A Symbolic Portable Debugger for Compilers that Generate C Code

JOSÉ M. PRIETO, JOSÉ L. ARJONA, RAFAEL CORCHUELO, MIGUEL TORO, AND DAVID RUIZ
Departamento de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadística, Universidad de Sevilla
Avenida de la Reina Mercedes s/n, 41.012, Sevilla
ESPAÑA — SPAIN

Abstract: Most compilers translate high-level programming languages into machine code, but, if we are interested in portability, this might not be a good idea because machine code is not portable among different platforms. This is the reason why many compilers do not produce machine code as output, but ANSI C code. The problem is that the code these compilers produce is not debugable because it does not include any references to the symbols appearing in the original program.

We have investigated some techniques that allow us to bridge this gap. As a result, we have produced a library compilers that generate C code can easily incorporate in order to generate self–debugging programs. This paper aims to explain its main features and also reports some experimental results that show that it performs quite well. *IMACS/IEEE CSCC'99 Proceedings*, Pages:3501-3506

Key words: symbolic debuggers, compilers that generate C code as output, portability, GDB.

1 Introduction

Compilers usually generate machine code as output, but, in general, this is not a good idea if we are interested in portability because if we want to port our compiler to other platforms, we need to spend a lot of time to rewrite our code generator. In addition, the effort to maintain different versions of our compiler increases. On the other hand, C has become a *de facto* standard available in most platforms. It is highly portable because most compilers are ANSI compliant, and there are commercial or free compilers that generate code so optimised that its performance is sometimes comparable to the performance of a machine code program written by a human programmer. This is the reason why many compilers do not produce machine code as output, but ANSI C code. Lately, these compilers are sprouting out at an increasing pace. A set of good examples include ISE Eiffel [8], Small Eiffel [9], SR [6], Ingres 4GL [10] or PL/SQL [7], and the list increases every day because this idea is very attractive.

Nevertheless, programmers rarely write correct programs, and they usually need to use debuggers to find out the reason why their programs do not work properly under certain circumstances. At present, there are some very good commercial debuggers, as well as wonderful free debuggers. GDB [2], the debugger by GNU, stands out because it is available in most platforms, and it incorporates many advanced debugging facilities. Unfortunately, none of these debuggers is suitable to debug the C code generated by the compilers we have just mentioned. The reason is that if we used such a debugger on this code, we would trace a C program instead of the original program written in Eiffel, Ingres 4GL or PL/SQL.

We have investigated some techniques that allow us to bridge this gap, and the result we have produced is a library compilers that generate C code can easily incorporate in order to generate self–debugging programs. This way a compiler can produce portable ANSI C code as output, which can be fully optimised, while keeping it debugable. This is a nice feature of our debugger others do not support.

This paper is organised as follows: section 2

gives some general information about debugger construction, and describes our library; section 3 compares our solution with GNU GDB, and section 4 reports some experimental results; finally, section 5 shows our conclusions and the work we are planning on doing.

2 Making self–debugging programs

We assume that we are interested in a language called "X", and that we have a compiler that translates X code into ANSI C code. Thus, after compiling an X program, we need to use a C compiler in order to translate generated C code into a set of object modules that a linker can combine with external libraries to produce an executable program. Our idea consists of making this executable program self–debugging, i.e., instead of implementing the debugger as a separate tool able to trace another program, we have decided to incorporate the debugger into the generated C code so that the final executable can debug itself. This makes our solution very portable because we do not need to rely on specific, platform–dependent ways to controlling the execution of another program. On the contrary, the executable is able to debug itself by inserting adequate calls to our debugging library in the C code our X compiler produces.

Our library provides routines that can trace a program step by step, inspect its variables, set conditional breakpoints, and so on, but it still needs some information about the symbols appearing in an X program. For example, it needs to know what variables, routines or types the user has declared. The best way to producing this information consists of dumping the symbol tables our X compiler uses on a file composed of several "stabs" [1]. Stabs are way to storing symbol tables that was invented by Peter Kessler, who worked in the University of California (Berkeley). We use a simplified version that a compiler can easily manage and store in a text file. Our debugging library has access to this file so that it can reconstruct symbol tables at run time, thus having easy access to all the information the compiler had about each symbol.

