SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(4), 305–316 (APRIL 1992)

Adding Run-time Checking to the Portable C Compiler

JOSEPH L. STEFFEN
AT&T Bell Laboratories, Room IHC 1P-336, P.O. Box 3013, Naperville, Illinois 60566-7013, U.S.A.

SUMMARY

Run-time checking of array subscripts and pointer bounds was added to the Portable C Compiler (PCC). Memory overwrite bugs are then caught as they happen instead of when the overwritten memory is used later in the program execution. The run-time checking compiler is used both to find the true cause of a core dump and to eliminate run-time errors as the cause of unexpected program behavior. On average, it takes about 40 percent longer to compile, the generated code is about three times larger, and it runs about ten times slower. This performance may seem slow, but it typically reduces days of debugging to less than an hour. The implementation described herein can be adapted to any C compiler as it describes how to generate run-time checking code in a machine/compiler independent way by changing the intermediate expression trees. In four years of use it has found latent bugs and the cause of intermittent core dumps in programs used for many years by thousands of people.

KEY WORDS PCC Error checking Range checking

INTRODUCTION

The need for run-time checking of array subscripts and pointer bounds is apparent to any experienced C programmer. For example, it once took six months to find the cause of an intermittent bug because it could not be reliably reproduced. Saying that some other language should have been used instead of C is not productive, as each language has its problems, and there are usually several constraints that lead to the choice of a particular language.

A language may do run-time checking by definition or through a standard implementation, e.g. SNOBOL4,[1] or make it easy to implement run-time checking of pointers, e.g. ALGOL 68 requires the run-time maintenance of pointer bounds for use by the lower- and upper-bound operators.[2] A language may restrict operations on pointers, such as PL/I[3] or Pascal,[4,5] thus eliminating some types of run-time errors. However, the C language allows completely unrestricted use of pointers, including pointer arithmetic and conversion of integers to pointers. Thus, C may have the greatest need for run-time checking and may also be the most difficult language in which to implement it.[6]

0038–0644/92/040305–12$06.00
© 1992 by John Wiley & Sons, Ltd.

Received 10 April 1991
Revised 21 October 1991

REQUIREMENTS

An excellent description of the language requirements for C language run-time checking is given in Reference 6. The intended use of run-time checking was during debugging and testing, not during production use of the programs, so the user requirements for run-time checking of C language array subscripts and pointer bounds were

(a) an acceptable increase in compilation time, code size, and run time
(b) minimal change to compile/link command names (cc/ld) and options
(c) no restictions on use of the UNIX* operating system calls and C function libraries
(d) minimal restrictions on C language usage, such as casting integer constant 0 (NULL) function arguments to pointers
(e) no restrictions on C preprocessor use, that is, file inclusion (#include), macro definition (#define), and conditional compilation (#if).

DESIGN ALTERNATIVES

Historically, it has been easier to do run-time checking in an interpreter than a compiler because an interpreter can keep symbol information in memory and can augment it with whatever is needed for run-time checking. The code to do the run-time checking and printing of error messages may be large, and is easier to add to an interpreter than to have a compiler generate. At the time this work was started nearly five years ago, interpreters either restricted the language, or had an unacceptable increase in run time. The latter can be solved by partial interpretation, where some files are compiled and others are interpreted, but run-time checking is not done on the compiled files, so the program is only partially checked. Saber-C†[7] allows partial interpretation so existing C function libraries can be used, and because its interpreted code is about 200 times slower than compiled code, so it may be too slow to interpret all the files all the time.

The SPITBOL[8] compiler for the SNOBOL4[1] language showed how a compiler could be written for a language whose only previous implementation had been an interpreter. SPITBOL generated mostly function calls to a large run-time package that the SPITBOL object file was linked to, so it can be thought of as a compiler/interpreter hybrid.

The *bcc*[6] C language run-time checker was a source-to-source translator that took each C language file and added function calls to a run-time package. These files were then compiled by the C compiler for that machine and linked to the run-time package. Note the similarity to the SPITBOL approach. *Bcc* was tried on a 0·8 MIPS machine on two programs of several thousand lines each. It took a long time to get them to compile because the source for parts of the standard C library and all of the *curses* and *termcap* screen terminal libraries had to be found and recompiled. Executing the *bcc*-generated code was supposed to be about 30 times slower than normal, but the programs ran too slow to be useful. For example, several minutes

* UNIX is a registered trademark of UNIX System Laboratories, Inc., a subsidiary of AT&T, in the U.S. and other countries.

