Asserting the Precision of Floating-Point Computations: a Simple Abstract Interpreter[1]

E. Goubault, M. Martel and S. Putot

CEA - Recherche Technologique, LIST-DTSI-SLA
CEA F91191 Gif-Sur-Yvette Cedex, France
e-mail : [goubault,mmartel,sputot]@cea.fr

1 Introduction

The manipulation of real numbers by computers is approximated by floating-point arithmetic, which uses a finite representation of numbers. This implies that a (small in general) rounding error may be committed at each operation. Although this approximation is accurate enough for most applications, there are some cases where results become irrelevant because of the precision lost at some stages of the computation, even when the underlying numerical scheme is stable. In this paper, we present a tool for studying the propagation of rounding errors in floating-point computations, that carries out some ideas proposed in [3], [7]. Its aim is to detect automatically a possible catastrophic loss of precision, and its source. The tool is intended to cope with real industrial problems, and we believe it is specially appropriate for critical instrumentation software. On these numerically quite simple programs, we believe our tool will bring some very helpful information, and allow us to find possible programming errors such as potentially dangerous double/float conversions, or blatant unstabilities or losses of accuracy. The techniques used being those of static analysis, the tool will not compete on numerically intensive codes with a numerician's study of stability. Neither is it designed for helping to find better numerical schemes. But, it is automatic and in comparison with a study of sensitivity to data, brings about the contribution of rounding errors occuring at every intermediary step of the computation. Moreover, static analyzes are sure (but may be pessimistic) and consider a set of possible executions and not just one, which is the essential requirement a verification tool for critical software must meet.

2 Main Features

Basically, the error $r - f$ between the results f and r of the same computation done with floating-point and real numbers is decomposed into a sum of error terms corresponding to the elementary operations done to obtain f. An elementary operation introduces a new rounding error which is then added, multiplied

[1] This work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

etc. by the next operations on the approximated partial result. For example, let x and y be initial data and let us assume that errors are attached to these numbers (we assume that we only have three digits of precision). The notation $x^{\ell_1} = 1.01\varepsilon + 0.005\varepsilon_{\ell_1}$ indicates that the floating-point value of x is 1.01 and that an error of magnitude 0.005 was introduced on this value at point ℓ_1. If $y^{\ell_2} = 10.1\varepsilon + 0.05\varepsilon_{\ell_2}$ then $x +^{\ell_3} y = 11.1\varepsilon + 0.005\varepsilon_{\ell_1} + 0.05\varepsilon_{\ell_2} + 0.01\varepsilon_{\ell_3}$ and $x \times^{\ell_3} y = 10.2\varepsilon + 0.0505\varepsilon_{\ell_1} + 0.0505\varepsilon_{\ell_2} + 0.001\varepsilon_{\ell_3} + 0.00025\varepsilon_{\varsigma}$. In $x +^{\ell_3} y$, the errors terms on x and y are added and this operation, done at point ℓ_3, introduces a new error term $0.001\varepsilon_{\ell_3}$ due to the truncation of the result. $x \times^{\ell_3} y$ also introduces an higher order error term $0.00025\varepsilon_{\varsigma}$ due to the factor $0.005\varepsilon_{\ell_1} \times 0.05\varepsilon_{\ell_2}$.

More generally, a floating-point number f with errors is denoted by an *error series* $f\varepsilon + \sum_{\ell \in \mathcal{L}} \omega^{\ell} \varepsilon_{\ell}$ where $\mathcal{L}$ is a set of syntactic program points and $\omega^{\ell} \varepsilon_{\ell}$ is the contribution to the global error of the error introduced at the point $\ell \in \mathcal{L}$ and propagated in the following computations. The tool described in this article implements an abstract interpretation [1] based on this model, where the value f and the coefficients ω^{ℓ} are abstracted by intervals.

As shown in Figure 1, the main window of the analyzer displays the code of the program being analyzed, the list of identifiers occurring in the abstract environment at the end of the analysis and a graph representation of the abstract value related to the selected identifier in the list. Scrollbars on the sides of the graph window are used to do various kinds of zooms.

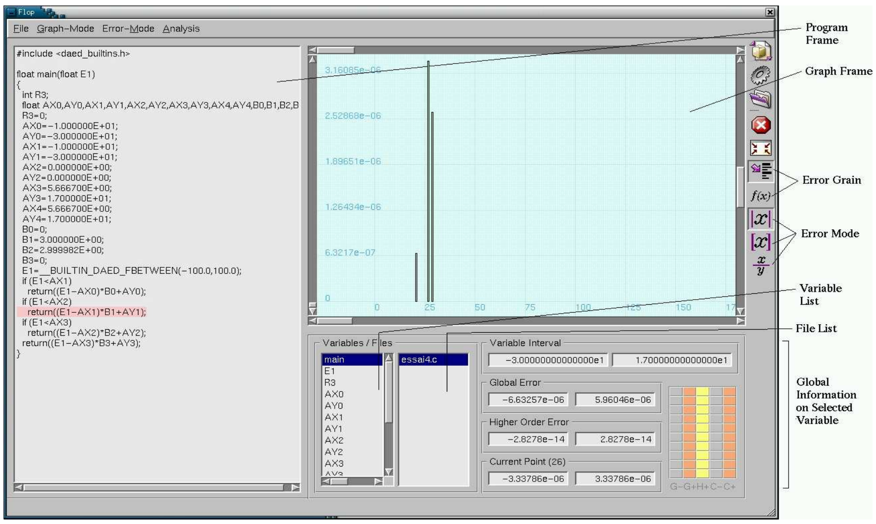

Fig. 1. Main window of the analyzer.