2.1 Stab files

A stab file is a structured text file where a compiler can store stabs of the following form:

```
name "class description"
```

Here, `name` is the name a programmer has given to an object in a program, `class` is a character that describes what it is (a variable, a routine, a constant, a type, and so forth), and `description` is a string that describes its features (for example the type of a variable, the number of dimensions of an array, the fields a record has, and so forth).

Stab files are divided into several sections, each of them describing the set of symbols that appear in a syntactic scope. Languages usually provide a way to modularising programs. For instance, typical imperative languages provide subroutines, object–oriented languages provide classes, and other languages such as Modula provide modules that encapsulate implementation details. Each module can be viewed as a different scope with its own naming space, and the structure of a stab file closely resembles this modules, the difference being that scopes cannot be nested in a stab file. Consider, for example, the following program written in Pascal:

```
 1: program Example(Output);
 2: var n: Integer;
 3: function Fact(num: Integer): Integer;
 4: begin
 5:   if num = 0 then
 6:     Fact := 1
 7:   else
 8:     Fact := num * Fact(num - 1)
 9: end;
10: begin
11:   n := 5;
12:   WriteLn(n, '! =', Fact(n))
13: end.
```

There are two different scopes in this simple program: the one associated with the main program, and the one associated with routine `Fact`. In each scope, there are several user–defined objects, such as variable `n` or parameter `num`, and several predefined objects, such as routine `WriteLn` or variable `Output`. For the sake of simplicity, we assume that this variable is of a predefined type called `Text` whose structure follows:

```
String = array[1..80] of Char;
Text = record
  handle: Integer; name: String
end
```

The stab file associated with this example follows:

```
[Example]
WriteLn     "Rv"
Output      "VText"
Text        "Ur:handle,i:name,String;"
String      "Uai;1,80;c"
n           "Vi"
Fact        "Ri"
[Fact]
num         "Pi"
```

Scope names are written in square brackets, and they are followed by the stabs corresponding to each symbol appearing in it. In this example, the first stab corresponds to the predefined routine `WriteLn`: its class is `R`, which indicates it is a routine, and its definition `v`, which indicates that it does not return any value (`void`). Its parameters, their definitions and its local data do not appear in this stab file because it is a predefined routine and debuggers do not usually allow programmers to trace such routines. In any case, the compiler might generate appropriate stabs if necessary. `Fact` is a user–defined routine, so it has its own scope where we have defined its parameter `num` as `"Pi"`, where `P` indicates it is a parameter and `i` that it is integer. Other supported basic types include bytes (`b`), booleans (`o`), reals (`f`), characters (`c`), i-character strings (`s`i) and void (`v`).

User–defined types are also very important in most programming languages, and our stabs support them by means of class `U`. Type `Text` in previous program is a good example because it is a predefined record type. We describe records by means of the following syntax:

```
r:field_1,type_1;...;field_m,type_m;
```

This way, the stab that describes `Text` indicates that it is a record having two fields: one called `handle` of type integer, and another called `name` of type `String`. This is also an user–defined type, an 80-character array with a dimension whose indexes are of type integer. We describe one-dimensional arrays by means of the following syntax:

```
a index_type;low,high;base_type
```

Our stab format also supports multi–dimensional arrays, enumerations, unions, pointers and other usual type constructors. Due to space limitations, we cannot list them all, but you can find a complete description in [4].

2.2 Debugging routines

In order to produce a self–debugging program, compilers also need to insert calls to our debugging routines in the code they generate. The way compilers should use these routines depend very much on the language they compile, but there are some basic rules they should follow. For example, routine `debug_init` initialises every data structure our library needs, and it should be called before any other routine. It reads the appropriate stab file and rebuilds every symbol table the compiler used to compile the original program. `debug_end` shuts down our library and releases all of the resources it acquired when `debug_init` was called. No debugging routine should be called after calling this routine.