† Saber-C is a registered trademark of Saber Software, Inc.

were insufficient for the initialization that normally took a second, so the program execution was aborted.

The main advantage of a source-to-source translator is the ease of porting it to a new machine type, but it was hoped that putting run-time checking into the machine-independent front end of the Portable C Compiler (PCC)[9] would require little if any code to be changed for a new machine type, and that it would be much faster than *bcc*.

The biggest problem with implementing run-time checking is finding a place to store the array and pointer bounds. One approach would be to have a normal size pointer containing a pointer to the real pointer and its bounds. Unfortunately there is now a harder problem of allocating and reclaiming memory associated with pointers that may be in static or dynamic (heap) memory or on the stack. Hashing the address of the pointer into a bounds table allocated from the heap was considered, but this could be too slow and use too much memory, because reclaiming the space for pointers on the stack or in the heap might not be practical. Consider how an array of pointers on the stack would be handled, or a structure (record) containing pointers in the heap.

Another approach is to save the bounds with the variable or object pointed to. SPITBOL had untyped variables and tagged memory so a variable pointed to a type tag, value, and bounds if it was a bounded type. PLUM[3] already had a run-time symbol table, so bounds were added to each entry. *Bcc* makes a pointer three times normal size so it can contain its lower and upper bounds, that is, the memory addresses of the first and last byte of the object the pointer originally pointed to. These larger pointers seemed to require the least change and had the fewest potential problems, despite the frequent use of an integer constant 0 (NULL) as a pointer argument to a function. They could be treated as structures when generating code to pass pointers to and return pointers from functions. See the 'Implementation' section for a discussion of the bounds deduced when addresses are assigned to pointers.

It was not clear how *bcc* checked the upper bound of an array subscript. In C, array names can be used like pointers, but a separate pointer is not allocated by the compiler for an array name. The size of an external array is not known when compiling a file because even if a size is given in the external declaration it may not match the size in the definition in another file that allocates the memory for the array. It was decided to put the upper integer subscript bound just before an external array in memory, which is similar to what SPITBOL does, and does not change the size of the array because the subscript bound is outside the array. Multi-dimension arrays are not a problem because in C they are actually arrays of arrays, so the bounds of each subscript are checked against the array at each subscript level.

COMPARISON OF BCC AND RTCC

Having chosen *bcc* as the model for the run-time checking compiler (*rtcc*), some of its checks were not reimplemented because they were too restrictive or were for unlikely errors:

1. It warned of indirection through a constant integer, but this is possible in code dealing with memory mapped hardware interrupt vectors or peripheral registers.
2. It warned of relational comparison (<=, <, >=, or >) to a null (0) pointer,

but while this may be poor coding it may not be a bug, e.g. $p > 0$ has the same effect as $p != 0$ because pointers are unsigned on most machines.

3. It warned of pointer arithmetic wraparound—it is not worth checking for this unlikely error, given that on many machines the code is at the bottom of the virtual address space and the stack grows from the top down, the program would have to be incrementing a pointer past the bottom of the stack.
4. It caught dangling pointers to the heap at the expense of never reusing freed memory; but it did not catch dangling pointers to the stack, which experience has shown are more frequent, e.g. returning the address of a local variable from a function.

BCC had great error messages including the file name, source line, and an arrow pointing to the part of the line causing the error, but to do this it had to pass the file name, source line and character number to the run-time package; thus increasing compile and run-time. The character number is not available from PCC; and the *sdb* debugger* gives the file name and line number in the stack trace from a core dump, so it was decided to have *rtcc*'s run-time package print a brief message and call the abort standard C library function to cause a core dump.

Since pointers are three times normal size, they must be converted to normal size at some point to interface with the operating system, and this is called encapsulation. *Bcc* did this for most system calls and standard C library functions, but dealing with missing system calls, functions, and libraries was a real user interface drawback, as noted before. It was decided to encapsulate all system calls and compile all libraries with *rtcc*. Many system calls have character string arguments, which should be a pointer to a character array containing at least one null (\0) character to terminate the string. Since there is no run-time checking of pointer bounds within system calls, while encapsulating system calls to remove/add pointer bounds, it was decided to verify that character string arguments are terminated with a null character within bounds, and that other pointer arguments point to an object that is large enough to hold the returned data. For example, the open system call's file name character string argument is checked for a non-null pointer within the bounds of a character array, and that there is a null character between where the pointer points and the end of the array. The read system call's buffer pointer argument is checked to be non-null, within bounds, and that the number of bytes between the pointer and the upper bound are as large as the number of bytes to be read.

