Static analyses of floating-point operations

Eric Goubault*
LIST (CEA - Recherche Technologique)
DTSI-SLA, CEA F91191 Gif-sur-Yvette Cedex

February 22, 2001

Abstract

Computers manipulate approximations of real numbers, called floating-point numbers. The calculations they make are accurate enough for most applications. Unfortunately, in some (catastrophic) situations, the floating-point operations lose so much precision that they quickly become irrelevant. In this article, we review some of the problems one can encounter, focussing on the IEEE754-1985 norm. We give a (sketch of a) semantics of its basic operations then abstract them (in the sense of abstract interpretation) to extract information about the possible loss of precision. The expected application is abstract debugging of software ranging from simple on-board systems (which use more and more on-the-shelf micro-processors with floating-point units) to scientific codes. The abstract analysis is demonstrated on simple examples and compared with related work.

1 Introduction

Everybody knows that computers calculate numerical results which are mostly wrong, yet they are intensively used for simulating highly complex physical processes and for predicting their behavior. Transcendental numbers (like π and e) cannot be represented exactly in a computer, since machines only use finite implementations of numbers (floating-point numbers instead of mathematical real numbers); they are truncated to a given number of decimals. Less known is that the usual algebraic laws (associativity for instance) that we use when thinking about numbers are no longer true in general when it comes to manipulating floating-point numbers.

It is actually surprising that very few studies on static analysis of floating-point operations or on their semantic foundations have been carried out. Our point of view in this article is that there are "numerical bugs" that a programmer can encounter, and that some are amenable to automatic detection using static analysis of the source code, using abstract interpretation. This new sort of bug includes what is normally called bug, i.e. run-time errors (here for instance, uncaught numerical exceptions), but also more subtle ones about the relevance of the numerical calculations that are made. We advocate that it is as much of a bug to terminate on a "segmentation fault" as to terminate with a completely meaningless numerical result (which might be used to control a physical apparatus with catastrophic consequences).

This problem is not very well-known to programmers of non-scientific codes. Let us just give one example showing this is also of importance for the non-scientific computing world. On the 25th of February 1991, during the Gulf war, a Patriot anti-missile missed a Scud in Dharan and crashed onto an American barracks, killing 28 soldiers. The official enquiry report (GAO/IMTEC-92-26) attributed this to a fairly simple "numerical bug". An internal clock that delivers a tick every tenth of a second controlled the missile. Internal time was converted in seconds by multiplying the number of ticks by $\frac{1}{10}$ in a 24 bits register. But $\frac{1}{10} = 0.00011001100110011001100\cdots$ in binary

*Work partially funded by IST project DAEDALUS. This paper follows a seminar given at Ecole Normale Supérieure in June 1998.

format, i.e. is not represented in an exact manner in memory. This produced a truncating error of about 0.000000095 (decimal), which made the internal computed time drift with respect to ground systems. The battery was in operation for about 100 hours which made the drift of about 0.34 seconds. A Scud flies at about 1676m/s, so the clock error corresponded to a localization error of about 500 meters. The proximity sensors supposed to trigger the explosion of the anti-missile could not find the Scud and therefore the missile fell and hit the ground, exploding onto the barracks.

Actually, more and more critical or on-board systems are using on-the-shelf floating-point units which used not to be approved beforehand. Therefore we believe that static analysis of floating-point operations is going to be very important in the near future, for safety-critical software as well as for numerical applications in the large.

These kinds of problems are better-known in scientific computing, at least when modeling the physical phenomena to be simulated. What we mean is that in many cases, the discretizations of the (continuous) problems that are modeled are sufficiently stable so that little truncation errors do not overly affect the result of their simulation. Unfortunately, it is difficult to find the exact semantics of floating-point operations, and even using some well-behaved numerical schemes, some unpredictable numerical errors can show up. Also some problems are inherently ill-conditioned, meaning that their sensitivity to numerical errors are very high. In this latter case it is in general very difficult to assess the relevance of the numerical simulation even by hand.

Organization of the paper In Section 2, we will explain what model of floating-point arithmetic we want to analyze (IEEE754-1985). We carry on in Section 2.1 by explaining what kind of properties we want to synthesize by the analysis. Then in Section 2.2 we give the syntax and informal meaning of a simple imperative toy language manipulating floating-point numbers; we give a first sketch of a formal semantics in Section 2.3, that we refine in section 3.2.

