top frame on the stack, with an abstract memory. This frame has machine-dependent methods used to find other frames on the stack and to build abstract memories containing saved registers. The `topFrame` procedure and some of the stack walking are implemented by generic code, as described in Section 8.2. The parameter to this generic code is a machine-dependent configuration interface. Figure 34 shows the complete configuration

```
1  TYPE
2    FrameType = MipsFrame.T;
3    Context = {onstack, mask, pc, regs, mdlo, mdhi, ownedfp,
4               fpregs, fpc_csr, fpc_eir, cause, badvaddr, badpaddr};
5  CONST
6    Offsets = ARRAY Context OF INTEGER
7                 {0, 4, 8, 12, 140, 144, 148, 152, 280, 284, 288, 292, 296};
8  VAR pc, hp: CommonFrame.Integer;
9  CONST
10   AliasSpaces = ARRAY OF ['a'..'z'] { 'r', 'f', 'x' };
11   AliasSizes  = ARRAY OF INTEGER    {  32,  32,   2  };
12   RegSpec     = RegisterMemory.Bits32;
13   RegSpaces   = SET   OF ['a'..'z'] { 'r',       'x' };
14   RegBases    = ARRAY OF Context    { Context.regs, Context.fpregs, Context.pc };
15   RegShifts   = ARRAY OF INTEGER    {  4,    4,   0  };
16   EeSizes     = ARRAY OF INTEGER    {  4,    4,   0  };
17   Preserved   = ARRAY OF RegSet.T   { RegSet.T { 16..23, 26..28, 30..31 },
18                                       RegSet.T { 20..31 }, RegSet.T {} };
```

Figure 34: The MIPS configuration interface.

interface for the MIPS, the pieces of which are explained in Sections 3.2.2 and 8.2. Line 2 identifies the subtype of frame used for the MIPS; the type `MipsFrame.T` defines machine-dependent implementations of `callerFrame`, `abstractMemory`, and the other machine-dependent methods described in Section 8.2. Lines 3–7 describe the structure of the process context; line 8 declares variables used to fetch program counter and heavy pointer from the context. Lines 10–13 describe the structure of MIPS abstract memories. Lines 14–15 tell where to find registers in the process context, and lines 16–18 tell how to restore preserved registers during stack walking.

The module in Figure 33 implements the MIPS configuration interface. `MipsGCF` is the instantiation of the generic stack-walking code, given the MIPS configuration interface as a parameter. Its `Integer` type provides access to fields of the process context; its `New` procedure uses the process context to create the topmost stack frame. `MipsDecoding` implements follow-set computation and symbolic disassembly, as described in Appendix B. Lines 1–4 of Figure 33 define the locations of the program counter and heavy pointer within a process context, and lines 5–12 create and install the `Architecture.T` that describes the MIPS. On lines 7 and 8, the breakpoint implementation is created by machine-independent code that requires only the specification of traps and the procedure that computes follow sets (Section 7.5).

In Figure 32, "configuration" labels the parts of the bars that measure the sizes of configuration modules and interfaces like those in Figures 33 and 34, plus the few lines needed to instantiate the generic stack-walking code on the different targets.

10.4 The compiler

`lcc` must be retargeted for each new machine. Retargeting the back end is beyond the scope of this thesis (Fraser and Hanson 1991a). The compiler's symbol-table code, which emits PostScript symbol tables and puts labels and anchor symbols into the compiler's assembly-language output, is retargeted by defining three machine-dependent macros. The compiler puts the name of the target architecture in the symbol table. `lcc` uses small integers to represent register sets, so a string is used to associate register-set numbers with the lower-case letters that name spaces in abstract memories. The letters used are determined by the array `AliasSpaces` in the configuration interface (Figure 34); register set 0 is associated with the first element of `AliasSpaces`, and so on. Finally, the compiler emits a label, and possibly other assembly code, at each stopping point.

The macros for the MIPS are

```
#define ARCHITECTURE "mips"
#define REGSPACES "rf"
#define STOP ".set noreorder\nL.%s:\n.set reorder\n"
```

`REGSPACES` indicates that register set 0 corresponds to space `r`, the integer registers, and set 1 to space `f`, the floating-point registers. The `.set` commands prevent the assembler's instruction scheduler from moving instructions across stopping points. The 68020 uses no-op breakpoints and a different collection of register sets. Register sets 0, 1, and 2 correspond to spaces `a`, `r`, and `f`—the address, data, and floating-point registers:

```
#define ARCHITECTURE "mc68"
#define REGSPACES "arf"
#define STOP "L%s: nop\n"
```

10.5 Discussion

`ldb`'s machine-dependent code covers three of the axes mentioned in Chapter 1: instruction set, calling sequence, and run-time support.

Describing an instruction set makes it possible to compute the follow set of any instruction, which makes it possible to implement breakpoints that can be planted anywhere except in a branch-delay slot. The pattern language described in Appendix B describes the MIPS and SPARC architectures concisely. The language is less well suited to describing the VAX and 68020, because it does not provide means of describing instructions that vary in size or of writing patterns that match sequences, e.g., an opcode followed by several operands. Although instruction decoding and follow-set computation could be implemented entirely by hand, such an implementation is tedious, error-prone, and possibly less efficient than the machine-generated one. It would be better to extend the pattern language to handle CISC instruction sets.

Instruction-set decoding and symbolic disassembly account for a large fraction of the number of lines of machine-dependent code: 40% on the MIPS, 60% on the SPARC (Figure 32). The intellectual effort required is not proportional to the size of the code. The instruction-set descriptions are much like the descriptions that appear in the manuals, and the symbolic-disassembly code requires only a print statement for each group of instructions having similar syntax. The real effort was expended on the design of the pattern language, which can be re-used for different tasks and different machines.

Less general breakpoints, which can be planted only at no-op instructions, can be implemented without describing instruction sets. `ldb` uses such breakpoints on the VAX and 68020. Their utility depends on cooperation from the compiler, which, on those targets, inserts a no-op at each stopping point. No-op breakpoints can be implemented using just a few lines of machine-dependent data, which describes no-op and trap instructions. Even when the more general follow-set breakpoints are necessary, no-op breakpoints provide a useful implementation path. They can be used in a first stage of retargeting, in which the debugger is made functional with the restricted breakpoints. Then, in a second stage, the general breakpoints are implemented.

Calling sequences affect retargeting in two areas: stack walking and procedure-call support. Because of `ldb`'s choice of abstractions, the problem of stack walking can be stated simply: identify frames by return address and heavy pointer, and build an abstract memory representing each frame's register contents. Because all the targets use the abstract-memory structure shown in Figure 6 on page 24, and because they use generic code to restore registers, there is little retargeting effort associated with abstract memories. As shown in Figure 34, one must specify the spaces in the abstract memory, the treatment of saved registers, the locations of registers in a process context, and the identities of the preserved registers. The choice of spaces in the abstract memory should correspond to the compiler's choice of register sets, which should correspond to the hardware. On some targets, it may be useful to add an "extra" space to hold values not accessible from the normal registers, such as the MIPS program counter and virtual frame pointer.

By contrast, writing the code that identifies stack frames is one of the difficult parts of retargeting. It is difficult because calling-sequence documents are either not available at all or not written with stack walking in mind. The calling sequence had to be inferred from the description of the return instruction on the VAX and from the assembly code generated by the compiler on the 68020. The MIPS and SPARC architecture manuals do specify calling sequences, but only in enough detail to write conforming procedures. The documents do not specify the structures of all the frames that might be found on a stack; for example, they omit descriptions of signal-handler frames and of the bottom-most frame on the stack. Coping with such special frames can represent significant retargeting effort (Linton 1990). `ldb` does not attempt to reverse engineer undocumented stack frames; for example, on the MIPS, it mistakenly identifies signal-handler frames as the bottom of the stack.

Calling sequences can be specified to make stack-walking easy (Digital 1992). Such specifications have two characteristics: they identify every kind of frame that might appear on the stack, and they place restrictions on entry and exit sequences so the intermediate states discussed in Section 8.4 can easily be identified. Such restrictions do not degrade performance; a typical restriction is that all registers to be saved by the callee must be saved before any such register is written.

Implementors of debuggers are not the only ones to benefit from careful specification of calling sequences. Stack walking can be used to implement exception handling in languages like C++ (Ellis and Stroustrup 1990), Ada (US DoD 1983) and Modula-3, which permit any active procedure to handle an exception. The method generalizes the handler-table method described by Liskov and Snyder (1979); when an exception is raised, the run-time system walks the stack, looking for a return address within the scope of an appropriate handler. This method requires no execution-time overhead in the normal case.

Despite its small size, the nub is the most difficult part of `ldb` to retarget. Acquiring the process context may require using an unfamiliar assembly language, and supporting procedure call requires detailed knowledge of the calling sequence, but undocumented surprises, like the need to swap the words of doubleword floating-point registers on the big-endian MIPS, are the real problem. Keeping the nub small makes surprises less likely.

The nub's support for procedure call assumes that the positions of arguments, not their types, determine in what registers, if any, arguments are passed. The MIPS violates this assumption when the first or first two arguments are of floating-point types, in which case register assignment is determined by both type and position. The registers in which other arguments are passed are not affected. This violation accounts for the extra retargeting effort required to support procedure call on the MIPS. If another machine had a calling sequence in which the assignment of arguments to registers depended on the positions and types of all arguments, retargeting the nub would be even more complex.