On machines where pointers and integers (int) are the same size, an integer constant 0 (NULL) is often used as a function argument without casting it to a pointer. The run-time checking code just takes the next two memory words as the bounds of this null pointer, and it may cause an out-of-bounds pointer error. Thus, before *rtcc* can be used to find the real run-time error that is causing a core dump, pointer casts must be added to all uses of 0 as a function pointer argument. *Bcc* had a user manual section on using *lint* to help find these places in the code. Unfortunately, the few *lint* messages that affect run-time checking must be found in the mass of *lint* output, and there may still be occurrences that *lint* does not find. This checking would be best done with an enhanced compiler that would store the equivalent of ANSI C function prototypes in the object file and an enhanced linker that would

* Systems based on Berkeley UNIX have an equivalent debugger called *dbx*.

check them, but this work was deferred. Since it is difficult to find 0 or NULL arguments visually in thousands of lines of code, it was decided to add an option to *lint* to just print messages for pointer/non-pointer function argument and return value mismatches. This option is necessary even if run-time checking is put in an ANSI C compiler because function prototypes are optional so the compiler cannot depend on every function having a template. Run-time checking was added to a PCC that had function templates, which are syntactically identical to prototypes but the compiler does not coerce function arguments to the correct type. Programmers found that templates were much work to add and maintain in a large system, so their use did not become widespread.

OVERVIEW

Before describing the implementation of *rtcc*, it is best to describe it and its related commands, how they are used, and problems that may be encountered.

Rtlint is a version of the *lint* C program static analyser command that only checks for use of an int where a pointer was expected in a function argument or return value. This check is needed as pointers are three times normal size because they contain the upper and lower bound of the object being pointed to, in addition to the pointer value, so before using *rtcc*, use *rtlint* on all C source files, e.g.

```
rtlint *.c
```

If *rtlint* prints any messages, fix the cause of them, and rerun *rtlint* until there are no messages. *Rtcc* is a version of the *cc* C program compiler command that generates code to check array subscripts and pointer bounds at run-time, so compile the program with *rtcc*

```
rtcc *.c
```

Now execute the program. If a message prefaced by rtcc: is printed: a run-time check has failed, e.g.

```
rtcc: null pointer used
sh: 29655 abort—core dumped
```

The second message occurs because the run-time checking code calls abort to cause a core dump.

Rtsdb is a version of the *sdb* symbolic debugging command that understands that pointers have an upper and lower bound, and prints the bounds in angular brackets. It also has access to the C language source files of the standard C library and other libraries. Now execute *rtsdb*, whose output is shown in italics

```
rtsdb
kill:address 0x15632
```

and use its t command to print a stack trace of the program

```
t
kill(0xf5e,0x6)
abort() [abort.c:15]
_error_(s=1048933) [rtcrt0.c.:80]
_ind_(pointer=0x1055a8) [rtcrt0.c:183]
main(argc=1,argv=0x100b30(0x100b30-0x100b47)) [main.c:7]
```

On the top of the stack is the kill system call made by abort, which is called by _error_ in the run-time checking code (rtcrt0.c). The run-time error was found in the function below the last rtcrt0.c function in the stack trace, which is main in this example. Use the e command to set the current procedure in *rtsdb* to main

```
e main
```

and print the source line by typing the line number in the stack trace

```
7
        if(*p==*q)
```

Print the values of the pointers in the source line to find the one that is null

```
p/
0x100b30(0x100b30-0x100b47)
q/
0(0-0)
```

The bounds of a pointer are printed inside angular brackets. Note that q is the null pointer because its value is 0. Its bounds are also 0 because a null pointer does not point to anything.

All run-time pointer errors except for dereferencing (using) a dangling pointer are caught. A dangling pointer is either a global pointer to an automatic (stack) variable after the function defining it has returned, or a pointer to memory allocated with malloc that has been freed with free. The latter form of dangling pointer can be caught by *rtcc* if the pointer is set to 0 after it has been freed, e.g.

```
        free(p);
#if RTCC
        p = 0;
#endif
```

If there is code assigning p to other pointers, then these pointers also have to be set to 0. Note that *rtcc* sets the RTCC preprocessor variable so the additional code above is compiled only when using *rtcc*.