In Section 3 we present a few abstract domains that are candidate for the abstract interpretation of the concrete semantics. We give some directions for improvement in Section 5.1. We give an example in Section 4, and compare with existing related work in Section 6. We conclude by giving some future directions of work in Section 7.

2 The IEEE 754 norm

The IEEE754-1985 norm specifies how real numbers are represented in *memory*[1] using floating-point numbers, see [Gol91, Kah96]. The norm itself relies on a simple observation:

Lemma 1

$$\begin{array}{rlcccccl} & \{-1,1\} & \times & [0,1[& \times & \mathbb{N} & \to & \mathbb{R}^* \\ \mathcal{F}: & (s, & & f, & & k) & \to & s(1+f)2^k \end{array}$$

is a bijection with inverse:

$$\begin{array}{rlclcccc} & \mathbb{R}^* & \to & \{-1,1\} & \times & [0,1[& \times & \mathbb{N} \\ \mathcal{G}: & x & \to & (s(x), & & f(x), & & k(x)) \end{array}$$

with $s(x)$ being the sign of x, $k(x) = \lfloor log_2(\mid x \mid) \rfloor$ where $\lfloor u \rfloor$ denotes the integral part[2] *of u and log_2 is the logarithm in base 2, and $f(x) = \frac{|x|}{2^{k(x)}} - 1$.*

Taking a representation with a fixed number of bits K for exponents (function $k(x)$) and a fixed number of bits N for the mantissa (function $f(x)$ or $m(x) = 1 + f(x)$), the norm defines several kinds of floating-point numbers,

[1] But not in the registers of micro-processors.

[2] i.e. the greatest integer less or equal than x. We will also use $\lceil u \rceil$ which is the least integer greater or equal than x.

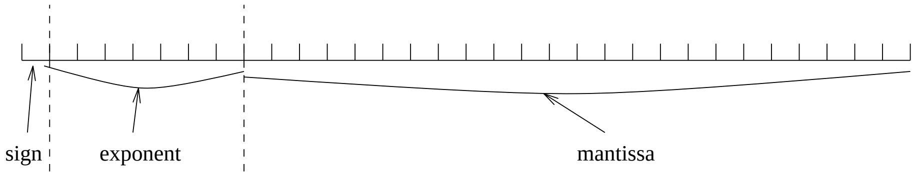

Figure 1: Representation of a floating-point number in memory.

- The standard numbers, $r = s * n * 2^{k+1-N}$, with $s \in \{-1, 1\}$, $1 - 2^K < k < 2^K$, $0 \leq n < 2^N$ normalized so that, $r = s * 2^k(1 + f)$ with $f < 1$,
- De-normalized numbers (to manage "underflow" in a gradual manner), $r = s * n * 2^{k+1-N} = s * 2^k(0 + f)$ with $k = 2 - 2^K$ and $0 < n < 2^{N-1}$ i.e. $0 < f < 1$,
- $+\infty$ and $-\infty$ (notice that their inverses, $+0$ et -0 are also there),
- `NaN` "Not a Number" signed or not (which are the results of dubious operations such as $0 * \infty$).

Normalized numbers come in several versions, according to different choices of K and N, so allowing more or less precision at will. Simple precision (`REAL*4`, `float`) has $K = 7$ and $N = 24$, double precision (`REAL*8`, `double`) has $K = 10$, $N = 53$, and double extended (`REAL*10` etc., `long double`) has $K \geq 14$, $N \geq 64$.

Just to give an order of magnitude of the numbers we are talking about, let us show a few examples. For a simple float, the maximum normalized number is $3.40282347 * 10^{38}$, the minimum positive normalized number is $1.17549435 * 10^{-38}$, the maximum de-normalized number is $1.17549421 * 10^{-38}$ minimum positive de-normalized number is $1.40129846 * 10^{-45}$. Around 1, the maximal error ("unit in the last place", or ulp, or $ulp(1)$) is 2^{-23} for a simple float, i.e. about $1.19200928955 * 10^{-7}$.

The norm also specifies some properties of some of the computations we can make on floating-point numbers. For instance, the norm specifies that $+, -, *, /, \sqrt{}$ are computed with an inaccuracy that cannot go beyond the ulp around the exact result (if there is no "overflow").