Some debugger code depends on a data structure defined in the nub; it is necessary to specify the structure of the process context so that the program counter, heavy pointer, and general-purpose registers can be recovered from that context. As shown in Figure 34, the specifications are just a few lines. The specification of the context is generated automatically from the nub's symbol table. An alternative that reduces the debugger's dependence on nub data structures is to provide an abstraction giving access to registers in a way that is independent of those data structures (Adams and Muchnick 1986). Such an extra layer of abstraction does not make sense for `ldb`, because the nub has no purpose other than to support debugging, and therefore there is no reason to make the debugger independent of its data structures.

There is a trade-off between retargeting effort and functionality. With more retargeting effort, `ldb` can provide some support for machine-level debugging. The code needed to implement general breakpoints also provides instruction-level single stepping. Once that is implemented, the extra effort needed to provide symbolic disassembly is low. Supporting other compilers also requires more retargeting effort. If programs contained only procedures compiled with `lcc`, the code that reads the MIPS run-time procedure table could be eliminated, as could the code that recovers register-save information on the VAX and SPARC. Because it is often difficult to recompile library code, the extra retargeting effort required to work with more than one compiler is justified—and necessary.

Unlike other debuggers, `ldb` does not have to be retargeted to handle a variety of executable-file and debugging-symbol formats. The implementors of Xerox PARC's Cirio debugger and those of DEC's ladebug debugger have both indicated that handling different formats required substantial implementation effort. Both `ups` and `gdb` devote roughly 10% of their source code to such formats. Over 7,000 lines of `ups`'s source is conditionally compiled code that reads symbol tables. `gdb` devotes 10,000 lines to five debugging-symbol formats, COFF, SunOS, `a.out`, `b.out`, and ECOFF, and another 4,000 lines to the corresponding executable-file formats. `ldb`'s use of a single, machine-independent symbol-table format eliminates this kind of retargeting effort, making the symbol table the cheapest and fastest part of `ldb` to retarget.

The cheapest, fastest, and most reliable parts of a computer system are those that aren't there.

—Gordon Bell (Bentley 1988, page 62)

Chapter 11

Evaluation

There is no single insight that makes it easy to implement a debugger; success is determined by the cumulative effect of many engineering decisions. This chapter reviews those decisions to judge which work and which do not. The most unusual decision is basing much of the debugger on PostScript, but there are several others worth exploring. Retargeting cost is affected by decisions not just about how to implement, but also about what to implement; among the thorniest are decisions about compatibility.

Engineering is not an end in itself; its purpose is to produce artifacts. One way to judge how well `ldb` works as an artifact is to compare it with `dbx` and `gdb`, its nearest competitors.

A research project is an experiment. Like any other experiment, `ldb` uncovers problems that other experiments might address, some of which are discussed below.

Lessons learned in building `ldb` can be applied elsewhere. This chapter suggests how the techniques used in `ldb` might affect other debuggers and related tools, and it closes with a short list of imperatives for implementors.

11.1 PostScript

Handling a variety of machine-dependent formats for debugging information imposes substantial retargeting costs on debuggers like `dbx` and `gdb`. Most of these formats are variations on `dbx`'s original format. One reason there are so many variations is that extensions are required to support new programming languages because the format must name all the type constructors of all the languages the debugger supports. For each target, the debugger must associate the names with machine-dependent information about the representations of those types.

By using PostScript, `ldb` eliminates the retargeting costs associated with debugging information. The most obvious benefit of using PostScript to represent debugging information is that it is a single, machine-independent format. PostScript also promotes retargetability in a less obvious way.

By providing extensible representations of symbols and types, and by including procedures as well as data, it eliminates the need to identify types by name. It isolates `ldb` from machine-dependent information needed only to evaluate expressions or to print values, like the compiler's representation of symbols and types and the machine representation of source-level data. This information is emitted by the compiler, not stored in the debugger. For example, `ldb` need not know the sizes and alignments of the basic types on each target. `ldb` is simplified because it uses this information only indirectly by interpreting PostScript. The compiler and expression-evaluation server are simplified because they can use a mutually convenient representation of the information. Neither the format nor the debugger need be extended to accommodate a new programming language; PostScript procedures and dictionaries provide the necessary extensibility.

PostScript is easy to read, understand, and change; both data and code can be manipulated with a standard text editor. These properties were invaluable during `ldb`'s development when the representation of symbols and types changed frequently; for example, a proposed change could be made with an editor and tested before the compiler was changed to emit the new representation. They were similarly useful during the development of the expression-evaluation server. The PostScript that the server emits is like an assembly language (see Section 5.3.3, especially Figures 19 and 20), and debugging the server is more like debugging a compiler that emits assembly language than like debugging a compiler that emits object code. The server can print its output on the screen as well as send it to `ldb`, so the server's output can be debugged by picking up pieces of PostScript with the mouse and interpreting them.

PostScript symbol tables impose a substantial performance cost on `ldb`. Aside from the extensibility of its dictionaries, PostScript is a poor data format. It must be executed sequentially, it requires lexical analysis, and it is voluminous. Compression would reduce volume, and there is a standard binary encoding that reduces volume and eliminates lexical analysis (Adobe 1990, Section 3.12), but neither technique removes the limitation of sequential access. Performance would be most improved by representing debugging information in a compactly encoded form designed for incremental reading and random access. Using PostScript values as the atomic elements of such a data structure would retain the advantages of using PostScript except for readability. As proposed in Chapter 4, it might be possible to use the current, readable, slow representation while `ldb` is being moved to a new compiler or machine and to switch to a fast representation after retargeting is complete.

`ldb`'s implementation of PostScript could be used to provide programmability for other applications. Its attraction is the low cost of adding types and operators, which is typically one or two lines of overhead per operator (Chapter 9). The cost is low because type-checking code is generated from compact specifications when the interpreter is compiled.

PostScript could be used as the basis for a loosely integrated environment in which several tools, not just a compiler and a debugger, communicate by sending PostScript to be evaluated remotely.

It might be useful for a debugger to send an editor a command that displays a particular source location, for example. A similar role has been proposed for Tcl (Ousterhout 1990), but Tcl has only one data type, the string. Tcl has awkward quoting conventions because of its dual roles as programming language and interactive command language. PostScript has a richer set of types, and its postfix syntax is easy for programs to generate. Tcl does have the advantage that its syntax is shell-like, making it a comfortable interactive command language.

11.2 Strengths

Much of `ldb`'s engineering effort is devoted to pushing machine dependence into small corners of the system. When the a machine-dependent subproblem is sufficiently small and isolated, standard techniques can be used to solve it. Printing values, expression evaluation, no-op breakpoints, and some of stack walking use machine-independent manipulation of machine-dependent data. Follow-set breakpoints use generation of machine-dependent code from compact specifications. The debug nub and some of stack walking use small sets of machine-dependent procedures, macros, or methods.

Abstract memories provide a simple, universal model of target memory and registers, usable anywhere in the debugger. Abstract memories provide a means for uniform treatment of target variables, whether located in registers or in memory. Associating an abstract memory with each procedure makes it possible to use simple PostScript code emitted by the compiler because the abstract memory represents the state of the machine itself when that particular procedure was active. With modest machine-dependent support, abstract memories could describe core files as well as running processes.

Symbolic disassembly and breakpoints are unrelated problems, but they share the subproblem of computing follow sets: finding the inline successor of an instruction and finding the target of a branch. Follow sets are also used to implement instruction-level single stepping. This subproblem is the only machine-dependent part of `ldb`'s follow-set breakpoint implementation. Symbolic disassembly has the second machine-dependent subproblem of computing textual representations of instructions. `ldb`'s solutions to both problems rely on compact specifications of instruction sets, like the MIPS specification in Appendix B. Once follow-set breakpoints are implemented, symbolic disassembly and instruction-level single stepping can be added at low and no cost, respectively.

`ldb` gets leverage from the compiler. Re-using `lcc` as an expression-evaluation server saves substantial implementation effort in the debugger and provides greater confidence that the debugger and compiler implement the same language. To support such re-use, the compiler must support the re-creation of symbols, and it must be modified to translate expressions into PostScript. In `lcc`, the code that re-creates symbols is similar to the code that handles `extern` symbols. In other languages with separate compilation, such code might also be analogous to the code that imports symbols from other compilation units. Many compilers already generate different machine or assembly codes from a single intermediate form; adding PostScript should not pose problems. Even single-target compilers should be able to generate PostScript from the trees, dags, or quadruples often used as intermediate forms.

`ldb` also gets leverage from the compiler in the debug nub's support for procedure call. The debugger does not reproduce the compiler's actions in setting up arguments and calling a procedure; the compiler generates that code and links it into the target address space in the nub. The debugger's job is reduced to accumulating arguments into an argument-build area, which is done by the PostScript that the expression server emits. Retargeting is simplified because several of the machine-dependent aspects of procedure call are handled automatically by compiling the nub with the right machine-dependent compiler; the other aspects, which mostly concern argument placement, can be handled by a few lines of machine-dependent code.

`ldb`'s ability to change architectures dynamically has unforeseen benefits. It simplifies configuration because the target architecture does not have to be chosen when `ldb` is built and because there is one instance of `ldb` for each host, not one for each host-target pair. Because `ldb` does cross-architecture debugging, it is not even necessary to build a debugger for each host. In tools in which architectures are chosen at build time, it is possible to make small machine-dependent changes by using conditional compilation. Unless great discipline is exercised, such changes proliferate, making it difficult to identify the machine-dependent parts of the program. Changing architectures dynamically means that code and data for all architectures must be present at once, imposing a discipline that makes it easier to identify and isolate machine-dependent code.