Run-time checking can be circumventing by casting a pointer to *int* and back again because this makes the pointer bounds all memory except location 0 (null). Occasionally it is easier to do this than to rewrite code, e.g. when the addresses of two structure members are used to copy a portion of one structure to another, or if the program has its own version of malloc. If the latter, a bounded pointer p to allocated memory of size s can be generated by calling the _pca_ function in the run-time checking package:

```
#if RTCC
        char **_pca_();
        p = (char *) _pca_((int) p, s - 1);
#endif
```

IMPLEMENTATION

The intent of this section is to describe a PCC host-independent implementation for *rtcc* and its related commands under the UNIX operating system.

Rtlint

Rtlint just calls *lint* with an option (−G) to just print messages for pointer/non-pointer function argument and return value mismatches.

Rtcc

Similarly, *rtcc* just calls the *cc* C program compiler command with the −G option, which also causes any execution profiling option (−p) to be ignored. *Cc* executes other files that do phases of the compilation:

(a) C preprocessing (cpp)
(b) compiling into assembly language (comp)
(c) optimizing (optim)
(d) assembling into an object file (as)
(e) linking into an executable file (ld)

and various options are passed to them for run-time checking, e.g. the −DRTCC option to define the RTCC preprocessor symbol for use in *#if* directives is passed to cpp.

The −G option is passed to comp to generate code to check array subscripts and pointer bounds by inserting calls to the run-time checking package rtcrt0.c into the expression tree as it is created in the machine-independent part of comp. Since there are few changes to the machine-dependent part of comp, porting the run-time checking code to another version of PCC is as easy as possible. The −g option is enabled and passed to comp to produce the symbolic debugging information.

The −G option is passed to as to set the run-time checking flag in the object file header, and to ld to verify that all object files were compiled with *rtcc* and to set the run-time checking flag in the executable file header. Library names in −l options have rt prepended to them before they are passed to ld, e.g. −lm becomes −lrtm. The −lg option is passed to ld so the object file is linked with the *sdb* library. The run-time checking package is passed to ld instead of the normal C run-time package as the interface between the UNIX exec operating system calls and a C program.

Compiler phase (comp)

If comp is passed the −G option, it generates assembly language code to call the run-time package to check array subscripts and to generate and check pointer bounds. The generated bounds of

(a) a null (0) pointer are 0,
(b) an integer constant are all memory except location 0,
(c) a string constant are the memory containing the string,
(d) a function pointer are the function address,
(e) other pointers are the memory containing the object they point to.

The upper integer subscript bound is put just before an external array in memory. The latter is one of two machine-dependent changes to comp because the assembly language for declaring the upper subscript bound's value differs among assemblers. In an undimensioned initialized global array declaration, the upper subscript bound is not known until the end of the initialization value list, so the value of the upper subscript bound must be an assembler symbol, and an assembler definition of the symbol must be emitted after initialization.

Constant array subscripts not in address (&) expressions are checked at compile-time for negative values and values greater than the array dimension, if known*. These checks are done even when run-time checking is not requested, so they cause warning instead of error messages to allow compilation to continue.

Since the assembly language to declare an external name also differs among assemblers, the other machine-dependent change is to add the leading and trailing underscores (_) to encapsulated names of system calls and the malloc and realloc memory allocation library functions.

Since pointers are three times normal size, to preserve the call-by-value semantics of pointer function arguments, they are passed to and returned from functions compiled with *rtcc* and encapsulation functions as if they were structures. This change to argument passing requires tighter checking of pointer/non-pointer type mismatches, so the 'illegal combination of pointer and integer' message was changed from a warning to an error so it stops compilation for run-time checking, and the 'function was not declared to return a pointer' error message was added. These machine-independent changes are in the code shared with *rtlint*.

Pointers are not passed to or returned from the non-encapsulation run-time checking functions as structures; instead the address of a pointer is passed or returned, partly because it is faster and the generated code is smaller. More significantly however, the temporary register allocation and spilling in comp does not have to be changed to allocate three registers for a pointer, spill the pointer registers into a temporary structure on the stack when it runs out of temporary registers, and reload them when needed later in the generated code for the expression.