The norm allows the user to use different round-off methods. One can use round-off towards zero, round-off towards the nearest, round-off towards plus infinity, and round-off towards minus infinity. A more subtle rule is that when we have the choice between two roundings (in the round-off towards the nearest mode), we choose the even mantissa. In fact, the norm even specifies[3] that $x.y$ (where . is one of the floating-point operations +, -, *, / on floating-point numbers x and y) is the rounding (in the corresponding rounding mode) of $x \circ y$ (where $\circ$ is the corresponding operation in $\mathbb{R}$). We will actually suppose in the sequel that this holds true also for the square root operation as well.

The conversions are to be given an explicit semantics as well. More annoying is that we should take care of the order of evaluation (in conflict with compiler optimizations!), since the round-offs destroy associativity in general.

Caveats As we said in the beginning of this section, the norm specifies what happens in memory but not in processor registers. There are conversions between memory and registers that we have to know about. In general, (except M680x0 and Ix86/Ix87 where all operations are computed in double extended before round-off), registers are like main memory. There can be some differences with RISC processors as well, like the IBM Power PC or Apple Power Macintosh, because of the use of compound instructions (multiply-add etc.) which do not use the same round-off methods. Most of the machines follow the norm anyway but not all the compilers in particular concerning the way they handle (or do not handle!) arithmetic exceptions (underflow etc.). CRAY used to

[3] This is done using extra "guard digits" for computation by the processor of the operations.

have a different arithmetic, which is a problem for actual applicability of our methods for scientific computing. Hopefully, it seems that it is now converging towards the norm. We have seen cases in which porting a scientific code to a computer with a different arithmetic produces dramatic changes.

Another problem is to know how to deal with the other mathematical operations (like the ones in `<math.h>` in C). In general we have to know the algorithm or its specifications (sometimes given by library providers). The problem of having "good" libraries is well-known in the literature, as the "Table makers' dilemna". In this article we will stick to the core of the norm, and consider only "simple" operations.

2.1 Examples and properties of interest

Our aim is to be able to analyze at compile-time the way floating-point operations are used or mis-used.

What we intend to automatically find is at least the exceptions that might be raised (and not caught), like "Overflow", "Underflow" and "NaN". This could be handled with other well-known analyses (interval analysis as used in Syntox [Bou92], polyhedra [CH78] etc.) so we will not describe this part so much. What we really would like to find is some not too pessimistic information about the precision of the values of the variables. This leads to estimates of branching reliability in tests and in expecting to partially solve some difficult termination problems (see Example 1).

Example 1 *Consider the expression* $x = \frac{c_1 b_2 - c_2 b_1}{a_1 b_2 - a_2 b_1}$ *which leads on an UltraSparc in simple precision, for* $c_1 = 0$, $c_2 = 1$, $b_1 = -46099201$, $b_2 = -35738642$, $a_1 = 37639840$ *and* $a_2 = 29180479$, *to* $x = 1046769994$ *(the true result is* $x = -46099201$*). This is an example of a problem known as "cancellation". The control flow might be wrong after this instruction, if it were followed by the (somewhat unlikely!) instructions:*

```
if (x==-46099201) { ... }
else { ... }
```

or non-termination could happen since this could be the termination test of a loop.

Here are some simple (and classic) examples of stable and unstable numerical computations:

Example 2 *Consider the following two implementations of the computation of the nth power of the gold number* ($g = \frac{\sqrt{5}-1}{2}$)*. The first one on the left (program (A)) relies on the simple property that if* u_n *is the nth power of the gold number,* $u_{n+2} = u_n - u_{n+1}$*. The second one, on the right hand side (program (B)), is the brute force approach.*

```
main()
{ float x,y,z;
  int i;
  x=1;
  y=(sqrt(5)-1)/2;
  for (i=1;i<=20;i++) {
    z=x;
    x=y;
    y=z-y;
    printf("phi^%d=%f\n",i,x); } }
```

```
main()
{ float t;
  int i;
  t=1;
  for (i=1;i<=20;i++) {
    t=t*(sqrt(5)-1)/2;
    printf("phi^%d=%f\n",i,t); } }
```

Program (A) gives the following results:

```
phi^1=0.618034                          phi^11=0.005026
phi^2=0.381966                          phi^12=0.003103
phi^3=0.236068                          phi^13=0.001923
phi^4=0.145898                          phi^14=0.001180
phi^5=0.090170                          phi^15=0.000743
phi^6=0.055728                          phi^16=0.000437
phi^7=0.034442                          phi^17=0.000306
phi^8=0.021286                          phi^18=0.000131
phi^9=0.013156                          phi^19=0.000176
phi^10=0.008130                         phi^20=-0.000045
```

Which of course does not make much sense! The fact is that the numerical scheme used on program (A) is not well-conditioned, meaning that it is very sensitive to the initial inaccuracy. In fact the initial inaccuracy on the computation of $(\sqrt(5)-1)/2$ which is of the order of $ulp(1)$ at most, is increased at each iteration and becomes more important than the real result.