11.3 Weaknesses

I made several mistakes in `ldb`'s design. I had planned to replace `ldb`'s user interface with a graphical user interface, a programmable command language, or a combination. To simplify this replacement, I limited the user-interface's access to the rest of the debugger. I mistakenly did so by gathering everything the user interface needed to know about the rest of the debugger into a single Modula-3 interface, `Target`. I compounded the mistake by hiding as much information as possible, revealing only such information as the user interface needed. The results were a large, unwieldy `Target`, procedures that did nothing but call procedures in other interfaces, and unnecessary subtypes. The subtypes led to extra layers in many abstractions, for example, the top layer of the stack-frame abstraction shown in Figure 27 on page 112. I made some improvement by changing the user interface to depend on several debugger interfaces, not just `Target`, which helped eliminate some redundant code. Similar problems remain, however, e.g., in the code that the user-interface calls to plant breakpoints. I have not identified a structure that would enable `ldb` to change user interfaces smoothly. It is not clear, for example, whether hiding information from the user interface serves this or any other purpose.

`ldb` provides useful information for programs containing procedures not compiled with `lcc`. On the MIPS and SPARC it provides instruction single stepping and symbolic disassembly, and it can be useful even when no procedures outside the nub are compiled with `lcc`. In other words, `ldb` can be used as a machine-level debugger. It was a mistake not to have planned for machine-level debugging, and that mistake has affected both the user interface and the entire design. The user interface has poor support for examining machine-level data; the best a user can do is use C, e.g., by dereferencing integers cast to pointers. The design does not separate machine-level debugging from source-level debugging, making it impossible to estimate the cost of source-level debugging or to try other approaches. A redesign would separate source and machine levels and make it possible to build a version of `ldb` with no source-level features.

The results in Chapters 4 and 5 show that an existing compiler can support `ldb` with only modest changes and that an expert in the compiler is not needed. One flaw in `lcc`'s support, in the code that emits the PostScript symbol tables, is that the parts that could be useful in another compiler are not separated from the parts that are useful only within `lcc`. Another flaw is that generating the PostScript slows down the compiler by 40–200%. Using the compiler's existing interface for symbol-table generation is of mixed benefit. Fewer changes are made to the compiler, but it is difficult to identify alternative interfaces within the compiler that might simplify its implementation or offer better performance.

`ldb`'s stack-walking code assumes that the process context is located somewhere in the target data space. This design complicates access to the context, because its offset within the data space must be passed to some stack-walking methods. A better design would be to allocate a separate abstract-memory space to hold the process context. To work with the existing nub, this design would require a new kind of abstract memory to map this new space onto part of the data space. If the nub support were moved from user space into the kernel, the process context might be better kept in kernel data structures, not user space. The proposed design would easily adapt to that change; the current design would not.

11.4 Comparison with dbx and gdb

`ldb` is a practical debugger and is competitive with `dbx` and `gdb`. It is most often used for debugging C programs of about 10,000 lines, including `lcc`, the expression server, and an experimental linker (Fernandez, Fraser, and Hanson 1992). It is also used to debug itself. The features that its users like best are the abilities to connect to a running process and to continue debugging that process even if the debugger fails (Chapter 6). These features are not new with `ldb`, but `dbx` and `gdb` do not provide them.

`ldb` omits many features found in `dbx` or `gdb`, including printing source code, debugging core files, temporary breakpoints, stepping into function calls, support for signals, and others. Although it is easy to argue in favor of many of these features, `ldb`'s primary objective is to provide basic debugging in a retargetable way. The feature that its users miss most is source-code display, although other debuggers have been successful without implementing such display (Cargill 1983).

`ldb`'s overall performance is worse than that of `dbx` or `gdb`, primarily because reading PostScript symbol tables is slow. As described in Section 4.7, `ldb` takes 5 to 15 seconds to start on programs of 10,000 to 100,000 lines. During debugging, delays of 5 to 10 seconds occur when new compilation units of 1,000 lines or so are touched. I logged the 20 slowest commands in 448 debugging sessions. In only 23 of the sessions were there more than 3 commands that took longer than 5 seconds to execute. Averaging over all sessions, 70% of commands took less than a tenth of a second. Averaging over long sessions, in which more than 20 commands are issued, 80% of commands took less than a tenth of a second.

`ldb` interacts with target processes at about the same speed as `dbx` and `gdb`. As described in Section 7.7, `ldb`'s breakpoint implementation is faster than `gdb`'s when both debug a child process. When `ldb` debugs a peer process on the same machine, performance is about the same as `gdb`'s debugging a child process. When `ldb` debugs a process on a different machine, it is less than twice as slow as `gdb` debugging a child process. `dbx`'s breakpoint performance is difficult to measure directly, but indirect measurements suggest that it is about the same as `gdb`'s. When implementing conditional breakpoints, `ldb` loses its relative advantage because the expression server makes no effort to minimize interactions with the debug nub. `ldb`'s conditional breakpoints are about 50% slower than `dbx`'s or `gdb`'s when all three debug child processes.

`ldb` is more reliable than `dbx` or `gdb`. `dbx` resolves names using an elaborate lookup scheme that involves the static scope, the call stack, and the local variables of all compilation units (Linton 1990, page 213). The purpose of the scheme is to make it possible to examine variables that might not ordinarily be visible, but the version supplied with Ultrix has errors, the most common of which is to fail to identify active local variables. When a user asks `dbx` to plant a breakpoint at the beginning of a procedure, `dbx` sometimes plants it at another location that cannot be determined. When this error occurs, the user must run `nm` to find the address of the procedure and use `dbx`'s `stopi` command to plant the breakpoint at that address. If a user calls a procedure from `ldb` and the procedure faults, `ldb` can debug the fault and then recover; `dbx` cannot recover and `gdb` cannot debug the fault. Finally, an `ldb` user can recover from debugger errors by starting a new debugger.

`ldb`'s greatest advantages lie in its implementation, which provides a small, retargetable basis for debugging. Retargetability implies not just that a tool works on more than one target, but that the effort needed to retarget the tool is much less than the effort needed to rewrite it. A retargetable implementation separates machine-dependent and machine-independent parts. Identifying the machine-dependent parts makes it possible to apply other techniques that reduce retargeting effort, like generating code from compact specifications. Separating such parts makes it unlikely that code for a new target will introduce bugs affecting other targets. `gdb`'s machine-dependent parts are impossible to identify precisely, but to support 20 targets it uses machine-dependent modules totalling 57,000 lines of C, an average of 2,850 lines per target. `ldb` uses 200-300 lines of specifications and Modula-3 to implement instruction decoding and another 200-350 lines of Modula-3, PostScript, and C to implement the other machine-dependent functions; it uses 535 lines total for the MIPS and 524 for the SPARC, the two machines for which both parts are implemented.

11.5 Compatibility and other retargeting costs

Any tool is more useful if it is compatible with existing systems, but compatibility can introduce unnecessary retargeting effort. Another drawback of compatibility is that one implements another's solution instead of studying the problem to be solved. `ldb` is compatible with existing systems only when such compatibility is judged valuable and requires at most modest retargeting effort. This criterion rules out the use of the debugging information generated by standard compilers and linkers. Conversations with the implementors of other debuggers, corroborated by examination of `gdb` and `ups`, suggest that substantial retargeting effort is expended coping with a multiplicity of executable-file and symbol-table formats. PostScript is the same on all machines. `ldb` is also incompatible with existing operating-system support for debugging, both because of the retargeting effort involved and because the problem is worth studying.

I judged it necessary that `ldb` be compatible with existing calling sequences so that users could debug code linked with existing libraries. The primary criterion of compatibility is that `ldb` should give accurate, if incomplete, information about every procedure on the call stack, even if no PostScript debugging information is available for some of the active procedures. In particular, it must be able to walk past such procedures, even if it is unable to restore registers saved by such procedures. It is common to find such procedures on the stack, e.g., library procedures not compiled with `lcc`. Meeting this criterion requires extra retargeting effort on the MIPS because `ldb` needs 100 lines of machine-dependent code to find the frame sizes of procedures not compiled with `lcc`. Only a few lines would be needed if frame sizes were stored in memory next to procedures, as are register-save masks on the VAX. The assembler could place sizes next to code without adding to the cost of a call.

`ldb` is not fully compatible with existing calling sequences because procedure calls are not atomic except on the VAX. As described in Chapter 8, a procedure is called in three steps, which change the program counter, allocate a stack frame, and save registers. Like `dbx`, `ldb` gives wrong answers if a target stops in an intermediate state between steps, e.g., after the program counter has changed but before all registers have been saved. Such stops are rare; they occur if a program is interrupted, if it branches to an invalid address, or if a user makes a mistake in instruction-level single stepping. The retargeting costs of identifying the steps are high. A debugger must scan machine code to determine if it is in a function prolog or epilog, whether a stack frame has been created, and whether registers have been saved. Because calling sequences are poorly documented, it is difficult to determine what instruction sequences constitute legal prologs and epilogs and to identify the steps. Instruction scheduling further complicates the problem; for example, the MIPS assembler may move instructions from the body of a procedure into its entry sequence. At best, such movement makes it more difficult to identify the steps. At worst, it can introduce new intermediate states in which the values of some preserved registers are only on the stack but the values of others are not on the stack. It is difficult to estimate the retargeting effort required to handle intermediate states in procedure calls. Scanning machine code presupposes instruction decoding of the kind used to implement follow sets and disassembly on the MIPS and SPARC. I expect the intellectual effort required to be greater than that required to implement follow sets and disassembly, but I expect less code to be required than for disassembly.

A more reasonable solution to the problem of intermediate states in procedure calls is to restrict calling sequences and instruction scheduling to make it easy to identify the steps (Digital 1992). Such restrictions need not have an adverse impact on performance. The retargeting cost could be further reduced by having compilers emit labels identifying the steps. The implementation should follow the strategy `ldb` uses for register-save information. Represent the information in a machine-independent form, and have the compiler provide it when possible. When the compiler does not provide it, use a machine-dependent method or procedure to get the information in a machine-dependent way (e.g., by scanning machine code), and translate the information so obtained into the machine-independent form that the compiler would have provided. This technique isolates the necessarily machine-dependent code that gets the information, separating it from the possibly machine-independent code that uses the information.

