Extended version of a paper from the *14th International Conference on Compiler Construction*

Source-level Debugging for Multiple Languages With Modest Programming Effort

Sukyoung Ryu and Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
{syryu,nr}@eecs.harvard.edu

Abstract. We present techniques that enable source-level debugging for multiple languages at the cost of only modest programming effort. The key idea is to avoid letting debugging requirements constrain the internal structure of the compiler. Constraints are minimized primarily by hiding the source-language type system and target-machine representations from the debugger. This approach enables us to support a new language and compiler while reusing existing elements: a multi-language, multi-platform debugger; the compiler's implementation of source-language types and expressions; information already present in the compiler's private data structures; and our *compile-time support library*, which helps the compiler meet its obligations to the debugger without exposing language-dependent details. We evaluate our approach using two case studies: the production compiler `lcc` and an instructional compiler for MiniJava.

1 Introduction

Wouldn't it be nice if every high-level programming language came with a source-level debugger? Unfortunately, debugging requires a wealth of information that depends on both source language and target machine: what the source-level type system is, how source-level values are represented on the target machine, how such values should be displayed to the user, and so on. A typical debugger receives this information through an interface like `dbx` "stabs" or DWARF (Linton 1990; Unix Int'l 1993). It is bad enough that these interfaces are complex and difficult to use, but what is worse, they overconstrain the compiler: the compiler writer must shoehorn the source language into the debugger's type model, and the compiler writer's choices of representations are limited by the debugger's assumptions.[1] If the compiler writer does not account for the debugger's limitations from the beginning, the programming effort required to add debugging support can be onerous.

We have addressed this problem by changing the contract between the compiler and debugger. Our new contract enables us to reduce programming effort by reusing code, by reusing information already present in the compiler's private data structures, and by avoiding constraints on the compiler's representation choices and phase ordering.

[1] For example, many debuggers won't let a compiler put a record value in a register, even if the representation of the record fits in a machine word.

- We have implemented a debugger that can be reused almost *in toto* even with a language or compiler that it did not previously support. The only part that is not easily reused is the stack walker—our current stack walker supports only calling conventions that are similar to the standard C calling convention.
- Rather than implement an interpreter for source-language expressions, our debugger reuses the compiler's code to parse expressions, type-check them, and translate them to intermediate form. In addition to reducing programming effort, this tactic makes it easier to guarantee that the compiler and debugger implement the same semantics for the source language.
- Our debugger receives information from a compiler through a reusable *compile-time support library*. The library helps the compiler meet its obligations to the debugger while hiding language-dependent details. Our library does not force the compiler's private data structures to fit the debugger's model, does not require changing the timing or ordering of the compiler's phases, and does not artificially prolong the lifetimes of the compiler's internal data structures.

  To be used with a particular compiler, the library must provide an interface in the implementation language of that compiler. We have therefore designed the interface in two layers: an abstract layer that is independent of implementation language (Ryu and Ramsey 2004), and a concrete layer that contains instances for four implementation languages (C, Java, Standard ML, and Objective Caml). Each concrete instance is backed up by an implementation.

Our primary goal is to reduce programming effort, which is notoriously difficult to evaluate. As usual, we cannot afford a quantitative, comparative study of different implementation techniques. Instead, we rely on simple metrics and one basic principle.

- Once we have added debugging support to a compiler, we measure the resulting code:
  - How many of the compiler's modules were changed? How many new modules were added?
  - How big are the new modules? In changed modules, how many lines of code were added or changed?
  - How much code is duplicated? In particular, what functionality is implemented both by the compiler and by the debugger?

  There are other software metrics, but these suffice to show that our technique requires significantly less programming effort than standard techniques.
- Our basic principle is that the less the compiler writer is constrained, the smaller the programming effort will be. Relevant constraints include requiring that the compiler writer present certain information to the debugger; requiring that the information be presented in a certain order; requiring that the information be kept live for a certain time during compilation; and requiring that the compiler use only certain source-language types and representations. Minimizing these constraints has been the major idea behind the design of our support library.

Our work builds on earlier work with `ldb`, which used information hiding to make it easier to retarget a debugger (Ramsey and Hanson 1992; Ramsey 1993). That work applied only to a single language and compiler, and the contract between compiler and debugger was too complex and put too many constraints on the compiler writer. Our novel research contribution is a new contract, which reduces programming effort in two ways: it minimizes constraints on the compiler, and it removes the intellectual burden of organizing the compiler's information in the way the debugger wants it. Instead of falling on the compiler writer, this burden is carried by the support library.

Our new contract supports not only multiple machines, but also multiple source languages. This does not mean *every* language—we assume that a language can be executed by threads with stacks, has mutable state that can be examined, has a meaningful notion of breakpoint, and so on. While these assumptions apply to most languages, they may not apply to a lazy functional language, a logic language, or a constraint language, for example. Even so, `ldb` can easily be applied to an interesting class of multi-language programs. For example, Fig. 1 shows a debugger stack trace in which some frames are implemented in C and others in the instructional language MiniJava.

```
ldb Fib (stopped) > t
  0 <_print:2> (Mips/mjr.c:25,2)  void _print(char *s = (0x1000008c) " ")
* 1 <fib:51+0x24> (Fib.java:23,32)
         void fib(Fib this = {int buffer = 10,
                              int[] a = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
                                         0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
                             }, int n = 10)
  2 <main:3+0x18> (Fib.java:5,23)  void main(String[] argv = {})
  3 <mAiN:end+0x1c> (mininub.c:7,9)
         int mAiN(int argc = 2, char **argv = 0x7fff7b24,
                  char **envp = 0x7fff7b34)
```

Fig. 1. Example stack trace showing support for multiple languages. Frame 0 contains a C procedure that is part of the MiniJava run-time system; frames 1 and 2 contain MiniJava methods; and frame 3 contains startup code written in C.