Program (B) leads to the following results,

```
phi^1=0.618034                          phi^11=0.005025
phi^2=0.381966                          phi^12=0.003106
phi^3=0.236068                          phi^13=0.001919
phi^4=0.145898                          phi^14=0.001186
phi^5=0.090170                          phi^15=0.000733
phi^6=0.055728                          phi^16=0.000453
phi^7=0.034442                          phi^17=0.000280
phi^8=0.021286                          phi^18=0.000173
phi^9=0.013156                          phi^19=0.000107
phi^10=0.008131                         phi^20=0.000066
```

Which is in fact completely acceptable. Take now program (C) below which looks like program (A) (at least it does not look simpler):

```
x=1;
y=-1.0/3.0;
for (i=1;i<=20;i++) {
  z=x;
  x=y;
  y=(x+z)/6; }
```

The results that are computed are accurate (they are roundings of $\left(-\frac{1}{3}\right)^n$):

```
phi^1=-0.333333                         phi^11=-0.000006
phi^2=0.111111                          phi^12=0.000002
phi^3=-0.037037                         phi^13=-0.000001
phi^4=0.012346                          phi^14=0.000000
phi^5=-0.004115                         phi^15=-0.000000
phi^6=0.001372                          phi^16=0.000000
phi^7=-0.000457                         phi^17=-0.000000
phi^8=0.000152                          phi^18=0.000000
phi^9=-0.000051                         phi^19=-0.000000
phi^10=0.000017                         phi^20=0.000000
```

This "numerical scheme" is well-conditioned, i.e. stable.

2.2 A language

In the language we consider in this paper, we confine ourselves to simple floating-point operations (which are fully specified in the IEEE754-1985 norm), with one type of floating-point number only (no double precision nor cast here),

$$
\begin{array}{lcll}
\texttt{Expr} & = & \texttt{cste} & \text{constant real expression} \\
& & X & \text{variable } X \in \texttt{Var} \\
& & \texttt{Expr} + \texttt{Expr} & \text{sum} \\
& & \texttt{Expr} * \texttt{Expr} & \text{product} \\
& & \texttt{Expr} - \texttt{Expr} & \text{difference} \\
& & \texttt{Expr}/\texttt{Expr} & \text{division} \\
& & \sqrt{\texttt{Expr}} & \text{square root} \\
& & (\texttt{Expr}) & \text{bracketing}
\end{array}
$$

The idea is that the evaluation of arithmetic expressions is determined by the syntax (left to right, innermost to outermost evaluation here). We confine ourselves in this paper to very simple test expressions, as follows,

$$
\begin{array}{lcll}
\texttt{test} & = & X == 0 & \text{zero} \\
& & X > 0 & \text{strict positivity} \\
& & X \geq 0 & \text{positivity}
\end{array}
$$

Instructions are,

$$
\begin{array}{lcll}
\texttt{Instr} & = & X = \texttt{Expr} & \text{assignment for } X \in \texttt{Var} \\
& & \texttt{if test then block else block} & \text{conditional statement} \\
& & \texttt{while test block} & \text{while loop}
\end{array}
$$

We have used in the examples "equivalent" C forms of a program in that syntax.

Blocks of instructions are concatenations of instructions,

$$
\begin{array}{lcll}
\texttt{block} & = & \emptyset & \text{empty block} \\
& & \texttt{Instr}; \texttt{block} & \text{block concatenation}
\end{array}
$$

Finally a program `P` is just a block.