The graph represents the error series of a variable *id* and thus shows the contribution of the operations to the global error on *id*. The operations are identified with their program point which is displayed on the X-axis.

In Figure 1, the bars indicate the maximum of the absolute values of the interval bounds. This enables to assert the correctness of the code, when the errors have a small magnitude. This kind of graph is well suited to identify some numerical errors, but other kinds of errors, like cancellations [2], are more easily detected by graphs showing relative errors. Different types of graphs can be drawn by clicking the adequate button in the right-hand side toolbar. The graph and program code frames are connected in the graphical user interface in such a way that clicking on a column of the graph makes the code frame emphasizes the related program block and conversely. This enables the user to easily identify which piece of code mainly contributes to the error on the selected variable. Another interesting feature is that different grains of program points (like code lines or functions) can be selected by clicking the adequate button. Hence, for the analysis of a program made of many functions, the user may first identify which functions introduce the most important errors and next refine the result.

In the example of Figure 1, a typical program of an instrumentation sofware is being analyzed. It is basically an interpolation function with thresholds. One can see from the graph at the right-hand side of the code that the only sources of imprecision for the final result of the main function are: the floating-point approximation of the constant 2.999982 at line 20 (click on the first bar to outline the corresponding line in the C code), which is negligible, the 2nd `return` line 26 (second bar in the graph), and the 3rd `return` line 28 (third and last bar in the error graph), the last two ones being the more important. In fact, using `__BUILTIN_DAED_FBETWEEN` (an assertion of the analyzer), we imposed that `E1` takes its value at the entry of the function between -100 and 100. So the analyzer derives that the function does not go through the first `return`. Then it derives that the function can go through the 4th and last `return`, but the multiplication is by zero and the constant is exact in the expression returned, so that there is no imprecision due to that case. The user can deduce from this that if he wants to improve the result, he can improve the accuracy of the computation of the 2nd and 3rd `return`. One simple way is to improve the accuracy of the two subtractions in these two expressions (using `double E1` in particular), whereas the improvement of the precision of the constant 2.999982 is not the better way. Notice that the analyzer also finds that the higher-order errors are always negligible. It will also be demonstrated that simple looping programs, such as linear filters can be proved stable or unstable accordingly by the analyzer, so the method does also work for loops (to a certain amount of precision).

3 Implementation

We are at a point where we have a first prototype of a static analyzer for full ANSI C programs, which abstracts floating-point variables by series of intervals. For the time being, if it can parse all ANSI C, it does not interpret aliases, arrays and struct information on the code. Nevertheless, it is already interprocedural, using simple static partitioning techniques [6].

The analyzer is based on a simple abstract interpreter [4] developed at CEA. The interval representing the floating-point value is implemented using classical floating-point numbers, and higher precision is used for the errors (necessary because numerical computations are done on these errors, and they must be done more accurately than usual floating-point). We use a library for multiprecision arithmetic, MPFR [5], based on GNU MP, but which provides the exact features of IEEE754, in particular the rounding modes. The interface is based on Trolltech's QT and communicates with the analyzer through data files.

The computational cost of the analysis is really reasonable. To give an idea, on small toy examples, typically a loop including a few operations, the analysis takes less than 0.1 second. On a more complex example of about 500 lines, with no loop, it takes only 45 seconds.

4 Conclusion and Future Work

The current implementation of our prototype already gives some interesting results on simple programs, which we propose to show during the demo. Concerning the threats detected by the analyzer, various reasons like cancellations or instabilites in loops may contribute to the loss of precision and some phenomena are easier to detect with a particular graph representation of the errors. In interaction with users, we are working on the best way to represent the many results collected by the analysis as well as on the methodology needed to their treatment. We also work on improving the precision of the analysis in loops. Because narrowings do not improve the analysis for the error terms, the approximation made by widenings must be fairly precise. We also plan to use relational lattices as discussed in [3].

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. *Principles of Programming Languages 4*, pages 238–252, 1977.

[2] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.

[3] E. Goubault. Static analyses of the precision of floating-point operations. In *SAS'01*, LNCS. Springer-Verlag, 2001.

[4] E. Goubault, D. Guilbaud, A. Pacalet, B. Starynkévitch, and F. Védrine. A simple abstract interpreter for threat detection and test case generation. In *Proceedings of WAPATV'01 (ICSE'01)*, May 2001.

[5] G. Hanrot, V. Lefevre, F. Rouillier, and P. Zimmermann. The MPFR library. Institut de Recherche en Informatique et Automatique, 2001.

[6] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, 1982.

[7] M. Martel. Propagation of rounding errors in finite precision computations: a semantics approach. ESOP, 2002.