2 Overview

Under a debugging contract, a compiler provides information about each program it compiles, and the debugger uses this information to give users a source-level view at run time. Information about a program is highly structured and may describe such elements as source-language types, variables, statements, functions, methods, and so on. We assume that in any given compiler, such elements have natural, native representations, which we call *language-level objects*. But a debugger that works with multiple languages and compilers must use a representation that is independent of any compiler; this representation is composed of elements we call *debug-level objects*. Under our contract, a compiler uses its language-level objects to create debug-level objects, which encapsulate the compiler's knowledge about the source program and its representation on the target machine. Both kinds of objects are described in Section 3.

A key property of debug-level objects is that language-dependent and machine-dependent information is hidden from the debugger. This information hiding leaves the compiler writer free to reflect the structure of the compiler's language-level objects directly in the structure of debug-level objects, reducing the effort required to create the debug-level objects. For example, if the compiler's natural representation of a record type contains a list of pointers to representations of the types of the record's fields, the compiler writer is free to create a debug-level representation of the record type that contains a similar list. But if the compiler's representation of a record type instead keeps the field information in an auxiliary symbol-table entry (Fraser and Hanson 1995, p 54), the compiler writer is free to reflect that representation instead.

To help the compiler create debug-level objects from language-level objects, we provide a compile-time support library. Each library function places a constraint on the lifetimes of language-level objects: all the language-level objects needed to create a debug-level object must be live at the same time. Section 4 describes these constraints and explains how we minimize them.

Section 5 presents two case studies and makes comparisons with `gdb`, and Section 6 discusses related work.

3 What a compiler must represent

Creating debug-level objects from language-level objects accounts for most of the programming effort of using `ldb`. To explain the effort, we discuss the language-level objects the compiler needs, the debug-level objects the compiler must create, and the associated programming effort. We also discuss expression evaluation, which not only requires its own effort but also affects the effort of creating debug-level objects.

To make examples concrete, we use procedure `fib`, shown in Fig. 2, which is written in MiniJava. Procedure `fib` computes and prints Fibonacci numbers. It is translated into the assembly file `Fib.s`, an excerpt of which is shown in Fig. 3.

3.1 Language-level objects

To create the debug-level objects that `ldb` requires, a compiler needs the following language-level objects.

- *Source-code locations*. The compiler must associate a source-code location with each instance of an interesting language construct, such as the declaration of a variable or the start of a statement. For example, in Fig. 2, the source-code location of variable `i` is line 7, column 11 of file `Fib.java`.
- *Variable placements*. The compiler must know where each live variable is placed at run time. Typically, a variable is placed either in a stack slot or in a machine register. For example, the compiler might place variable `i` in a stack slot addressed at offset -4 from register `$r2`. Placement may vary as the program counter changes.
- *Labels*. On request from the support library, the compiler must insert a label into assembly-language output. For example, the library may ask the compiler to insert a label to mark the start of a statement. In Fig. 3, label `$L.X6` is such a label.

```
1  void fib(int n) throws
2          java.io.IOException {
3      int[] a = new int[20];
4      if (n > 20) n = 20;
5      a[0] = 1;
6      a[1] = 1;
7      { int i = 1;
8        while (i<=n) {
9            i = i+1;
10           a[i] = a[i-1] + a[i-2];
11       }
12     }
13     { int j = 0;
14       while (j<n) {
15           this.printint(a[j]);
16           System.out.print(" ");
17           j = j+1;
18       }
19     System.out.println();
20     }
21 }
```

Fig. 2. `Fib.java`: MiniJava code.

```
        ⋮
    .text
    .globl  $L.Fib.fib
    .ent    $L.Fib.fib
$L.Fib.fib:
$L.Fib.fib_framesize=52
        ⋮
    addu $22, $0, $5
.set noreorder
$L.X6:
.set reorder
    addu $fp, $0, $21
    beq $fp, 0, _BADPTR1
$L1:
    add $8, $fp, 4
    addu $fp, $0, $8
.set noreorder
$L.X7:
.set reorder
    ori $8, $0, 20
        ⋮
```

Fig. 3. `Fib.s`: assembly code.

- *Types*. The compiler must know the source-language type of a variable or expression. (If the source language is dynamically typed, it suffices that every variable and expression have the same type, "value.") For example, `i`'s type is `int`.
- *Symbols*. If a named entity is to be visible at debug time, the compiler must describe it with a symbol. For example, if a user can ask the debugger for `i`'s value, the compiler must describe `i` with a symbol.
- *Stopping points*. The compiler must identify *stopping points*: source-code locations where execution of a program might stop at debug time. For example, MiniJava uses very fine-grained stopping points: not only before and between statements, but also before each expression, even nested expressions. A stopping point is also called a *locus of control*.
- *Environments*. The compiler must build an environment (symbol table) mapping names to symbols. In a language that allows local declarations, a single name can mean different things at different points in a program, so the compiler must be able to reconstruct a suitable environment at each stopping point. For example, the compiler must be able to reconstruct two different environments at the conditional expressions of the `while` statements in Fig. 2—variable `i` is visible from the conditional expression on line 8 but invisible from the conditional expression on line 14.

Because such language-level objects already exist in a typical compiler, even a student compiler (Aho, Sethi, and Ullman 1986; Appel 1998), providing them usually takes little programming effort. The two exceptions are source-code locations and stopping points.

Serious compilers track source-code locations, but a student's compiler may not. Adding source-code locations requires capturing them during lexical analysis and pushing them through the parser to an intermediate representation. If adding source locations takes too much effort, a compiler writer can expend less effort in return for less debugging functionality. For example, if a compiler provides the same source-code location for everything, the debugger will still be useful, but it will not support such operations as setting a breakpoint at a given source-code location.

Stopping points require more effort than source locations. First is the intellectual effort of determining what sorts of program points should be considered stopping points. This determination is language-specific but should not be difficult; common choices include statements, control-flow points, and declarations of named variables. (We recommend against another common choice: source lines.) Second is the programming effort, which we discuss in more detail in Section 3.2. While debugging without source locations is still useful, debugging without stopping points is not; without stopping points, the debugger cannot support breakpoints or single-stepping.