Not all unusual retargeting costs are unreasonable or due to poor specifications. For example, the extra cost of supporting procedure call on the MIPS is inherent in the calling sequence because the registers in which arguments are passed depend not only on the positions of the arguments, but also on their types. This arrangement speeds up procedures that pass floating-point arguments because it reduces movement between floating-point and integer registers. This purpose justifies the modest extra effort required in the debugger. Luckily, the calling sequence permits that the same argument be passed in both an integer and a floating-point register; otherwise the nub's supporting code would be even more complicated.

Different retargeting costs are associated with different instruction sets. The specification of the SPARC instruction set is 67% longer than that of the MIPS because the SPARC instruction set is less regular and has more formats. Comparison with the VAX and 68020 awaits the extension of `ldb`'s specification language better to handle instructions that vary in size. The complexity of the instruction set determines the difficulty of implementing breakpoints. As discussed in Chapter 7, no instruction-set specification is needed if one settles for breakpoints only at stopping points. Even though the cost of implementing follow-set breakpoints is high relative to the cost of implementing `ldb`'s other machine-dependent parts, it is still low in absolute terms—less than 350 lines of specification and Modula-3. Although some machines provide hardware support for single stepping, these results suggest that hardware assistance for debugging would be better used to solve other problems, like implementing watchpoints (Intel 1986, Section 2.12).

11.6 Recommendations

The designers of language run-time environments and operating systems can make implementing and retargeting debuggers easier. Better specifications of calling sequences are needed. The specification of a calling sequence should describe *all* of the frames that might appear on the stack, including the bottom-most frame and "interrupt" frames that contain the machine state after delivery of a signal (Linton 1990). As argued above, such a specification should also restrict entry and exit sequences to make it easy to identify the intermediate steps in calling a procedure.

At the operating-system level, `ldb` confirms that treating all debugging as remote debugging is as practical and efficient as using the operating system to mediate all interactions between debugger and target (Redell 1989). Both techniques are much less efficient than running the debugger and target in the same address space. `ldb` suggests that implementing procedure call in the user space of the target process, not in the debugger, reduces retargeting effort. To duplicate such an implementation in a more traditional system, in which the kernel provides the debugging support, would require using mechanisms like those used to deliver signals to a target process, with the nub's `call` function playing the role of signal handler.

On only one of `ldb`'s four targets does the operating system give signal handlers access to the entire state of the machine. The other operating systems save this state in user space, but they do not make all of it accessible to signal handlers. I can identify no technical reason why not. Such access would simplify `ldb`'s debug nub, as well as other applications that need the state, like user-level threads (Cormack 1988).

In addition to the PostScript interpreter, some of the techniques used in `ldb` should have applications beyond debugging. Any interpreter could profit by using compact specifications to generate code that performs run-time type and limit checks. `ldb`'s support for fast lexical analysis makes standard, unsafe techniques available in a safe way in Modula-3. Instruction decoding and encoding are closely related, and further work on `ldb`'s instruction-specification language could make it useful for writing assemblers as well as disassemblers. The techniques used to choose a target architecture dynamically, particularly that of using machine-dependent subtypes of machine-independent types, could be used in other retargetable tools.

11.7 Further work

`ldb` implements basic debugging in a retargetable way. It can be used as a basis for improving the way debugging is done. Many improvements proposed for debugging would be considered part of `ldb`'s user interface, including direct-manipulation interfaces (Cargill 1986; Russell 1992) and programmability and support for events (Olsson, Crawford, and Ho 1991).

`ldb` is designed to debug more than one process simultaneously, although the current user interface can manipulate at most one at a time. In a single-threaded operating system, using network connections to manipulate target processes has advantages for a multi-process debugger, because facilities already exist for reading from any of a set of connections. If a program computes different answers when run on different machines, it might be possible to run both versions under the control of a single debugger, asking the debugger to run them in parallel until the states diverge. What it means for two versions to be "the same program" in the presence of conditional compilation is a research question. Instruction- and cycle-level simulators are used to study problems in architecture (Larus 1990; Rogers and Li 1992); a two-process debugger could be used to execute a program on two versions of such a simulator or on the simulator and the hardware and to explore differences in the two executions. `ldb` could also be used to debug distributed applications, like file systems, that run on networks of heterogeneous computers.

Events have been used as a basis for debugging both distributed and sequential programs (Bates and Wileden 1983; Bruegge 1985; Olsson, Crawford, and Ho 1991). As described in Chapter 7, `ldb` uses two event-matching mechanisms: one translates machine-dependent events into machine-independent events, and one handles machine-independent events. `ldb`'s breakpoint commands are easily implemented using low-level breakpoints plus the machine-independent event mechanism. The problem is that no event mechanism is available at the user level; for example, there is no way for a user to arrange for the values of certain expressions to be printed every time a particular breakpoint occurs. Similarly, there is no way for users to single step by instructions until a new procedure is called. These commands could be implemented using the existing mechanism, but it might be wise to make the general mechanism available to sophisticated users until there is a consensus on a standard set of breakpoint facilities.

`ldb` supports only one programming language, ANSI C, and one compiler, `lcc`. Although it appears that the results in Chapters 4 and 5 could be repeated using another language or compiler, it is too soon to tell. One unresolved question is how to manage name spaces different from C's, particularly in languages that support overloading.

Support for multiple languages simultaneously is more difficult. Some debuggers require the user to choose a language (Beander 1983); others identify symbols as belonging to a particular language (Linton 1990). Some procedures may belong to more than one language at a time, for example, a procedure might be written in a high-level language that is compiled to C, like Modula-3 or C++. The implementors of such languages would benefit from a debugger providing more than one view of such procedures, for example, a high-level view, a C view, and a machine-language view. In a graphical debugger, it might be useful to overload certain commands with different meanings depending on the view, for example, to show assembly, C, or high-level source depending on the view, or to vary the size of a single step depending on the view.

Previous work has made dramatic improvements in the efficiency of basic debugging operations by putting the target and debugger in the same address space. Code-patching implementations of breakpoints are a thousand times faster than those requiring operating-system intervention. Conditional breakpoints can be made efficient by generating and patching in machine code that evaluates conditions. Latency can be reduced by offloading debugging work onto a separate processor in the same address space. These techniques are not yet widely used, and they could benefit from a retargetability study analogous to `ldb`'s study of more commonly used techniques.

There are many questions about language and operating-system support for debugging that `ldb` leaves unexplored. This area is exciting because it offers an opportunity to expand the basic functionality that debuggers provide. Different computer systems provide such features as data watchpoints, memory protection, multiple threads, signals, and exceptions. Such features can be provided by hardware, kernel software, user software (language run-time system), or a combination. A more ambitious retargetable debugger would take advantage of such features when they were available and cope gracefully when they were unavailable. For example, a debugger might make it possible to switch among threads of a multithreaded program, to learn what threads hold what locks, or to display the "waits for" relation. If a language has exceptions, a debugger might cooperate with the run-time system to get control just before a particular exception is handled, letting the run-time system identify the handler. Problems to be solved include identifying what facilities the implementors of threads and exceptions should provide to support debugging and showing how a debugger should be structured to take advantage of such facilities. The other features mentioned above pose similar problems. Support for some such features will require extensions to `ldb`'s debug-nub protocol; another problem is to construct a protocol that admits subsets.

11.8 Envoi

Experience implementing `ldb` suggests which engineering techniques deserve to be re-used by implementors of other debuggers. This section casts those techniques as a set of imperatives.

Use a standard, machine-independent format for debugging information. Do not use a language-dependent format that must be extended for new languages; build extensibility into the format. Simplify the design by starting with a basic format containing no language-dependent information, but providing for procedures that print values. Extend it once to hold the machine-dependent, language-dependent information needed by those procedures, and extend it again to hold the information needed for expression evaluation. Design it for incremental input. For C, find a way to eliminate duplication that arises from including the same header files in multiple compilation units.

Get the compiler to do as much work as possible. It knows how data structures are represented and how to evaluate expressions; re-implementing those features in a debugger wastes effort. Leverage the compiler by having it include all machine-dependent information in its debugging output; do not force the debugger to infer machine-dependent information, like sizes and alignments, from the names of types. Get all compilers to provide register-save information; machine-dependent debugger code can be eliminated. For expression evaluation, leverage the compiler by having the debugger implement a language that looks like the compiler's intermediate code. The language the debugger implements should manipulate an abstraction that resembles hardware, e.g., an abstract memory. For run-time support, do not have the debugger implement procedure call by simulating the compiler's actions in calling a procedure; get the compiler itself to generate the code needed.

Work closely with the design of the language run-time system. Careful specification of calling sequences makes it easy to implement stack walking for debugging and exception handling, context switching for user-level threads, and nonlocal goto (e.g., C `longjmp`). Poor specification makes them all difficult. If the operating system puts frames on the stack, e.g., for signals or interrupts, make sure those frames conform to the specification.

Do not limit a debugger to debugging child processes. Encourage operating-system designers to provide support for remote and cross-architecture debugging, e.g., a network protocol with fixed byte order. If support is not available from the operating system, consider whether user code like `ldb`'s nub will serve.

A few suggestions are applicable to other programs. Do not carry information hiding to extremes. Generate run-time checking code and machine-dependent code from compact specifications. Represent machine-dependent information in a machine-independent form; create opportunities for using machine-independent code by separating the gathering and use of such information. Write programs that handle multiple targets simultaneously; call machine-dependent procedures instead of using conditional compilation. When conditional compilation cannot be avoided, using macros that have machine-independent meanings but machine-dependent definitions is better than littering programs with conditional-compilation directives.