2.3 A (almost) standard concrete semantics

We plunge floating-point numbers (parameterized here by N and K, the length of the binary words representing respectively the mantissa and the exponent) into Val which is the union of the (mathematical) real numbers $\mathbb{R}$ extended with values $\{\infty, -\infty, NaN, \omega, \upsilon, \delta, \sigma\}$ which stand respectively for $+\infty$ and $-\infty$ (mathematical infinities, coming from a compactification of the set of reals for instance), NaN, a special element denoting the value "not a number", and ω denoting overflow, υ, underflow, δ, division by zero error and σ is the error resulting from taking the square root of a strictly negative number. The semantics is given as a transition system, where states are elements of $Ctrl \times Env$ where $Env = Var \to Val$ and $Ctrl$ is the text of the program yet to be executed. The semantics also depends on the round-off mode $\mathcal{M} : Val \to Val$ (a partial function[4]) and on the use (or not) of some standard handlers in case of overflow, taken care of by a (partial) function $\mathcal{E} : Val \to Val$. By convention, all our (partial) functions (if not otherwise stated) will not be defined on "errors" ω, υ, δ and σ, nor on NaN and will be the identity on ∞ and $-\infty$. For the sake of simplicity, we will consider only normalized floating-point numbers and will not use signed NaN nor signed zero. We re-define now the following mathematical functions acting on Val,

- We "overload" the exponent function we had at lemma 1; $k : Val \to Val$ is the exponent (partial) function with, $k(\infty) = k(-\infty) = \infty$, $k(x) = max(\lfloor log_2(\mid x \mid) \rfloor, 2 - 2^K)$ if $x \in \mathbb{R}$, $x \neq 0$, $k(0) = 0$, $k(x) = \perp$ (i.e. not defined) in all other cases. This enables us to have the right f (as in lemma 1) function and thus the wright underflow mechanism.

[4] We write in an equivalent manner $\mathcal{M}(x) = \perp$ and $\mathcal{M}(x)$ undefined.

- $\mathcal{M}(x) = s\left(\frac{\lfloor 2^N f(x) \rfloor}{2^N} + 1\right) 2^{k(x)}$ (this is the rounding towards zero mode, which we write when there is a risk of ambiguity $\mathcal{M}_0$), other modes include: $\mathcal{M}(x) = s\left(\frac{s\lceil 2^N s f(x) \rceil}{2^N} + 1\right) 2^{k(x)}$ (rounding towards plus infinity or $\mathcal{M}_+$), and $\mathcal{M}(x) = s\left(\frac{s\lfloor 2^N s f(x) \rfloor}{2^N} + 1\right) 2^{k(x)}$ (rounding towards minus infinity or $\mathcal{M}_-$),
- $\mathcal{E}(x) = O$ if $\mid x \mid > 2^{2^K+1} - 2^{2^K-N}$, $\mathcal{E}(x) = U$ if $\mid x \mid < 2^{2-2^K}$ (so that we are not dealing here with "gradual underflow" or de-normalized numbers), otherwise $\mathcal{E}(x) = x$ (this is the "no handler" option).

We look at the semantics of an expression `Expr`. Given $\rho \in Env$,

$$
\begin{array}{lcl}
[\![\mathtt{cste}]\!]^f \rho & = & \mathcal{E} \circ \mathcal{M}(\mathtt{cste}) \\
[\![X]\!]^f \rho & = & \rho(X) \\
[\![\mathtt{Expr}_1 + \mathtt{Expr}_2]\!]^f \rho & = & [\![\mathtt{Expr}_1]\!]^f \rho +^f [\![\mathtt{Expr}_2]\!]^f \rho \\
[\![\mathtt{Expr}_1 * \mathtt{Expr}_2]\!]^f \rho & = & [\![\mathtt{Expr}_1]\!]^f \rho *^f [\![\mathtt{Expr}_2]\!]^f \rho \\
[\![\mathtt{Expr}_1 / \mathtt{Expr}_2]\!]^f \rho & = & [\![\mathtt{Expr}_1]\!]^f \rho /^f [\![\mathtt{Expr}_2]\!]^f \rho \\
[\![\mathtt{Expr}_1 - \mathtt{Expr}_2]\!]^f \rho & = & [\![\mathtt{Expr}_1]\!]^f \rho -^f [\![\mathtt{Expr}_2]\!]^f \rho \\
[\![\sqrt{\mathtt{Expr}}]\!]^f \rho & = & \sqrt{[\![\mathtt{Expr}]\!]^f}^f
\end{array}
$$

where the functions $+^f$, $*^f$, $/^f$, $-^f$ and $\sqrt{\ }^f$ are defined as follows,

- $a +^f b = \mathcal{E} \circ \mathcal{M}(a + b)$
- $a *^f b = \mathcal{E} \circ \mathcal{M}(ab)$
- $a -^f b = \mathcal{E} \circ \mathcal{M}(a - b)$ if a and b are not both the same infinity. In the latter case, $a -^f b = NaN$.
- $a /^f b = \mathcal{E} \circ \mathcal{M}(\frac{a}{b})$ if $b \neq 0$. If $b = 0$ then $a /^f b = \delta$.
- $\sqrt{a}^f = \mathcal{E} \circ \mathcal{M}(\sqrt{a})$ if $a \geq 0$ otherwise $\sqrt{a}^f = \sigma$.