3.2 Debug-level objects to be created by the compiler

The language-level objects listed above are used to create debug-level objects. The structure of the language-level objects, especially types and symbols, necessarily reflects the structure of the language being compiled. But different languages have different structures. How, then, can we define one set of debug-level objects that supports multiple languages? The answer is that `ldb`'s debug-level objects make it *possible* to reflect linguistic structure at debug level, but they do not *require* any particular linguistic structure at debug level.

The interface to our support library is based on a hierarchy of types. The base type of a debug-level object is *info*. Some debug-level objects can be extended with key-value pairs (property lists) that hold information private to the compiler; such objects are *tables*. Particular instances of tables include *symbols* and *types*. In C, our interface defines these types as follows:

```
typedef struct ldbinfo   LdbInfo;   /* a debug-level object */
typedef struct ldbtable  LdbTable;  /* extensible with private data */
typedef struct ldbsymbol LdbSymbol;
typedef struct ldbtype   LdbType;
```

When our interface is instantiated in an object-oriented language, we define these types as classes, and we express relationships among them using subtyping and implicit subsumption. In C, however, we define an abstract type as an incomplete structure type, and we express relationships among types using explicit conversion functions.

```
LdbInfo  *ldbTable_asInfo  (LdbTable  *table);
LdbTable *ldbSymbol_asTable(LdbSymbol *symbol);
LdbTable *ldbType_asTable  (LdbType   *ty);
```

Private key-value pairs (properties) are added to a table using `ldbTable_put`.

```
void ldbTable_put(LdbTable *t, const char *key, LdbInfo *val);
```

With this hierarchy of types as background, we describe `ldb`'s requirements and the associated programming effort.

Simple objects: Source-code locations, labels, and variable placements To `ldb`, a source-code location is a triple containing a file name, line number, and column number; a label is a string; and a placement is a term in an algebra with labels, registers, and address arithmetic. Such objects are created using constructor functions like these:

```
LdbSrcLoc *ldbSrcLoc_make          (const char *file, int line, int col);
LdbLabel  *ldbLabel_makeInCodeSpace(const char *asmname);
LdbLabel  *ldbLabel_makeInDataSpace(const char *asmname);
LdbPlcmt  *ldbPlcmt_makeAtLabel    (LdbLabel   *label);
LdbPlcmt  *ldbPlcmt_makeAbsolute   (char space,                int offset);
LdbPlcmt  *ldbPlcmt_makeShifted    (            LdbPlcmt *loc, int offset);
LdbPlcmt  *ldbPlcmt_makeIndirect   (char space, LdbPlcmt *loc, int offset);
```

For example, `i`'s debug-level source location and placement are created as follows:

```
ldbSrcloc_make("Fib.java", 7, 11)
ldbPlcmt_makeIndirect('d', ldbPlcmt_makeAbsolute('r', 2), -4)
```

and `fib`'s assembly label is created by `ldbLabel_makeInCodeSpace("$L.Fib.fib")`.

Creating these objects typically requires modifying only a couple of existing modules. For example, `lcc` required 19 lines to create label names and 13 lines to translate its register names into `ldb`'s placement algebra.

Types `ldb` imposes no structure on types; it uses a debug-level type only to determine how a value is printed. To support printing, each debug-level type must include a procedure or method that prints values of that type. Writing these procedures requires significant effort. For each type constructor in the source language, the compiler writer must write a procedure that `ldb` can use to print a value whose type is formed using that type constructor. To write such a procedure, one must know how values are represented and how they should be printed. For printing, `ldb` provides a flexible prettyprinter, but for manipulating representations, `ldb` provides only basic machine-level primitives like load, store, and arithmetic. But because expressing source-level manipulations using machine-level primitives is what a compiler does, a compiler writer is well equipped to write printing procedures.

In what language are these printing procedures to be written? They can't be written in the source language of the target program, because `ldb` must work with multiple source languages. They could be written in machine language, but this is a bad idea; not only can machine code be tricky to load dynamically, but because `ldb` can debug over a network, machine code compiled for `ldb`'s target might not run on `ldb`'s machine. Ideally, printing procedures would be written in a simple, high-level scripting language. Today, popular languages like Perl, Python, and Scheme can be embedded in applications; there are even languages designed expressly to be embedded, like Lua and Tcl.

Any of these languages could work in `ldb`. But when the `ldb` project was started, these options did not exist. Instead, Ramsey and Hanson (1992, §5) chose to extend `ldb` with a new implementation of an existing language: PostScript. PostScript does have some advantages, but if we were to rebuild `ldb` from scratch, we would choose a language that is better known and more friendly to programmers.

The choice of language affects the effort required to write printing procedures. This effort, while modest, is not trivial; some examples may help you judge. A PostScript printing procedure receives three arguments on the stack: an "abstract memory," which represents the state of the target program; the location of the value to be printed; and a debug-level object containing the printing procedure. This last argument means that printing procedures are effectively equivalent to methods in an object-oriented setting.

The basic technique can be illustrated by a very simple example. The `lcc` compiler supports two different machine-level representations of floating-point numbers. In the debug-level object representing a floating-point type, `lcc` identifies the representation by using a private property `mtype`. This property, which is bound into the debug-level object using the support-library function `ldbTable_put`, is then used in the printing procedure:

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

The PostScript code "`/mtype get`" fetches the value of the `mtype` property, after which the PostScript stack holds exactly the arguments needed by `Memory.Fetch`, which leaves the floating-point number on the stack. As a primitive machine-level value, the floating-point number can then be printed using `ldb`'s primitive `Put`. Crucially, the very existence of the `mtype` field is hidden from the debugger.

The printing procedures that `lcc` uses for unsigned integers and for characters are similar. The printing procedure for signed integers is `PI`:

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

"`dup 4 1 roll`" saves a copy of the debug-level type before extracting `mtype` and fetching the value. In the debug-level type, the private key `bitsize` is associated with the number of bits in the integer, and it is used to sign-extend the integer before printing.