References

Adams, Evan and Steven S. Muchnick. 1986 (July).
Dbxtool: A window-based symbolic debugger for Sun workstations.
Software—Practice & Experience, 16(7):653–669.

Adobe Systems Incorporated. 1985.
PostScript Language Reference Manual.
Reading, Ma: Addison-Wesley.

Adobe Systems Incorporated. 1990.
PostScript Language Reference Manual. Second edition.
Reading, Ma: Addison-Wesley.

American National Standard Institute, Inc. 1990.
American National Standards for Information Systems, Programming Language C ANSI X3.159–1989.
New York.

Appel, Andrew W. 1989 (February).
Simple generational garbage collection and fast allocation.
Software—Practice & Experience, 19(2):171–183.

———. 1990.
A runtime system.
Lisp and Symbolic Computation, 3:343–380.

———. 1992.
Compiling with Continuations.
Cambridge: Cambridge University Press.

Aral, Ziya, Ilya Gertner, and Greg Schaffer. 1989 (May).
Efficient debugging primitives for multiprocessors.
Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems, in a special issue of *SIGPLAN Notices*, 24:87–95.

Balzer, Robert M. 1969.
EXDAMS—EXtendable Debugging and Monitoring System.
In *AFIPS Proceedings Spring Joint Computer Conference*, pages 567–580, Arlington, VA. AFIPS Press.

Bates, Peter C. and Jack C. Wileden. 1983 (December).
High-level debugging of distributed systems: The behavioral abstraction approach.
Journal of Systems and Software, 3(4):255–264.

Baudinet, Marianne and David MacQueen. 1985 (December).
Tree pattern matching for ML (extended abstract).
Unpublished manuscript, AT&T Bell Laboratories.

Beander, Bert. 1983 (August).
`VAX DEBUG`: An interactive, symbolic, multilingual debugger.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, in *SIGPLAN Notices*, 18(8):173–179.

Bentley, Jon. 1988.
More Programming Pearls: Confessions of a Coder.
Reading, MA: Addison-Wesley.

Birrell, Andrew D. 1991.
An introduction to programming with threads.
In Nelson, Greg, editor, *Systems Programming with Modula-3*, chapter 4, pages 88–118. Englewood Cliffs, NJ: Prentice Hall.

Brooks, Gary, Gilbert J. Hansen, and Steve Simmons. 1992 (July).
A new approach to debugging optimized code.
ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 27(7):1–11.

Bruegge, Bernd. 1985 (September).
Adaptability and Portability of Symbolic Debuggers.
PhD thesis, Carnegie Mellon University.

Cargill, Thomas A. 1983 (August).
The Blit debugger.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, in *SIGPLAN Notices*, 18(8):190–200.

———. 1986 (January).
The feel of Pi.
In *Proceedings of the Winter USENIX Conference*, pages 62–71, Denver, CO.

Caswell, Deborah and David Black. 1990 (January).
Implementing a Mach debugger for multithreaded applications.
In *Proceedings of the Winter USENIX Conference*, pages 25–39, Washington, DC.

Chow, Fred C. and John L. Hennessy. 1990 (October).
The priority-based coloring approach to register allocation.
ACM Transactions on Programming Languages and Systems, 12(4):501–536.

Clinger, William D. and Jonathan Rees. 1991 (July-September).
Revised^4 report on the algorithmic language Scheme.
LISP Pointers, IV(3):1–55.

Clinger, William D. 1990 (June).
How to read floating-point numbers accurately.
Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 25(6):92–101.

Cormack, Gordon V. 1988 (May).
A micro-kernel for concurrency in C.
Software—Practice & Experience, 18(5):485–491.

Crawford, Richard H., Ronald A. Olsson, W. Wilson Ho, and Christopher E. Wee. 1992 (April).
Semantic issues in the design of languages for debugging.
In *Proceedings of the International Conference on Computer Languages*, pages 252–261, Oakland, CA.

DeTreville, John. 1986 (March).
Designing Loupe: A Modula-2+ debugger.
Unpublished manuscript, DEC Systems Research Center.

Digital Equipment Corporation. 1975.
DDT—Dynamic Debugging Technique.
Maynard, MA.

Digital Equipment Corporation. 1992 (October).
Open VMS Calling Standard. Order Number AA-PQY2A-TK.
Maynard, MA.

Ellis, Margaret A. and Bjarne Stroustrup. 1990.
The Annotated C++ Reference Manual.
Reading, MA: Addison-Wesley.

Feldman, Stuart I. and Channing B. Brown. 1988 (May).
IGOR: A system for program debugging via reversible execution.
Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, in *SIGPLAN Notices*, 24(1):112–123.

Fernandez, Mary F., Christopher W. Fraser, and David R. Hanson. 1992 (November).
Retargetable link-time code generation.
Unpublished manuscript, Department of Computer Science, Princeton University.

Fraser, Christopher W. and David R. Hanson. 1982 (April).
A machine-independent linker.
Software—Practice & Experience, 12(4):351–366.

———. 1991a (September).
A code generation interface for ANSI C.
Software—Practice & Experience, 21(9):963–988.

———. 1991b (October).
A retargetable compiler for ANSI C.
SIGPLAN Notices, 26(10):29–43.

———. 1992 (January).
Simple register spilling in a retargetable compiler.
Software—Practice & Experience, 22(1):85–99.

Fritzson, Peter. 1983 (December).
Symbolic debugging through incremental compilation in an integrated environment.
Journal of Systems and Software, 3(4):285–294.

Golan, Michael and David R. Hanson. 1993 (January).
DUEL — A very high-level debugging language.
In *Proceedings of the Winter USENIX Conference*, pages 109–120, San Diego, CA.

Gorlick, Michael M. 1991 (December).
The flight recorder: An architectural aid for system monitoring.
Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, in *SIGPLAN Notices*, 26(12):175–183.

Gramlich, Wayne C. 1983 (August).
Debugging methodology (session summary).
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, in *SIGPLAN Notices*, 18(8):1–3.

Griswold, Ralph E. and Madge T. Griswold. 1990.
The Icon Programming Language. Second edition.
Englewood Cliffs, NJ: Prentice Hall.

Hennessy, John L. 1982 (July).
Symbolic debugging of optimized code.
ACM Transactions on Programming Languages and Systems, 4(4):323–344.

Holzmann, Gerard J. 1991.
Design and Validation of Computer Protocols.
Englewood Cliffs, NJ: Prentice Hall.

Intel Corporation. 1986 (April).
80386 data sheet.
Santa Clara, CA.

Kalsow, Bill and Eric Muller. 1992 (June).
SRC Modula-3.
DEC Systems Research Center. Version 2.07. Available by anonymous `ftp` from Internet host `gatekeeper.dec.com`.

Kane, Gerry. 1988.
MIPS RISC Architecture.
Englewood Cliffs, NJ: Prentice Hall.

Kessler, Peter B. 1990 (June).
Fast breakpoints: Design and implementation.
Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 25(6):78–84.

Killian, T. J. 1984.
Processes as files.
In *Proceedings of the Summer USENIX Conference*, pages 203–207, Salt Lake City.

Knuth, Donald E. 1973.
The Art of Computer Programming. Volume 1, *Fundamental Algorithms.* Second edition.
Reading, MA: Addison-Wesley.

Lampson, Butler W. and David D. Redell. 1980 (February).
Experience with processes and monitors in Mesa.
Communications of the ACM, 23(2):105–117.

Larus, James R. 1990 (September).
SPIM S20: A MIPS R2000 simulator.
Technical Report 966, Computer Sciences Department, University of Wisconsin, Madison, WI.

Leblang, David B. and Robert P. Chase, J. 1984 (May).
Computer-aided software engineering in a distributed workstation environment.
Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, in *SIGPLAN Notices*, 19(5):104–112.

Linton, Mark A. 1990 (June).
The evolution of Dbx.
In *Proceedings of the Summer USENIX Conference*, pages 211–220, Anaheim, CA.

Liskov, Barbara H. and Alan Snyder. 1979 (November).
Exception handling in CLU.
IEEE Transactions on Software Engineering, SE-5(6):546–558.

Miller, Barton and Thomas LeBlanc, editors. 1988 (May).
ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging.
Available as *SIGPLAN Notices* 24(1), January, 1989.

Miller, Barton and Charles McDowell, editors. 1991 (May).
ACM/ONR Workshop on Parallel and Distributed Debugging.
Available as *SIGPLAN Notices* 26(12), December, 1991.

MIPS Computer Systems. 1989 (May).
MIPS Assembly Language Programmer's Guide.
Mountain View, CA.

Moore, C. H. 1974.
FORTH: A new way to program a mini-computer.
Astron. Astrophys. Suppl., 15:497–511.

Myers, Brad Allan. 1980 (June).
Displaying data structures for interactive debugging.
Technical Report CSL-80-7, Xerox PARC, Palo Alto, CA.

Nelson, Greg, editor. 1991.
Systems Programming with Modula-3.
Englewood Cliffs, NJ: Prentice Hall.

Olsson, Ronald A., Richard H. Crawford, and W. Wilson Ho. 1991 (February).
A dataflow approach to event-based debugging.
Software—Practice & Experience, 21(2):209–229.

Oppen, Derek C. 1980 (October).
Prettyprinting.
ACM Transacations on Programming Languages and Systems, 2(4):465–483.

Ousterhout, John K. 1990 (January).
Tcl: An embeddable command language.
In *Proceedings of the Winter USENIX Conference*, pages 133–146, Washington, DC.