C run-time package (rtcrt0.c)

The exec system calls do not call the main function of a C program; they call the _crt function in the C run-time package. In the C language, it looks something like

```
_crt0(argc, argv, envp)
int    argc;
char   *argv[], *envp[];
{
       extern char **environ;
```

* The latter check caught surprisingly many off-by-one errors in a large software system.

```
        environ = envp;
        exit(main(argc, argv, envp));
    }
```

The name of this function may differ among UNIX variants, e.g. it may be start or _start because it is sometimes called the start-up function. For run-time checking, before they are passed to the main function, the normal pointer size argument (argv) and environment (envp) string arrays must be copied into three times larger arrays with pointer bounds calculated from the string lengths.

The previously described system call encapsulations are also in this file, and they remove the bounds from pointers passed to system calls, including pointers in arrays or structures. Pointer bounds are added to pointers returned from system calls such as sbrk and the malloc and realloc memory allocation library functions. For example, the sbrk system call returns a pointer that is the old break value with bounds of the original break value (&end) and the new break value less one; malloc and realloc return pointers with the lower bound equal to the pointer and the upper bound equal to the pointer plus the size argument less one. The system call encapsulations also verify that string pointers arguments are terminated a null (0) character, pointers to buffers have enough space for the data to be read, and other pointers are non-null unless null is allowed. The realloc library function encapsulation's first argument is not checked for null because if it is null it causes a run-time error ayway.

This file also contains the run-time checking functions called by the compiler-generated run-time checking code in the program, and they and the encapsulations produce these error messages:

(a) array subscript is too big
An array subscript is larger than the size of the array. Note that the largest valid array subscript is one less than the size of the array, because the first valid subscript of an array is 0, not 1.

(b) negative array subscript
An array subscript is less than 0.

(c) null pointer used
A null (0) pointer was used with the indirection (*) operator, the −> operator, an array subscript, or as an argument to the brk or shmdt system calls. Note that a null pointer is not a null ("") string.

(d) out-of-bounds pointer used
A pointer is not within the bounds of the object pointed to, and it was used with the indirection (*) operator, the −> operator, or an array subscript.

(e) use of pointer to an object that is too small
A pointer is within the bounds of the object pointed to, but this object is too small to contain the stored value. For example, the address of a char variable instead of an int was passed to the scanf library function.

(f) *system call*: *argument* points to an object that is too small, must be >= *size* bytes
This *argument* to this *system call* points to an array, structure, or union that is too small. *Argument* matches the argument name in the user manual entry for this *system call.*

(g) *system call*: *argument* is not a null-terminated string
This *argument* to this *system call* points to a character array that does not

contain a null ('\0') character. *Argument* matches the argument name in the user manual entry for this *system call.*

(h) msgctl:invalid cmd
(i) semctl:invalid cmd
(j) shmctl:invalid cmd
This system call's cmd argument is invalid.

Rtsdb

Rtsdb is just a consistent alias for the enhanced *sdb* symbolic debugger that recognizes an executable file compiled with *rtcc* and displays a pointer's bound in angular brackets after its value, e.g.

```
0x100b30<0x100b30-0x100b47>
```

However, pointers in the run-time package do not have bounds so they are not printed. *Rtsdb* also searches a directory of library source files so the *sdb* subcommands to display the C language source can be used when a run-time error occurs in a library function such as strcpy (string copy).

TESTING

Rtcc was tested with an extensive set of run-time checking tests; and sets of compiler, system call, and library regression tests. Not only were bugs found in the new run-time checking code, but also in the old compiler code where it was unable to generate code for some expressions involving functions returning structures or pointers, e.g.

```
function().member
function()->member
```

Many run-time errors were found while testing libraries; some of these required circumventing the run-time checking by casting a pointer to an integer and back to a pointer to give bounds of all memory except location 0. For example, in the *libc* free and realloc functions the bounds had to be removed from the pointer argument so they could access the preceding union containing the busy bit or the free list pointer. The bounds of the pointer returned by malloc do not include this union so it cannot be accidentally overwritten by the program being checked. Other run-time errors were caused by poor coding practices such as

(a) not using the varargs macros for functions with variable numbers of arguments,
(b) casting a pointer to a pointer to a different sized object,
(c) dereferencing a pointer that had been incremented past the structure member it originally pointed to.

For the latter it was often easiest to use a cast to defeat the run-time checking rather than extensively rewrite the code. The remaining four run-time errors were bugs that are probably harmless but had to be fixed. Some were surprising in that they were in code that had been used for years, such as an uninitialized local variable in fseek.