MiniJava supports only one basic type integer type, whose printing procedure is

```
/INT { pop Memory.Type.I32 Memory.Fetch 32 SignExtend Put } def
```

Because every MiniJava value fits in one word, this procedure is simpler than `lcc`'s printing procedure for integers.

Structured types, like arrays, require more complicated printing procedures. To help print an array, `lcc` includes the following private values in the debug-level object that represents the type of the array:

`type`	The type of an element of the array (as a debug-level object)
`size`	Number of bytes in the array

Using those private fields, `lcc`'s printing procedure for arrays is

```
/ARRAY {
  16 dict begin
    /&type exch def                          % name the type
    /&loc  exch def                          % name the location
    /&m    exch def                          % name the memory
    /&arraysize &type     /size get def      % name the array's size
    /&elemtype  &type     /type get def      % name the element type
    /&elemsize  &elemtype /size get def      % name the size of one element
    /&limit $ArrayLimit &elemsize mul def    % find the end of the array
    ({) Put 0 Begin
    0 &elemsize &arraysize 1 sub             % loop from 0 to &arraysize-1
                                             %      by &elemsize
    { dup 0 ne { (, ) Put 0 Break } if
      dup &limit ge { (...) Put pop exit } if
      &m &loc 3 -1 roll Shifted &elemtype print   % print an element
    } for
    (}) Put End
    end
} bind def
```

The print procedure first puts relevant information in local variables, whose names begin with ampersands. The code then prints an opening brace and loops through the offsets of the array elements. At each iteration, the `Shifted` function adds the offset to the location of the array, and `print` prints the element. Each element except the first is preceded by a comma and a potential line break. If the number of elements exceeds an adjustable limit, an ellipsis is printed and the loop terminates. Finally, a closing brace is printed.

MiniJava uses a different set of private keys for an array type:

`elemty` The type of an element of the array

In MiniJava, unlike in C, the size of an array is not part of the array's type; instead it is stored in memory at offset -4 from the start of the array.

MiniJava's printing procedure for arrays is

```
/ARRAY {
  16 dict begin
    /&type exch def
    /&loc  exch def
    /&m    exch def
    /&newloc &m &loc Memory.Type.I32 Memory.Fetch 'd' Absolute def
    /&length &m &newloc -4 Shifted Memory.Type.I32 Memory.Fetch def
    /&elemtype &type /elemty get def
    ({) Put 0 Begin
    0 1 &length 1 sub
    { /&i exch def
      &i 0 ne { (, ) Put 0 Break } if
      &i $ArrayLimit ge { (...) Put pop exit } if
```

```
    &m &newloc &i 4 mul Shifted &elemtype print
  } for
  (}) Put End
  end
} bind def
```

After accounting for differences in representation, this procedure is quite similar to `lcc`'s procedure. But it is precisely the ability to accomodate small differences that makes it easier to work with `ldb` than with a large, complex type system that (for example) prescribes one and only one representation for arrays.

The examples above may be slightly intimidating, but once one masters some rudimentary PostScript, writing printing procedures is straightforward. Moreover, our case studies showed that printing procedures for `lcc` and MiniJava are very similar. We expect, therefore, that one could use existing printing procedures as guides to implement new ones. And our case-study compilers do not require many printing procedures. For example, the `lcc` front end recognizes 14 language-level type constructors, which it prints using 13 printing procedures. (Some type constructors share a printing procedure, and some types, notably `char *`, require specialized printing procedures.)

One creates a printing procedure by supplying source code to `ldbPsProc_make`. One can then use the procedure to create a debug-level type:

```
LdbPsProc *ldbPsProc_make(const char *code);
LdbType   *ldbType_make  (const char *decl, LdbPsProc *printer);
```

The type also takes a `decl` field, which is a `printf` format string that enables the debugger to print a declaration of a variable of the type. For example, `i`'s debug-level type is created by `ldbType_make("int %s", ldbPsProc_make("INT"))`.

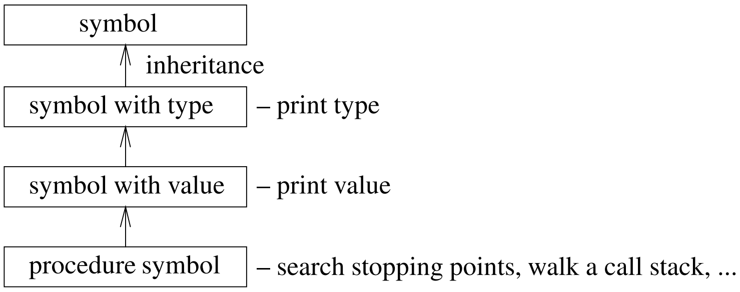

Fig. 4. Symbols and the operations they support

Symbols Debug-level symbols come in four flavors: bare, with type, with type and value, and procedure. In Fig. 4, the least featureful flavor is at the top and the most featureful at the bottom. Each flavor is also a *table* and so can be extended with private properties. A bare symbol has a name and a source-code location but no other information; such a symbol is not directly useful to `ldb`, but it may help a compiler writer embed the compiler's private data structures into debug-level objects. A symbol with only a type can have its type printed; such a symbol might represent a language-level type. A symbol with a type and value can have both its type and its value printed; such a symbol might represent a language-level variable or constant. A procedure symbol can support many debug-time operations, including enumerating formal parameters and local variables, searching stopping points, and walking a call stack. To support these operations, a procedure symbol must provide the name of its return type, an environment that includes its arguments, its stack-frame size, information about callee-saved registers, its assembly-output label, and a list of stopping points.

Because the linguistic structure of symbols is not part of the contract between `ldb` and the compiler, this contract supports multiple languages and compilers. The compiler writer must embed the compiler's symbols into `ldb`'s symbols, but because each flavor can be extended with private properties, this does not require too much programming effort. For example, `lcc` maintains five kinds of symbols: types, constants, variables, procedures, and `typesyms`. (The `typesym` is part of `lcc`'s private representation of a C-language type.) `lcc`'s types, constants, variables, and procedures are embedded in symbols with types, symbols with values, symbols with values, and procedure symbols, respectively. Because `ldb` does not use `typesyms` directly, `typesyms` are embedded in bare symbols.