Pike, Rob, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1992 (September).
The use of name spaces in Plan 9.
In *SIGOPS European Workshop on Distributed Systems*, Mont Saint-Michel.

Ramsey, Norman and David R. Hanson. 1992 (July).
A retargetable debugger.
ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 27(7):22–31.

Ramsey, Norman. 1992 (August).
Literate-programming tools need not be complex.
Technical Report CS-TR-351-91, Department of Computer Science, Princeton University.
Submitted to *IEEE Software*.

Redell, David D., Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. 1980 (February).
Pilot: An operating system for a personal computer.
Communications of the ACM, 23(2):81–92.

Redell, David D. 1989 (January).
Experience with Topaz TeleDebugging.
Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, in *SIGPLAN Notices*, 24(1):35–44.

Rogers, Anne and Kai Li. 1992 (September).
Software support for speculative loads.
Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, in *SIGPLAN Notices*, 27(9):38–50.

Russell, Mark. 1992.
`ups`.
`ups` is a graphical, source-level debugger for C. It is available for anonymous `ftp` from host `unix.hensa.ac.uk` (129.12.21.7), in `/misc/unix/ups/ups-2.45.tar.Z`. The author's email address is `mtr@ukc.ac.uk`.

Silicon Graphics, Inc. 1991.
Graphics Library Programming Guide. Document Number 007-1210-040.
Mountain View, CA.

Stallman, Richard M. and Roland H. Pesch. 1991.
Using GDB: A guide to the GNU source-level debugger, GDB version 4.0.
Technical report, Free Software Foundation, Cambridge, MA.

Steele, Guy L., J and Jon L. White. 1990 (June).
How to print floating-point numbers accurately.
Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 25(6):112–126.

Sun Microsystems. 1990a.
Network Programming. Part number 800-3850-10, revision A.
Mountain View, CA.

Sun Microsystems. 1990b (January).
UNIX Programmer's manual, Sun Release 4.1, Section 1.
Mountain View, CA.
The manual page for `ld` describes shared libraries.

Swinehart, Daniel. C., Polle T. Zellweger, Richard J. Beach, and Robert. B. Hagmann. 1986 (October).
A structural view of the Cedar programming environment.
ACM Transactions on Programming Languages and Systems, 8(4):419–490.

Tolmach, Andrew P. and Andrew W. Appel. 1990 (June).
Debugging Standard ML without reverse engineering.
In *ACM Conference on LISP and Functional Programming*, pages 1–12, Nice, France.

US Department of Defense. 1983.
The Ada Programming Language Reference Manual.
US Government Printing Office.
ANSI/MILSTD 1815A.

Wahbe, Robert. 1992 (September).
Efficient data breakpoints.
Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, in *SIGPLAN Notices*, 27(9):200–212.

Waite, William M. 1986 (May).
The cost of lexical analysis.
Software—Practice & Experience, 16(5):473–488.

Wall, David W. 1992.
Experience with a software-defined machine architecture.
ACM Transactions on Programming Languages and Systems, 14(3):299–338.

Weiser, Mark, Alan Demers, and Carl Hauser. 1989 (December).
The portable common runtime approach to interoperability.
Proceedings of the 12th Symposium on Operating Systems Principles, in *Operating Systems Review*, 23(5):114–122.

Welch, Terry A. 1984 (June).
A technique for high-performance data compression.
IEEE Computer, 17(6):8–19.

Wu, Sun and Udi Manber. 1992 (October).
Fast text searching allowing errors.
Communications of the ACM, 35(10):83–91.

Zellweger, Polle T. 1984.
Interactive Source-Level Debugging of Optimized Programs.
PhD thesis, University of California, Berkeley.
Available from Xerox PARC as CSL Technical Report 84–5.

Zurawski, Larry W. and Ralph E. Johnson. 1991 (April).
Debugging optimized code with expected behavior.
Unpublished manuscript. Available by anonymous `ftp` from `st.cs.uiuc.edu` in `/pub/papers`.

Appendix A

A Formal Model of Breakpoints

This appendix provides a formal model of `ldb`'s follow-set breakpoints. The model takes the form of a PROMELA program (Holzmann 1991). PROMELA programs define several threads of control that communicate by passing messages. Each thread of control runs a program written in a guarded-command language with a C-like syntax. Programs may be nondeterministic. PROMELA can simulate the execution of a program and search its state space for states violating assertions embedded in the program. The simulator also searches for states with no successors, i.e., deadlocks.

The PROMELA code in this appendix models `ldb`'s implementation of breakpoints. Although `ldb` does not work with multithreaded programs, the model uses multiple threads because a procedure call from `ldb` to a target process effectively creates a new thread. The assertions embedded in the model specify that the debugger takes a breakpoint action just before any thread's successful execution of the instruction at the breakpoint. Breakpoints may be implemented either in the operating system or in the debugger itself; the choice does not affect the model used here. The model assumes it can plant trap instructions in the instruction stream of the target program, and that it will be notified when the target program encounters a trap. The model also suits a machine with a "trace mode" that causes a trap after the execution of every instruction.

The model has a single breakpoint. To keep the state space small, the model has only two threads, so that a single bit can represent thread `ids`.

175 ⟨*declarations* 175⟩≡

```
#define NTHREADS 2
#define threadid bit
```

The 175 in ⟨*declarations* 175⟩ is the page number on which the definition appears.

A.1 Modeling the program counter and execution

To keep things simple, I partition the possible values of the program counter into three sets:

`Break`	the breakpoint itself,
`Follow`	the instruction(s) following the breakpoint,
`Outside`	outside the breakpoint.

The three sets are modeled by the following constants.

176a ⟨*declarations* 175⟩+≡

```
#define NPCS 3
#define Break   0       /* pc at the breakpoint */
#define Follow  1       /* pc in breakpoint's follow set */
#define Outside 2       /* all other pc's */
```

The ability to plant traps is modeled by the array `trapped`, which records whether a trap instruction has been stored at a particular location:

176b ⟨*declarations* 175⟩+≡

```
bool trapped[NPCS];
```

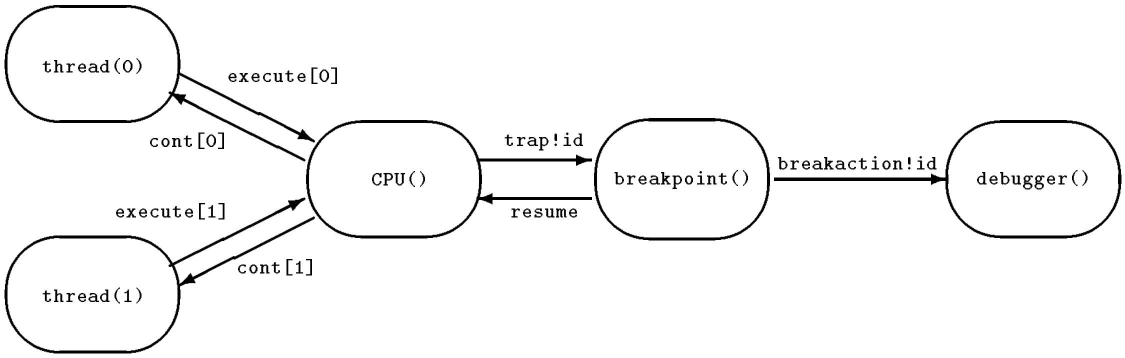
The model has five active components: two threads, a CPU that executes one thread at a time, the breakpoint, and the rest of the debugger. Here are the channels that are used for communication between the threads, the CPU, the breakpoint, and the debugger. Taking a breakpoint action is modeled by sending a message on the channel `breakaction`.

176c ⟨*declarations* 175⟩+≡

```
chan execute[NTHREADS] = [0] of {bit};    /* try to execute instruction */
chan cont[NTHREADS]    = [0] of {bit};    /* instruction executed */
chan trap              = [0] of {byte};   /* CPU trapped on id! */
chan resume            = [0] of {bit};    /* debugger resumed after trap */
chan breakaction       = [0] of {byte};   /* deliver breakpoint to debugger */
```

`[0]` indicates that the channels are synchronous; senders block until a receiver is ready and vice versa.

The communication structure is:



The CPU repeats the following steps.

1. Wait for a thread to attempt to execute the instruction at `pc`.
2. If the instruction is a trap, notify the debugger. When the debugger tells the CPU to resume, `pc` is unchanged.
3. If the instruction is not a trap, advance `pc`.
4. Ask the thread to continue executing.

There is only one debugger, but there are multiple threads, and each one has its own `pc` and its own communication with the CPU. When the CPU notifies the debugger of a trap, it identifies the trapping thread. Other messages are used only for synchronization, so they send and receive the nonsense variable `x`.

177 ⟨*declarations* 175⟩+≡

```
bit x;                  /* junk variable for sending messages */
```

A `proctype` is a procedure that a thread can execute; this one models the CPU. `c?x` receives the value `x` on channel `c`; `c!x` sends. Arrows (`->`) separate guards from commands.

178a ⟨*proctypes* 178a⟩≡

```
proctype CPU(byte count) {
  threadid id = 0;
  do
  :: execute[id]?x ->
       if
       ::  trapped[pc[id]] -> trap!id ; resume?x
       :: !trapped[pc[id]] -> ⟨advance pc[id]{} 178b⟩
       fi;
       cont[id]!x;
       ⟨possible context switch (change of id) 184b⟩
  od
}
```

Context switching is discussed below.

Since the program counter is an abstraction, advancing it does not mean incrementing it. A successful execution at `Break` is guaranteed to be followed by an attempt to execute `Follow`; aside from that, any instruction can follow any other.