Routine name	Short description
`debug_init`	Initialises the library
`debug_open`	Opens and stacks a new scope
`debug_close`	Closes and unstacks the last opened scope
`debug_link`	Links a symbol to its runtime address
`debug_trace`	Interprets debugging commands and traces programs
`debug_end`	Shuts down the library

In the following subsections, we give a short explanation of each routine. There we refer to the following fragment a compiler might produce for the Pascal program we have presented previously.

```
...
100: long _Fact(long _num) {
101:   long result;
102:   debug_open("Fact");
103:   debug_link("num", &_num);
104:
105:   debug_trace(5, "fact.pas");
106:   if (_num == 0) {
107:     debug_trace(6, "fact.pas");
108:     result = 1;
109:   } else {
110:     debug_trace(8, "fact.pas");
111:     result = _num*_Fact(_num - 1);
112:   }
113:
114:   debug_trace(9, "fact.pas");
115:   /* end */
116:
117:   debug_close();
118:   return result;
119: }
...
```

2.2.1 Dealing with scopes

Each section in a stab file corresponds to a different scope in a source file. Scopes are usually stacked at runtime. For instance, when a routine calls another, the scope associated with the callee is stacked on top of the scope associated with the caller. Our library provides two routines to perform this task: `debug_open`, which opens and stacks a scope, and `debug_close`, which closes and unstacks the last stacked scope.

Compilers are responsible for using these routines the right way depending on the scope rules the language they compile define. When a scope is stacked on top of another scope symbols appearing in it override the definitions of symbols with the same name in previous scopes. When the debugger needs to know what a symbol is, it begins searching for it in the current scope (the one on top of the scope stack). If it was not found there, the debugger would search for it in previous scopes.

`debug_open` takes a parameter that indicates what scope it has to open and stack, and it is the name we have written in square brackets in the corresponding stab file. `debug_close` dos not take any arguments because it just unstacks the last opened scope. In our example, calls to these routines have been inserted in lines 102 and 117. Note that `debug_open` is called after declaring local data, but before any other routine is called, and that `debug_close` is called immediately before `_Fact` returns its result.

2.2.2 Linking addressable symbols

A scope provides information about what each symbol is, but they do not have information about where they reside at runtime. Since this information is only available at runtime, our library provides a routine `debug_link` to link each addressable symbol to its physical location.

`debug_link` has two arguments called `name` and `addr`, being the former the name of the object we want to link, and the latter its memory address. In our example, this routine is called in line 103 to link parameter `num` to its address. This parameter, variable `Output` and variable `n` are the only symbols our library can address. Routine `Fact` is also an addressable symbol, but, unfortunately our library is unable to handle it this way, i.e., it is unable to call routines dynamically.

2.2.3 Tracing programs

`debug_trace` is the core of our library. This routine has two parameters that indicate the line and the file the code that follows corresponds to in the original program. The first time it is called, it displays a prompt where the user can type any of the commands we describe in table 2.2.3. This routine is responsible for interpreting the command the user types, and it adjusts its internal data so that subsequent calls to this routine can be handled the right way. For example, if the user types `run` the program is run until its end or a breakpoint is reached, thus causing that subsequent calls to this routine do not prompt the user again.

In our example, we have called this routine several times. For example, in line 105 we call it with arguments `5` and `"fact.pas"`. This means that the code that follows is the translation into C code of the statement at line 5 in file `fact.pas`. Similar calls have been inserted in order to trace the statements at lines 6, 8, and 9. Notice that the statement at line 9 is a fictitious statement, but it helps know when a routine reaches its end.

3 Related work

GNU GDB is probably one of the most successful free debuggers. It is a multi-platform debugger, which means that it can be run on computers ranging from low cost PCs to UNIX workstations and mainframes. It can debug several languages, and all you need to support a new language or a new platform is to write a set of files dealing with the specific features of that language or platform. Nevertheless, in spite of its flexibility, GDB is not suitable for debugging C code generated by a compiler because all references to the original source are lost.