Stopping points At debug time, stopping points are used not only to plant breakpoints by source-code location but also to identify the source-code location nearest the point of a program fault. These operations require the debugger to map between object-code locations and source-code locations. The map is defined by the set of all stopping points in the program, so `ldb` requires the compiler to associate each stopping point with both a source-code location and an object-code location. (The object-code location is represented by an assembly-language label like `$L.X6` in Fig. 3.) At a stopping point, the debugger must be able to look up symbols, so `ldb` requires the compiler to associate each stopping point with an environment. Therefore, a debug-level stopping point, which is also called a "locus of control," is created with three arguments:

```
LdbLocus *ldbLocus_make(LdbSrcLoc *src, LdbLabel *label, LdbEnv *env);
```

As described in Section 3.1, the primary effort required to support stopping points is to create an explicit, language-level representation in the compiler. Given a suitable language-level representation, creating the corresponding debug-level object is straightforward. Section 5.2 discusses how we modified the `lcc` and MiniJava compilers to create stopping points.

Environments At debug time, looking up symbols by name requires an environment. Because a compiler already maintains its own language-level environments, creating debug-level environments should not require much programming effort. Some extra effort may be required to propagate environments through the intermediate representation so they can be associated with stopping points.

Compilation unit The `ldb` support library maintains an abstraction that represents the compiler's knowledge about the entire compilation unit. This debug-level compilation unit is created incrementally: every time the compiler processes a top-level symbol, it should announce the symbol to the compilation unit. In the C interface, for example, `ldbCompUnit_addProc` adds a procedure to the compilation unit, which is an implicit part of the library's state:

```
void ldbCompUnit_addProc(LdbProc *proc);
```

Similar functions add private and exported symbols. Because a symbol can be announced from wherever it is created, little programming effort is required.

3.3 Expression evaluation

Unlike other debuggers, which require that the source language be reimplemented inside the debugger, `ldb` evaluates expressions by reusing a key component of the compiler: the translation from source language to low-level intermediate code. After being wrapped in a thin layer that communicates with the debugger, this translation component acts as an *expression-evaluation server*. The server communicates with `ldb` over a TCP connection: `ldb` sends ASCII to the server, and the server replies with PostScript code that `ldb` interprets. To evaluate an expression, `ldb` sends the text of the expression to the server, which parses the expression, type-checks it, and replies with a PostScript procedure followed by code that, when interpreted, has the effect of evaluating the expression (Ramsey and Hanson 1992).

Building an expression-evaluation server may require significant intellectual effort as well as programming effort: one must define what it means to evaluate an expression at a stopping point. For an explicitly typed language such as Java or C, this task is easy; one can simply reconstruct the original environment in which the stopping point occurs. But for an implicitly typed language such as Haskell or ML, an expression that is evaluated at debug time cannot participate in type inference in the same way as an expression that is part of the original program. For example, Hindley-Milner type variables cannot be unified with known types but must be treated as abstract types. For such languages, the semantics of debug-time evaluation remains a topic for future work.

Even given a semantics, implementing expression evaluation requires significant programming effort. The process is described in detail by Ramsey (1993, Chapter 5), but we summarize here.

- The expression server must reconstruct the language-level objects that represent the context at a stopping point. The private contents of these objects can be stored in property lists on `ldb`'s debug-level objects. An additional property, "serialize," should be PostScript code that, when interpreted, sends the private properties to the expression server. To get its private data, the server asks `ldb` to execute "serialize." A server should reconstruct contexts incrementally. For example, `lcc`'s server starts working with an empty symbol table. When it looks up an identifier `a` that is not present, it asks `ldb` for the debug-level object that represents symbol `a`, and it uses the properties on that object to reconstruct a suitable language-level symbol for `a`.
- Once the context has been reconstructed and an expression translated to intermediate code, the server must turn this code into PostScript. As a stack-based language, PostScript is an ideal target for such translation, so there is not much intellectual effort involved. But the programming effort is proportional to the number of different kinds of nodes in the compiler's intermediate code, which can be considerable.

– If the expressions to be evaluated include procedure calls, the target-language run-time system may have to be modified to be able to execute a procedure call on behalf of `ldb`. Depending on the complexity of the calling convention, this feature can be quite difficult to implement.
– If the compiler supports multiple target machines, the expression server must be specialized to the requirements of the target being debugged. For example, the expression server must know the sizes and alignments of basic data types. The compiler writer may create a specialized expression server for each target, or better, create a single expression server that is specialized at startup time.
  The server for `lcc` is specialized at startup time. Adding this capability took some effort, but because startup-time specialization offers major benefits for configuration management, the production version of `lcc` has adopted it, so it is now "free."

While implementing an expression-evaluation server requires significant programming effort, this effort is not absolutely required—because a debugger is useful even without an expression-evaluation server, we can again trade programming effort for debugging functionality. In particular, if no expression-evaluation server is provided, `ldb` uses a "default" evaluator that can print the values of variables only.

4 When representations must be available

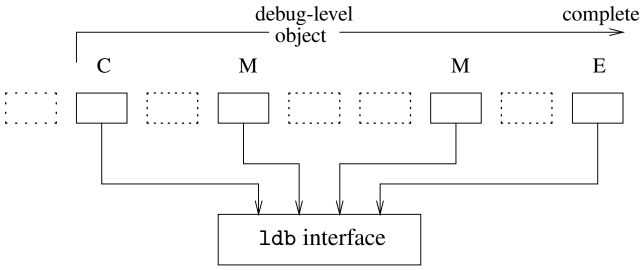

Fig. 5. Incremental construction of a debug-level object

Section 3 explains what language-level objects a compiler must provide in order to create debug-level objects for `ldb`. But when must these objects be provided? One strategy is to accumulate *all* the objects, and once compilation is over, emit them (Ramsey 1993, Chapter 4). Although this strategy has the merit of simplicity, it prolongs the lifetimes of the compiler's language-level objects, and it can complicate memory management and slow the compiler. A better strategy is as follows:

– *Create* debug-level objects as soon as is convenient, possibly leaving out some parts. For example, a procedure's object might be created without stopping points.
– Incrementally *mutate* debug-level objects to accumulate missing parts.
– When all information has been accumulated, *externalize* the object by writing it to assembly-language output.