RESULTS

On average, the *rtcc* compile time is about 40 per cent longer than *cc* because it is generating code (.text) that is about three times larger, and the code runs about ten times slower. Rtcc has been used for four years by many programmers; it found the cause of intermittent core dumps in the compiler and an internal version of the *emacs* editor, and it found four latent bugs in *ctrace*[10] that had not surfaced in nine years of use. It reduces the time needed to find memory overwrite bugs because they are caught as they happen instead of when the overwritten memory is used later in the program execution. Even if a run-time error is not found in a misbehaving program, it aids debugging because this eliminates the whole class of memory overwrite bugs so another cause for the problem can be searched for.

Some programmers assume that the size of a pointer equals the size of an int or long integer, which can require code changes so *rtcc* generated code does not falsely report run-time errors. For example, the PCC parse tree node union of structures has this assumption, so this code had to be added after the integer structure member corresponding to a pointer in another structure in the union

```
#if RTCC
    int lowerbound, upperbound;       /*allow for "NODE*left;" bounds*/
#endif
```

Only one program, *sdb*, has resisted all efforts to use *rtcc* on it because it is unable to recognize a core dump file because the process user structure contains pointers. When the core dump file is created, these pointers are normal size pointers, but *sdb* compiled with *rtcc* expects them to have bounds and thus be three times normal size.

Porting the run-time checking code to another compiler will not be as easy as hoped, mainly because of the compiler bugs that are exposed by the run-time checking code generation, and the extensive testing required for confidence in the generated code. It took about six months to implement *rtcc* on an Amdahl/IBM mainframe, and it is estimated it would take two months to port it to the PCC for another machine. Funding for porting was never approved, possibly because the rapid rise and fall in use of different machine types makes it difficult to justify this effort for machines that may be in use only a few years, and possibly because most past users of *rtcc* still have access to an Amdahl/IBM mainframe and can copy their code to it to use *rtcc*. Even more effort would be needed to add run-time checking to the ANSI C and C++ languages because their compilers/translators are not based on PCC, so all the machine-independent changes would have to be redone.

CONCLUSIONS

The user interface of *rtcc* could be improved; users do not realize that they must use *rtlint* before *rtcc*, and find *rtsdb* hard to learn to use if they have not used *sdb* before. Surprisingly, many experienced C programmers rarely if ever use *lint* or *sdb*.

The performance of *rtcc* is good and the code it produces is of manageable size and runs fast enough. All in all, *rtcc* works better and faster than *bcc* with fewer restrictions on the C language code used, and no restrictions on library usage, so all the requirements were met. It has proved the usefulness of run-time checking time and again in the last four years.

REFERENCES

1. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd edn, Prentice-Hall, Englewood Cliffs, NJ 1971.
2. C. M. Thompson, 'Error checking, tracing, and dumping in an ALGOL 68 checkout compiler', *ACM SIGPLAN Notices*, **12**, 106–111 (1977).
3. M. V. Zelkowitz, Paul R. McMullin, Keith R. Merkel and Howard J. Larsen, 'Error checking with pointer variables', *Proceedings of the 1976 ACM National Conference*, ACM, New York, 1976.
4. J. Welsh, 'Economics range checks in Pascal', *Software—Practice and Experience*, **8**, 85–97 (1978).
5. C. N. Fischer and R.N. LeBlanc, 'The implementation of run-time diagnostics in Pascal', *IEEE Trans. Software Engineering*, **6**, 313–319 (1980).
6. S. C. Kendall, 'Bcc: runtime checking for C programs'. *USENIX Toronto 1983 Summer Conference Proceedings*, USENIX Association, El Cerrito, CA 1983.
7. S. Kaufer, R. Lopez and S. Pratap, 'Saber-C: an interpreter-based programming environment for the C language', *USENIX San Francisco 1988 Summer Conference Proceedings*, USENIX Association, El Cerrito, CA 1988.
8. R. B. K. Dewar, *SPITBOL Version 2.0*, Illinois Institute of Technology, Chicago 1971.
9. S. C. Johnson, 'A portable compiler: theory and practice', *Fifth ACM Symposium on Principles of Programming Languages Conference Record*, ACM, New York, 1978.
10. J. L. Steffen, 'Experience with a portable debugging tool', *Software—Practice and Experience*, **14**, 323–334 (1984).