Very little has been published about GDB internals, and one needs to look into its source code in order to find out how it works. We have inspected its code and the little documentation that comes with it thoroughly, and we have found out that it relies on some routines that allow it to write or read machine registers or memory, so each new port needs to provide such low–level routines. This implies that only gurus can really port this debugger. The only thing you need to compile our library on a new platform is to use an ANSI C compiler. The way to supporting a

Command	Description
`next`	Executes next statement. It goes into routine calls
`step`	Similar to `next`, but steps over routine calls
`watch` exp	Evaluates exp
`whatis` exp	Shows information about exp (type, scope, and so forth)
`run`	Execute the following statements until a breakpoint or the end of the program is reached
`animate` num	Similar to `run`, but each sentence is delayed num milliseconds
`set` exp_1 = exp_2	Assigns exp_2 to exp_1
`break` $[f]$ $\#n$ $[c]$	Puts a breakpoint at line n in file f. This breakpoint will stop the program as long as condition c holds when the line containing it is reached. Several breakpoint may be set on the same line
`unbreak` $[f]$ $\#n$ $[i]$	Removes the i–th breakpoint at line n in file f
`dbreak/ebreak` $[f]$ $\#n$ $[i]$	Disable/Enable the i–th breakpoint at line n in file f
`vbreak` $[f]$ $[\#n]$	Lists breakpoints set at line n in file f
`line`	Shows next statement
`source` $[f]$	Shows source file f
`exit`	Aborts program execution and exits
`help`	Displays a help screen where commands are summarized

Table 1: Commands our debugging library provides

new language consists of writing a new parser for the expressions that language uses.

GDB also relies on stabs for storing information about programs, but they are not stored in a separate file, but in the executable itself. GDB assumes that the compiler we are using to compile our programs writes stabs into the assembler code it produces. The assembler just puts these stabs into a special section of the executable program. In contrast, our debugging library reads this information from a separate, portable ASCII file.

Another interesting point is how GDB traces programs. In order to keep track of what statement is being executed at each moment, we insert appropriate calls to `debug_trace` into the generated C code, but GDB makes calls to a UNIX routine called `ptrace`. It allows a debugger to control the execution of a program, which behaves normally until it encounters a signal, at which time it enters a stopped state and the debugger is notified via the `wait` function. When the program is in the stopped state, the debugger can examine and modify its core image using `ptrace`. Unfortunately, this routine is only available on UNIX–like systems, and porting it to other systems seems difficult.

GDB is a command–oriented debugger, which means that it interacts with the user by means of textual commands. Fortunately, its input and output routines can be easily piped so that commands can come from a graphical front–end. Several front–ends have been developed for GDB, being the most important DDD [5]. Our library behaves in a similar way, and we have also developed a graphical front-end that allows the user to debug a program very easily. For the time being, our graphical interface does not allow to display data structures graphically, but we are working hard on this topic.

Finally, the most important difference is that GDB can not deal with fully optimised code, but our debugging library can debug such programs. This is because it has been incorporated into the generated C code, and it traces the logic of our programs, instead of optimised machine code that is very difficult to map into source code.

4 Experimental results

In order to test how our library performs, we have carried out a set of experiments with the following benchmarks:

Bubble sort	Quick sort
Insertion sort	Selection sort
Dichotomic search	Sequential search
Euclide's algorithm	Cramer's rule
Jordan's method	Euler's algorithm
Newton's method	Prime number test
Matrix product	Canonical decomposition
Factorial calculation	Determinant calculation
Fibonacci's succession	Array merging
Non–linear approximation	Polynomial product