This strategy, as applied to a single object, is depicted in Fig. 5. Each small box denotes a phase of the compiler. The phase labelled “C” creates the object; phases labelled “M” mutate the object, the phase labelled “E” externalizes the object, and dotted phases ignore the object. The lifetime of the object runs from “C” to “E;” after being externalized, the object can be discarded.

The create-mutate-externalize strategy is built into the design of our support library. The library provides the create and mutate operations for each type of debug-level object. For example, a debug-level procedure is created without stopping points; when a stopping point becomes available, the compiler mutates the procedure by calling `ldbProc_addLocus`.

```
LdbProc *ldbProc_make    (const char *name, LdbSrcLoc *src);
void     ldbProc_addLocus(LdbProc    *proc, LdbLocus  *locus);
```

In the library interface, an externalize operation is exported only for a compilation unit; other objects are externalized by the library's implementation.

The primary benefit of our approach is that the compiler writer need not worry about how long objects should live or when objects should be written to disk; these tasks are handled by the support library. A secondary benefit is that the support library becomes free to change and experiment with the external representation of debug-level objects, perhaps to improve performance. For example, it is possible to externalize individual objects incrementally, by writing create and mutate operations to a log. Experimentation with these possibilities is a topic for future work.

5 Results

To assess the programming effort required by our approach, we undertook case studies with the `lcc` (Fraser and Hanson 1995) and MiniJava (Appel and Palsberg 2002; Hosking 2003) compilers. We explain how programming effort is decomposed, present internal metrics for the modifications done to each compiler, discuss the effort of creating the compile-time support library, and compare with `gdb`.

5.1 Decomposition of programming effort

We distinguish two kinds of programming effort: modifying the compiler's existing *phase modules* and adding new *utility modules*. A phase module implements a phase of the compiler; a utility module is used by phase modules or by the support library. Utility modules have well-defined interfaces and do not depend on other parts of the code, so it is easy to add them. To modify an existing phase module requires more effort: the compiler writer must understand how to do the modification safely. We therefore reduce programming effort by putting most new code, especially most of the intellectual work of creating debug-level objects, into utility modules. This organization requires less programming effort than would be required to do the same work in phase modules.

		`lcc`			MiniJava		
		core	driver & back ends	total	core	driver & back ends	total
original compiler	total modules	36	9	45	132	6	138
	total source lines	13,730	4,839	18,569	5,673	1,472	7,145
	Driver source		268			121	
	Alpha source		1,192				
	MIPS source		1,129			1,351	
	SPARC source		1,163				
	x86-linux source		1,087				
effort to add source locations & stopping points	utility modules added	(none)			(none)		
	phase modules changed				8	0	8
	• lines added				131	0	131
	• lines changed				53	0	53
effort to support MIPS hardware	utility modules added	(none)			0	1	1
	• lines therein				0	49	49
	phase modules changed				4	3	7
	• lines added				45	429	474
	• lines changed				9	1	10
effort to add debugging support	utility modules added	6	0	6	2	0	2
	• lines therein	990	0	990	343	0	343
	phase modules changed	4	4	8	29	2	31
	• lines added	18	43	61	340	46	386
	• lines changed	0	4	4	57	9	66
effort to add an expression-evaluation server	utility modules added	0	8	8	(none)		
	• lines therein	0	1,172	1,172			
	phase modules changed	24	0	24			
	• lines added	730	0	730			
	• lines changed	105	0	105			
total source lines added or changed		**1,843**	**1,219**	**3,062**	**978**	**534**	**1,512**

Fig. 6. Case studies: `lcc` and MiniJava

	C/C++	Java	Fortran	Pascal	Modula-2
lang`-exp.y`	1,715	1,462	1,175	1,485	1,094
lang`-lang.c`	522	1,097	957	465	468
lang`-lang.h`	84	66	98	75	31
lang`-typeprint.c`	1,154	343	435	858	41
lang`-valprint.c`	573	527	739	1,115	39
total source lines	**4,048**	**3,495**	**3,404**	**3,998**	**1,673**

Fig. 7. Lines of code for `gdb`'s language support

5.2 Internal metrics

Measurements of our two case-study compilers are summarized in Fig. 6. The block at the top of Fig. 6 shows the sizes of the original compilers; back-end code is split by target machine. The next four blocks summarize four kinds of modifications:

- We modified MiniJava to propagate source-code locations into intermediate code and to define stopping points.
- We modified MiniJava to generate code that could run on MIPS hardware, not only on the SPIM simulator (Larus 2003).
- We modified both compilers to add debugging support.
- We implemented an expression-evaluation server for `lcc`.

Each kind of modification is described in more detail below.

The original MiniJava compiler provided no source-code locations for symbols and no stopping points. We modified MiniJava's parser to capture each symbol's source-code location and to propagate the locations into MiniJava's intermediate representation. We also modified MiniJava to include a stopping point before each expression, including nested expressions, as well as before each statement and at the end of each block. Adding source-code locations changed only 17 lines; adding stopping points added 131 lines and changed 36 lines.

The original MiniJava compiler generated code that ran only on the SPIM simulator (Larus 2003). Original MiniJava also assumed that the `main` method is passed an empty array, which is true on SPIM but not on real hardware. Since our debugger works on real hardware, we modified MiniJava to produce suitable assembly code. In particular, we modified 6 modules to emit MIPS instructions, and we added a new module to the MiniJava run-time system. Treating the command-line arguments consistently with SPIM required 4 modules to be modified. In total, we added 1 C module and modified 7 Java modules for MIPS support.