178b ⟨*advance* `pc[id]{}` 178b⟩≡

```
if
:: pc[id] == Break -> pc[id] = Follow
:: pc[id] != Break -> /* any instruction can be next */
     if
     :: pc[id] = Outside
     :: pc[id] = Break
     :: pc[id] = Follow
     fi
fi
```

The second `if` statement has no guards, so an alternative is chosen nondeterministically.

All threads begin execution outside the breakpoint.

178c ⟨*declarations* 175⟩+≡

```
byte pc[NTHREADS];
```

178d ⟨*initialize data for thread* `id` 178d⟩≡

```
pc[id] = Outside;
```

A.2 Counting events

The correctness criterion for the breakpoint implementation is that one breakpoint action must be taken for every successful execution of an instruction at `Break`. `threadcount[id]` counts how many times thread `id` has executed the breakpoint, and `actioncount[id]` counts how many breakpoint actions have been taken on behalf of thread `id`.

179a ⟨*declarations* 175⟩+≡

```
byte threadcount[NTHREADS];
byte actioncount[NTHREADS];
```

179b ⟨*initialize data for thread* `id` 178d⟩+≡

```
threadcount[id] = 0;
actioncount[id] = 0;
```

Here is the model of a thread, including the assertion that the thread and debugger counts are the same:

179c ⟨*proctypes* 178a⟩+≡

```
proctype thread(threadid id) {
  do
  :: if
     :: pc[id] == Break -> execute[id]!x; cont[id]?x;
           ⟨if successfully executed Break, increment threadcount[id]{} 180a⟩
     :: pc[id] != Break -> execute[id]!x; cont[id]?x
     fi;
     assert(pc[id] != Outside || threadcount[id] == actioncount[id])
  od
}
```

The corresponding model of the debugger is

179d ⟨*proctypes* 178a⟩+≡

```
proctype debugger() {
  threadid id;
  do
  :: atomic { breakaction?id -> ⟨increment actioncount[id]{} 180c⟩ }
  od
}
```

`atomic` groups statements into a single atomic action. When the debugger takes a breakpoint action, it atomically increments `actioncount[id]`. Without `atomic`, it might delay incrementing the counter and invalidate the assertion above.

A thread knows it has successfully executed `Break` if the `pc` has changed:

180a ⟨*if successfully executed* `Break`*, increment* `threadcount[id]{}` 180a⟩≡

```
if
:: pc[id] != Break -> ⟨increment threadcount[id]{} 180b⟩
:: pc[id] == Break -> skip
fi
```

To keep the state space small, I restrict the values of the counters to be in the range `0..3`.

180b ⟨*increment* `threadcount[id]{}` 180b⟩≡

```
threadcount[id] = (threadcount[id] + 1) % 4
```

180c ⟨*increment* `actioncount[id]{}` 180c⟩≡

```
actioncount[id] = (actioncount[id] + 1) % 4
```

A.3 Implementing the breakpoint

There is a long tradition of implementing breakpoints using traps and single stepping. To set a breakpoint at I, plant a trap at I. When the target program hits the trap, that's a breakpoint event. To resume execution after the breakpoint, restore the original instruction to I, single step the machine to execute just the instruction at I, and once again plant a trap at I and continue execution. Not all machines have a single-step mode in hardware, but single stepping can be simulated in software by using more trap instructions. In my model, I eliminate single stepping entirely, working directly with trap instructions and a follow set (modeled by `Follow`).

The simpler model does not preclude the use of hardware single stepping. One of the operations in the model is planting traps at the locations in the follow set of an instruction. This operation can be implemented either by computing the follow set and planting actual traps, or by setting a trace bit on a machine with hardware single stepping.

An active breakpoint is trapped either on the instruction of the breakpoint itself or on that instruction's follow set. The breakpoint keeps track of which state it is in, with the following invariant.

```
   breakstate == Break  && trapped[Break] = 1 && trapped[Follow] = 0
|| breakstate == Follow && trapped[Break] = 0 && trapped[Follow] = 1
```

180d ⟨*declarations* 175⟩+≡

```
byte breakstate = Break;
```

181a ⟨*initialization* 181a⟩≡

```
trapped[Break] = 1;
```

Changing the state preserves the invariant.

181b ⟨*move traps to* `Break` 181b⟩≡

```
atomic { breakstate = Break;  trapped[Break] = 1; trapped[Follow] = 0 }
```

181c ⟨*move traps to* `Follow` 181c⟩≡

```
atomic { breakstate = Follow; trapped[Break] = 0; trapped[Follow] = 1 }
```

It's necessary to keep track of the state of each thread with respect to the breakpoint. A thread is "in the breakpoint" if it has trapped at `Break`, and it does not "leave the breakpoint" until it traps at `Follow`. Threads are initially outside the breakpoint.

181d ⟨*declarations* 175⟩+≡

```
bit  inbreak[NTHREADS];
```

181e ⟨*initialize data for thread* `id` 178d⟩+≡

```
inbreak[id] = 0;
```

One possible implementation just keeps track of the various states and delivers a breakpoint event at the right time:

182 ⟨*candidate breakpoint implementation* 182⟩≡

```
proctype breakpoint() {
  threadid id;

  do
  :: trap?id ->
       if
       :: breakstate == Break ->
            if
            :: !inbreak[id] -> breakaction!id ; inbreak[id] = 1
            ::  inbreak[id] -> skip /* no event */
            fi;
            ⟨move traps to Follow 181c⟩
       :: breakstate == Follow ->
            if
            ::  inbreak[id] -> inbreak[id] = 0
            :: !inbreak[id] -> skip
            fi;
            ⟨move traps to Break 181b⟩
       fi;
       resume!x
  od
}
```

This implementation works fine for a single thread. With two threads, the PROMELA state-space search finds the following erroneous execution sequence (attempted executions that trap are marked with a *):

breakpoint (debugger)	CPU	thread 0	thread 1
		`Outside`	
		`Break`*	
⟨*take breakpoint action*⟩			
⟨*move traps to* `Follow` *181*⟩			
resume			
	context switch		
			`Outside`
			`Break`
			`Follow`*
⟨*take no action*⟩			
⟨*move traps to* `Break` *181*⟩			
resume			
			`Outside`
	context switch		
		`Follow`	
		`Outside`	

In this execution sequence, thread 1 goes through the breakpoint without triggering a breakpoint action. In an earlier version of `ldb`, this sequence could be provoked by executing a procedure call after the user's program hit a breakpoint; the user's program was thread 0, and the procedure call was thread 1.

To prevent such an occurrence, the CPU must not be permitted to change contexts when a thread is in the middle of a breakpoint. If the CPU can change contexts only when `noswitch == 0`, then the following breakpoint implementation works correctly.

183 ⟨*proctypes* 178a⟩+≡

```
proctype breakpoint() {
  threadid id;

  do
  :: trap?id ->
       if
       :: breakstate == Break ->
            if
            :: !inbreak[id] -> breakaction!id ; inbreak[id] = 1
            ::  inbreak[id] -> assert(0)
            fi;
            noswitch = noswitch + 1;
            ⟨move traps to Follow 181c⟩
```

```
        :: breakstate == Follow ->
             if
             ::  inbreak[id] -> inbreak[id] = 0
             :: !inbreak[id] -> assert(0)
             fi;
             noswitch = noswitch - 1;
             ⟨move traps to Break 181b⟩
        fi;
        resume!x
   od
 }
```

The ban on context switching makes it possible to strengthen `skip` to `assert(0)`.

`noswitch` is declared to be a counter, not a bit, because that implementation generalizes to multiple breakpoints.

184a ⟨*declarations* 175⟩+≡

```
 byte noswitch = 0;
```

The CPU code to do the context switching correctly is:

184b ⟨*possible context switch (change of* `id`*)* 184b⟩≡

```
 if
 :: noswitch == 0 -> ⟨set id randomly 184c⟩
 :: noswitch  > 0 -> skip
 fi
```

184c ⟨*set* `id` *randomly* 184c⟩≡

```
 atomic {
   if
   :: id = 0
   :: id = 1
   fi
 }
```

A.4 Completing the model

The boilerplate needed to turn the model into a complete PROMELA specification is:

185a $\langle$* 185a$\rangle\equiv$

```
⟨declarations 175⟩
⟨proctypes 178a⟩
init {
  threadid id;
  atomic {
    ⟨initialization 181a⟩
    ⟨for 0 ≤ id < NTHREADS, initialize data for thread id 185b⟩;
    run thread(0);
    run thread(1);
    run debugger();
    run breakpoint();
    run CPU (2)
  }
}
```

185b $\langle$*for* $0 \leq$ `id` $<$ `NTHREADS`, *initialize data for thread* `id` 185b$\rangle\equiv$

```
id = 0;
do
:: id < NTHREADS -> ⟨initialize data for thread id 178d⟩
   if
   :: id == NTHREADS - 1  -> break
   :: id <  NTHREADS - 1  -> id = id + 1
   fi
od
```

Appendix B

MIPS Instruction Specification and Follow-set Computation

This appendix demonstrates the language `ldb` uses to specify instruction decoding. It shows both the instruction specification and the follow-set computation for the MIPS. `ldb` uses the same specification to implement symbolic disassembly. `ldb` itself does not need to disassemble instructions, but some users like to look at machine code.

B.1 Instruction specification

The specification describes the size of instructions, gives names to fields within instructions, and gives patterns that match different instructions. MIPS instructions are 32 bits wide:

187 ⟨*instruction spec* 187⟩≡

```
bitsize 32
⟨field names 188⟩
⟨pattern bindings 190a⟩
```

The specification is divided into a CPU specification and a floating-point-coprocessor specification.

A CPU instruction field is specified by giving a range of bit positions. Code in the debugger that uses this specification can refer to the values of fields by name. Such values are unsigned, but the sign-extended values can be referred to by prepending "`extend_`" to the name. For example, `extend_offset` refers to the sign-extended value of the `offset` field.