We have compiled this set of benchmarks us-

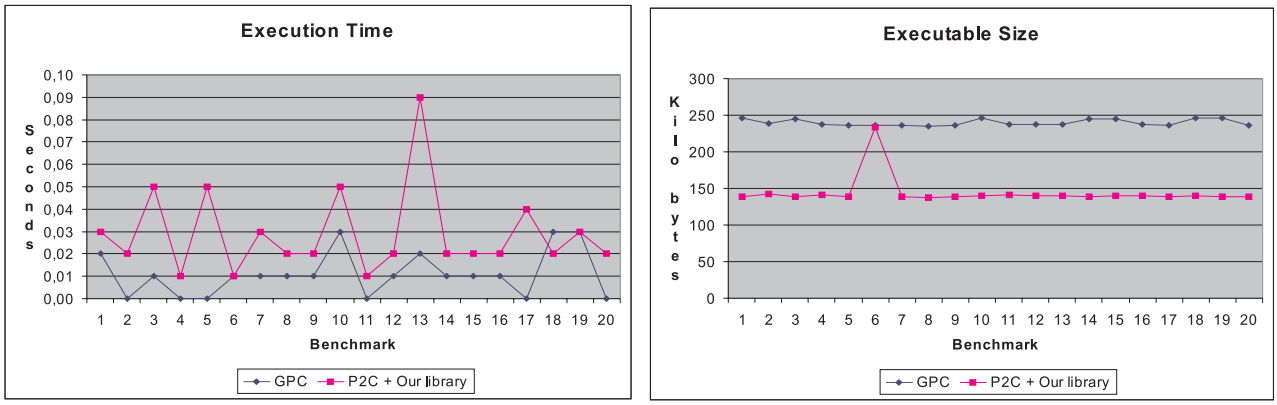

Figure 1: Experimental results.

ing GPC, the GNU Pascal compiler, P2C, a popular PASCAL to C translator whose output we have modified so that it incorporates debugging information. Our comparison concentrated on the size of the executable file with and without debugging information, and on how many CPU time they consume. The experiments were conducted on a low cost Pentium machine running at 200MHz. It was equipped with 64 Mb of memory, Linux RedHat 5.2 Kernel 2.0.36, GPC 2.9.61, and P2C 1.20.

Figure 4 shows the time our benchmarks took to execute when they were compiled using GPC and P2C. Our times are slightly worse in every case because calling `debug_trace` produces a slight overhead. Anyway, this overhead is not significant because the executables that incorporate our library run only 0.02 seconds slower in average, with a standard deviation of 0.02 seconds. This shows that the overhead our library introduces is negligible. We have also analysed the size of the executable programs when we incorporate our library. The amount of code we need to insert is not significant, as well. In fact the executables that incorporate our library are 95.45 Kb smaller than their counterparts compiled with GPC, with a standard deviation of 22.46 Kb. Therefore, our library performs well–enough to be used in practical applications.

5 Conclusions and future work

We have presented a library that allow compilers that generate C code as output to incorporate debugging information that allows to debug such programs. We have added a front–end graphical interface that allows programmers for easy debugging sessions. The library provides a rich set of commands that can help the user find errors out, and we have carried out some experiments that show that our library performs quite well on low cost computers. Moreover, it is portable and it is able to debug fully optimised C code, which is a nice feature other debuggers do not support.

For the time being, we do not support object–oriented programming languages, but we are working hard on this topic.

References

[1] P. Kessler. *The "stabs" Debug Format*. Available on the Internet at `minastirith.cip2b.tu-harburg.de/info/html/stabs_toc.html`

[2] R.M. Stallman. *Debugging with DGB*. Free Software Fundation. 1998

[3] J. Gilmore. *GDB Internals*. Available on the Internet at `www.cs.utah.edu/csinfo/texinfo/gdb/gdbint_toc.html`

[4] J.L. Arjona, J.M. Prieto, and R. Corchuelo. *A symbolic Portable Debugger for Compilers that Generate C code*. Technical Report. Depto. de Lenguajes y Sistemas Informáticos. Universidad de Sevilla

[5] A. Zeller et al. *The Data Display Debugger*. Available on the Internet at `http://www.cs.tu-bs.de/softech/ddd`

[6] G.E. Andrews and R.A. Olson. *The SR Programming Language*. The Benjamin–Cummings Publishing Company. 1993

[7] S. Feurstein et al. *Oracle PL/SQL Programming*. O'Reilly and Associates. 1997

[8] B. Meyer. *Eiffel: The Language*. Prentice–Hall. 1992

[9] *Small Eiffel Home Page*. Available on the Internet at `www.loria.fr/projets/SmallEiffel`

[10] *Ingres II Home Page*. Available the Internet at `www.cai.com/products/ingres.htm`