The bulk of our effort was invested in true debugging support, which we did for both MiniJava and `lcc`. Because `lcc` has a well-defined internal interface for emitting debugging information, most of our effort was in writing a new implementation of that interface. We added 6 modules to `lcc`: 1 big module emits debugging information, and the other 5 implement utility functions. For MiniJava, we added 2 modules to the core of MiniJava: 1 for variable placements and 1 for printing procedures. We had to change 31 modules, but because MiniJava uses visitor patterns, this number is deceptively large: of the 31 changed modules, 22 collaborate to define MiniJava's intermediate form, defining one type of node per module. Leaving aside these modules, we changed roughly the same fraction of modules as in `lcc`: about 10%.

We implemented an expression-evaluation server only for `lcc`. To implement communication between `ldb` and the server, we modified 105 lines and added 1,172 lines. Of the 8 modules added, 1 module of 239 lines is a driver, and 1 module of 617 lines implements a new back end that generates PostScript. The remaining 6 modules, totalling 316 lines, enable the expression server to be specialized at startup time; there is one specialization module for each of 6 different architectures.

We also consider duplication of effort. Only name resolution is implemented both by `ldb` and by our case-study compilers. The code is 316 lines for `lcc` and 77 lines for

MiniJava. Because the compiler and debugger both need name resolution, and because they can be implemented in different languages, we see no way to avoid this duplication.

5.3 The compile-time support library

Our case studies do not include the effort required to create `ldb`'s support library. We have invested significant effort in developing four implementations of this library: one in each of C, Java, Standard ML, and Objective Caml. If a compiler is written in one of these languages, the appropriate library can be reused with no additional programming effort. But if a compiler is written in another language, the library interface must be instantiated for the new language, and the new interface must be implemented.

To instantiate the interface requires choosing suitable idioms in the new language. But our four existing instances already embody idioms from a range of paradigms: Java is object-oriented, Standard ML is functional, Objective Caml is both, and C is neither. We expect, therefore, that one of the existing instances would be a good guide to creating a new one. Furthermore, the library itself is not large; for example, the C interface is 1,154 lines, of which only 366 are non-blank, non-comment lines. The corresponding implementation is 1,001 lines. Given a new programming language, we could probably instantiate the library interface and implementation in less than a week.

5.4 Comparisons with `gdb`

Our case studies look at programming effort for individual compilers. Here, we compare our method with `gdb`. According to our basic principle—the less the compiler writer is constrained, the smaller the programming effort will be—we compare how our method and `gdb` constrain the compiler writer.

The compiler writer must present certain information to the debugger. In Section 3, we describe what language-level objects the compiler must provide. We didn't need to add any language-level objects to `lcc`; to MiniJava, we added 131 lines and modified 53 lines for source-code locations of symbols and for stopping points. These changes affected only small, isolated parts of the compiler.

`gdb` does not describe its requirements clearly. Gilmore (2000, Chapter 7) tells a compiler writer to add a new source language to `gdb` by providing 5 files. Fig. 7 shows the sizes of the files for the source languages supported by `gdb-5.1.1`: C/C++, Java, Fortran, Pascal, and Modula-2. Language support is three or four thousand lines of C code, except for Modula-2. The support for Modula-2 is much smaller because it reuses the modules for printing C types and values. Comments in the source code indicate that this reuse is a stopgap measure, and `gdb`'s implementors intend eventually to implement correct printing support for Modula-2. By contrast, `ldb`'s language support is three thousand lines of C code for `lcc`, including its expression-evaluation server, and half that for MiniJava, which lacks an expression server. According to Gilmore, a compiler writer who wants to add a new language to `gdb` must understand at least 3 header files and 5 source files, which are 11,521 lines altogether, of which 7,727 lines are non-blank, non-comment lines. Our support library is far smaller and simpler.

The information must be presented in a certain order and be kept live for a certain time during compilation. For each symbol, `gdb` represents the symbol information as a "stabstring" which is the symbol's name appended by the encoding of type information. While the stabstring requires a compiler to generate a symbol's information all at once, our support library can accumulate information incrementally, which gives the compiler the freedom to generate different parts of the information in any order. Moreover, because `ldb` uses a native-language interface, not strings, our support library can guarantee the well-formedness of the generated information.

The less the source language is constrained, the smaller the programming effort. To work with `gdb`, a compiler must use only types that `gdb` understands, and it must use `gdb`'s representation. By contrast, our techniques hide the source-language type system and data representations, so the compiler writer never has to change them in order to suit the debugger—this is a whole category of programming effort we guarantee to eliminate. For example, we kept `lcc`'s and MiniJava's data representations unchanged.

The more the compiler's code is reused, the smaller the programming effort. `gdb`'s code for expression evaluation duplicates substantial functionality that is already present in the compiler, including parsing, type checking, and generation of intermediate code. Because the code in `gdb` must respect `gdb`'s naming conventions and other internal constraints, it is not possible simply to reuse the corresponding code in the compiler, even if the compiler happens to be implemented in C. By contrast, as described in Section 3.3, our method reuses the compiler's code.

The less the debugger's code is revealed, the smaller the programming effort. The compiler writer interacts with `ldb` only through the compiler-support interface, which hides the details of the debugger. By contrast, `gdb` requires the compiler writer read and understand large chunks of C code.

6 Related work

There are two fundamental approaches to source-level debugging of compiled code. To support the *reverse engineering* approach, a compiler generates code much as it normally would, and it emits additional information that enables the debugger to analyze the object code and report information at the source level. To support the *instrumentation* approach, a compiler or other tool modifies the program before code generation; for example, a compiler might insert a conditional branch at every stopping point. Debuggers that use reverse engineering include `ldb`, `gdb`, and `dbx`; debuggers that use instrumentation include `smld` (Tolmach and Appel 1990; Tolmach 1992) and `cdb` (Hanson and Raghavachari 1996).

Instrumentation can support debugging with modest programming effort. Instrumentation can also support advanced debugging features such as time travel, which is more difficult to support using reverse engineering (Feldman and Brown 1988). Unfortunately, the convenience comes at a substantial cost in performance: code instrumented for debugging typically runs 3–4 times as long as uninstrumented code. We therefore limit our attention to debuggers that use reverse engineering.