Some of the MIPS fields are listed on page A-3 of the MIPS architecture manual (Kane 1988); others, like `breakcode`, apply to only a couple of instructions. The MIPS manual sometimes uses more than one name for the same field, for example `offset` and `base` are used in place of `immed` and `rs` when describing load and store instructions. The numbers in the specification are the numbers of the starting and ending bit positions, where 0 is the least and 31 the most significant bit.

188 ⟨*field names* 188⟩≡

```
fields word 0:31 op 26:31 rs 21:25 rt 16:20 immed 0:15 offset 0:15 base 21:25
       target 0:25 rd 11:15 shamt 6:10 funct 0:5 cond 16:20 breakcode 6:25
```

Opcode

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	†	†	†	†
3	†	†	†	†	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	†
5	SB	SH	SWL	SW	†	†	SWR	†
6	LWC0	LWC1	LWC2	LWC3	†	†	†	†
7	SWC0	SWC1	SWC2	SWC3	†	†	†	†

SPECIAL

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSCALL	BREAK	†	†
2	MFHI	MTHI	MFLO	MTLO	†	†	†	†
3	MULT	MULTU	DIV	DIVU	†	†	†	†
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	†	†	†	†
6	†	†	†	†	†	†	†	†
7	†	†	†	†	†	†	†	†

BCOND

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						
3								

† Operation codes marked with a dagger cause reserved instruction exceptions and are reserved for future versions of the architecture.

Figure 35: Opcode tables from MIPS architecture manual.

A *pattern* is a predicate on a word. Patterns are described by the following grammar:

Pattern	→	Field *Eqop* Value	constrains a field
	|	*Pattern* `|` *Pattern*	matches either pattern
	|	*Pattern* `&` *Pattern*	matches both patterns
Eqop	→	`==` | `!=`	equality or inequality

where "Field" is an identifier naming a field, and a "Value" is an integer. Names are bound to patterns by the `patterns` statement:

`patterns` {name : *Pattern*}

Finally, a sequence of names enclosed in square brackets can be bound to a sequence of patterns. A sequence of patterns is specified as a single pattern in which one "Value" is a *generator*:

Generator	→	`[` Integer ... `]`	sequence of literals
	|	{ Integer `to` Integer [`columns` Integer]}	generates a sequence of integers

Generators reduce the sizes of specifications and make them look more like the tables that appear in architecture manuals. The `columns` option generates integers in a sequence that corresponds to counting down the columns of a rectangular array instead of across the rows.

The numeric codes for all the MIPS opcodes are described in three tables on page A-87 of the MIPS architecture manual. The tables are reproduced in Figure 35. Normal opcodes are six bits, and they appear in the `op` field of the instruction.

190a ⟨*pattern bindings* 190a⟩≡

```
patterns
 [ special bcond   j       jal     beq     bne     blez    bgtz
   addi    addiu   slti    sltiu   andi    ori     xori    lui
   cop0    cop1    cop2    cop3    _       _       _       _
   _       _       _       _       _       _       _       _
   lb      lh      lwl     lw      lbu     lhu     lwr     _
   sb      sh      swl     sw      _       _       swr     _
   lwc0    lwc1    lwc2    lwc3    _       _       _       _
   swc0    swc1    swc2    swc3    _       _       _       _ ] : op == {0 to 63}
```

This statement creates names for patterns constraining the `op` field. "`op == {0 to 63}`" generates a list of 64 patterns, ranging from "`op == 0`" to "`op == 63`". Each pattern is associated with the corresponding name in the list to the left of the colon. Patterns associated with the name "`_`" are discarded. For example, a word matches the pattern `addiu` if its `op` field is 9. The statement names patterns for each of the opcodes in the table at the top of Figure 35.

Two opcodes, `special` and `bcond`, are used for several instructions. These instructions are decoded by checking the bit-pattern in the `funct` and `cond` fields of the instructions, respectively.

190b ⟨*pattern bindings* 190a⟩+≡

```
 [ sll     _       srl     sra     sllv    _       srlv    srav
   jr      jalr    _       _       syscall break   _       _
   mfhi    mthi    mflo    mtlo    _       _       _       _
   mult    multu   div     divu    _       _       _       _
   add     addu    sub     subu    and     or      xor     nor
   _       _       slt     sltu    _       _       _       _ ] :
                special & funct == {0 to 47}

 [ bltz bgez bltzal bgezal ] : bcond & cond == [ 0 1 16 17 ]
```

These statements create new patterns by adding further constraints to the existing patterns `special` and `bcond`.

The rest of the patterns organize the instructions into classes, much as is done in Chapter 3 of the MIPS manual. Immediate-mode instructions are grouped into signed and unsigned variants, depending on whether the immediate operand is to be sign-extended. Other instructions are grouped by assembly-language syntax.

191a ⟨*pattern bindings* 190a⟩+≡

```
patterns
  load    : lb | lbu | lh | lhu | lw | lwl | lwr | sb | sh | sw | swl | swr
  immedS  : addi | addiu | slti | sltiu
  immedU  : andi | ori | xori
  arith3  : add | addu | sub | subu | slt | sltu | and | or | xor | nor
  shift   : sll | srl | sra
  vshift  : sllv | srlv | srav
  arith2  : mult | multu | div | divu
  mfthilo : mfhi | mflo | mthi | mtlo
  jump    : j  | jal
  jumpr   : jr | jalr
  branch1 : blez | bgtz | bltz | bgez | bltzal | bgezal
  branch2 : beq | bne
  copls   : lwc0 | lwc1 | lwc2 | lwc3 | swc0 | swc1 | swc2 | swc3
```

Pages B-5 through B-7 of the MIPS manual introduce a few more field names for the convenience of specifying the floating-point instructions:

191b ⟨*field names* 188⟩+≡

```
fields ft 16:20 fs 11:15 fd 6:10 format 21:24 bit25 25:25
```

The instruction codes for Coprocessor 1 (floating point) are given on page B-28 of the MIPS manual. I use names like "`add.`" instead of "`add.fmt`" because they make it possible to form the full name of the instruction by concatenating the name of the opcode pattern and the name of the format.

192a ⟨*pattern bindings* 190a⟩+≡

```
patterns
 [ add.   sub.   mul.   div.   _      abs.   mov.   neg.
   _      _      _      _      _      _      _      _
   _      _      _      _      _      _      _      _
   _      _      _      _      _      _      _      _
   cvt.s  cvt.d  _      _      cvt.w  _      _      _
   _      _      _      _      _      _      _      _
   c.f    c.un   c.eq   c.ueq  c.olt  c.ult  c.ole  c.ule
   c.sf   c.ngle c.seq  c.ngl  c.lt   c.nge  c.le   c.ngt ] :
                        cop1 & funct == {0 to 63} & bit25 == 1
```

Specifying the branch and move instructions is more complicated, because the relevant codes span several entries in the table, and the pattern language is designed to bind one pattern to one entry. I introduce two new fields to simplify the specification. The field names do not have to be defined before all of the pattern bindings, only before those bindings in which they are used.

192b ⟨*pattern bindings* 190a⟩+≡

```
fields cop1code 22:25 copbcode 16:16
patterns
   [ mfc1 cfc1 mtc1 ctc1 ] : cop1 & cop1code == {0 to 3} & funct == 0
   bc1x : cop1 & (cop1code == 4 | cop1code == 6)
   bc1f : bc1x & copbcode == 0
   bc1t : bc1x & copbcode == 1
```

Finally, here is the grouping according to assembly-language syntax.

192c ⟨*pattern bindings* 190a⟩+≡

```
patterns
  arith3. : add. | div. | mul. | sub.
  arith2. : abs. | mov. | neg.
  movec1  : mfc1 | mtc1 | cfc1 | ctc1
  c.cond  : c.f  | c.un   | c.eq  | c.ueq | c.olt | c.ult | c.ole | c.ule |
            c.sf | c.ngle | c.seq | c.ngl | c.lt  | c.nge | c.le  | c.ngt
  bc1     : bc1f | bc1t
  lsc1    : lwc1 | swc1
  convert : cvt.s | cvt.d | cvt.w
```

B.2 Computing follow sets

Patterns can be used in programs by writing "pattern case statements," which have the form

```
case value of
| pattern => action
  ⋮
else action
esac
```

Such statements may be imbedded in C or Modula-3 programs, with the *value*s and *action*s written in C or Modula-3. The *pattern transformer* reads such a program, together with a pattern specification, and transforms it into a valid C or Modula-3 program. The pattern case statement is replaced with nested if or case statements that determine which action is to be taken. The pattern transformer uses heuristics to keep the number of tests small; Baudinet and MacQueen (1985) describe similar heuristics.

The patterns simplify follow-set computation. For the MIPS, I define two extra patterns that describe straight-line code and conditional branches:

```
patterns
  inline      : load   | immedS  | immedU  | lui | shift | vshift |
                arith3 | arith2  | mfthilo | syscall | break |
                copls  | arith3. | arith2. | convert | movec1 | c.cond | lsc1
  conditional : branch1 | branch2 | bc1
```

Computing follow sets requires only five cases. To show the structure of the decoding, the details of the implementations of each case are elided:

```
case Memory.FetchAbs(m, pc, 'c', Memory.Type.I32).n of
| inline                    => ⟨straight-line code⟩
| beq & rt = 0 & rs = 0 => ⟨unconditional branch (16-bit displacement)⟩
| conditional               => ⟨conditional branch⟩
| jump                      => ⟨unconditional branch (26-bit displacement)⟩
| jumpr                     => ⟨computed branch⟩
else ⟨Fail with UnrecognizedInstruction⟩
esac
```