The standard approach to reverse engineering, which is exemplified by `dbx`, `gdb`, and DWARF, is for the debugger to provide a *union model* of all languages it supports. Because the union model provides the interface by which the compiler tells the debugger about the types of variables, it must include every type constructor used in any language the debugger might support.

There are two difficulties with a union model:

- The compiler and debugger must agree on a representation for each type. Because the representation may be machine-dependent, a different agreement may be needed for each target machine. Typically, the compiler writer cannot choose representations for high-level values such as records and strings; the compiler must use the representations that are assumed by the debugger.
- A union model might not include the types needed by a new language. Forcing a new language to fit an old union model may require substantial effort, and success is not guaranteed, as is shown by experience with SRC Modula-3.[2]

A union model exposes the details of source-language types and target-machine representations. By hiding these details, we leave the compiler writer free to change them, so no programming effort is expended forcing the compiler to be compatible with an unsuitable union model.

Like `ldb`, the Acid debugger avoids a union model and instead prints source values by using functions written in an internal programming language (Winterbottom 1994). As with `ldb`, these functions must be emitted by the compiler. But Acid takes its internal language much further: the language is the debugger's only user interface. For example, commands such as breakpoints and stepping, which are primitives in other debuggers, are instead implemented in the Acid language, where they can be adjusted to suit the needs of debugging particular target programs. The Acid language can also be turned to other uses, including fault detection and test-coverage analysis. It is difficult to evaluate the programming effort required for a compiler to work with Acid, but it looks similar to the effort required to work with early versions of `ldb`: it appears to be up to the compiler writer, without assistance, to emit information in the form that the debugger expects.

A great deal of related work has been invested in debugging optimized code. To debug optimized code, the debugger needs to know the relation between optimized code and source code. In particular, when execution is suspended, the debugger must find a way to explain the actual state of the machine, even if that state is not consistent with a sequential execution of the source program. This may happen if, for example, the optimizer has changed the order of execution, eliminated dead assignments, eliminated induction variables from loops, and so on. Even for a fairly simple optimization, building a debugger that is capable of finding such explanations requires substantial intellectual and programming effort. This problem has engendered a large body of literature, which falls into two broad camps. Hennessy (1982) exemplifies the camp that tries to "undo" optimizations so that the debugger can present an explanation that makes it appear as if the program had never been optimized. Brooks, Hansen, and Simmons (1992), Tice and

[2] In Modula-3 folklore, `gdb` support was coming Real Soon Now from the early 1990s on. Partial support arrived eventually, but it was too late to help prevent Modula-3 from falling into disuse.

Graham (1998), and Jaramillo, Gupta, and Soffa (1999, 2000) exemplify the camp that tries to explain how the optimized code is executed as it is. In either camp, building a debugger requires lots of compiler support and also a deep understanding of optimization. We believe that debugging optimized code is orthogonal to the problem of building a debugger that supports multiple languages, compilers, and machines.

7 Conclusion

We have presented a new kind of contract between a compiler and debugger. The key ideas are to distinguish language-level objects from debug-level objects; to build debug-level objects incrementally; to hide the memory management and external representation of debug-level objects in a reusable support library; and to hide language-dependent, machine-dependent information behind the methods of the debug-level objects. The contract supports multiple programming languages and target machines, and it helps a compiler writer add debugging support while expending only modest programming effort.

Acknowledgements

Anonymous referees provided helpful comments on an earlier version of this paper.

This work has been funded by an Alfred P. Sloan Research Fellowship and by National Science Foundation grant EIA-0096091.

Bibliography

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

Andrew W. Appel. 1998. *Modern Compiler Implementation*. Cambridge University Press. Available in three editions: C, Java, and ML.

Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition.

Gary Brooks, Gilbert J. Hansen, and Steve Simmons. 1992 (July). A new approach to debugging optimized code. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(7):1–11.

Stuart I. Feldman 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.

Christopher W. Fraser and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley.

John Gilmore. 2000. GDB internals—a guide to the internals of the GNU debugger. Found in the `doc` directory of `gdb` distribution version 5.1.1.

David R. Hanson and Mukund Raghavachari. 1996. A machine-independent debugger. *Software–Practice and Experience*, 26(11):1277–1299.

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

Antony Hosking. 2003. The MiniJava compiler. Provided by the author, whose email address is `hosking@acm.org`.

Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 1999 (March). Comparison checking: an approach to avoid debugging of optimized code. In *Proceedings of the 7th European Software Engineering Conference (ESEC) held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, volume 1687 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag.

Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 2000 (July). FULLDOC: A full reporting debugger for optimized code. In *Proceedings of the 7th International Symposium on Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 240–259. Springer-Verlag.

James Larus. 2003. SPIM: A MIPS R2000/R3000 simulator. `http://www.cs.wisc.edu/~larus/spim.html`.

Mark A. Linton. 1990 (June). The evolution of Dbx. In *Proceedings of the Summer USENIX Conference*, pages 211–220.

Norman Ramsey. 1993 (January). *A Retargetable Debugger*. PhD thesis, Princeton University. Also technical report CS-TR-403-92.

Norman Ramsey 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.

Sukyoung Ryu and Norman Ramsey. 2004. The `ldb` interface. Technical Report TR-23-04, Division of Engineering and Applied Sciences, Harvard University.

Caroline Tice and Susan L. Graham. 1998 (July). OPTVIEW: A new approach for examining optimized code. *Proceedings of the 1998 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering,* in *SIGPLAN Notices*, 33(7):19–26.

Andrew P. Tolmach. 1992 (October). *Debugging Standard ML*. PhD thesis, Princeton University. Also technical report CS-TR-378-92.

Andrew P. Tolmach and Andrew W. Appel. 1990 (June). Debugging Standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 1–12.

Unix Int'l. 1993 (July). *DWARF Debugging Information Format*. Unix International, Parsippany, NJ.

Philip Winterbottom. 1994. Acid: A debugger based on a language. In *Proceedings of the Winter 1994 USENIX Conference*, pages 211–222.