Assignments have the following semantics: $[\![X = \mathtt{Expr}]\!]^f \rho = \rho[X \leftarrow [\![\mathtt{Expr}]\!]^f \rho]$ where $\rho[u \leftarrow v]$ denotes the new environment in which $\rho(u)$ is now equal to v, whereas all other variables are mapped to the value they had by ρ. Tests are also quite straightforward ($[\![\mathtt{test}\ ]\!]_f$ is a boolean value indicating whether the test is true or not). Transitions from state $(\mathtt{Instr}; Prog, \rho)$ to $(Prog, \rho')$ are now rather easy to write down, given the evaluation of expressions above. We spare the reader the details, given that this is rather standard (in SOS style [Plo81] for instance). In order to be able to write the abstract semantics in an easier manner in the sequel, we suppose that all expressions are decomposed into sequences of single operations (like $+$, $-$ etc. respecting the evaluation strategy). For instance, the assignment `x=y*z+2` will be supposed to be decomposed using an auxiliary variable `t` as `t=y*z; x=t+2`. This refines the transition system described above by splitting the transition representing the evaluation of a (complex) expression into a sequence of transitions, one for each simple floating-point operation.

Notations In the sequel, operations $+^f$, $-^f$ etc. (respectively $+$, $-$ etc.) will have to be understood as the floating-point (respectively "real") operations. We will also introduce new operations $\oplus$, $\ominus$ etc. (next section) and $+^a$, $-^a$ etc., that are "abstractions" of these operations.

3 Abstract domains

A correct (in the sense of abstract interpretation) domain for abstracting the semantics above is given by intervals of floating-point numbers (in the style of F. Bourdoncle's Syntox integer interval analyzer). Basically the "best" correct abstract operations (forward semantics) are:

- the abstraction of the + operation is $[a,b] \oplus [c,d] = [\mathcal{M}'_-(a +^f c), \mathcal{M}'_+(b +^f d)]$
- for subtraction: $[a,b] \ominus [c,d] = [\mathcal{M}'_-(a -^f d), \mathcal{M}'_+(b -^f c)]$
- for multiplication: $[a,b] \otimes [c,d] = [\mathcal{M}'_-(min(a *^f c, a *^f d, b *^f c, b *^f d)), \mathcal{M}'_+(max(a *^f c, a *^f d, b *^f c, b *^f d))]$
- for the inverse (here $d \geq c > 0$): $inv^o([c,d]) = [\mathcal{M}'_-(1/^f d), \mathcal{M}'_+(1/^f c)]$
- for the square root: $\sqrt{[c,d]}^o = [\mathcal{M}'_-(\sqrt{c}^f), \mathcal{M}_+(\sqrt{d}^f)]$ (when $c, d \geq 0$)
- For tests, one should be cautious with the rule for strict positivity: $[\![X > 0]\!]\rho = (\rho(X) \geq 2^{2-2^K})$.

where $\mathcal{M}'$ is the rounding function on the analyzer's internal representation of floating-point numbers, $+^f$, $-^f$ etc. are the floating-point operations on the target architecture where the program which is statically analyzed should be run, and K is the corresponding number of bits used for representing exponents. In general, we can (should?) hope for the analyzer to have a better precision than the target architecture[5], and in that case we can simplify the rules above (forgetting about the $\mathcal{M}'$ in the right-hand side of the definitions). This semantics has been implemented as part of the abstract domains used in the static analyzer TWO (ESPRIT project 28940), but is obviously very unsuitable for having a precise information on floating-point operations. Actually, we used an even less precise semantics in that we supposed we did not know the rounding mode in the analyzed program, so we had to assume the worst case which is, for instance in the case of the abstraction of addition: $[a,b] \oplus [c,d] = [\mathcal{M}_-(a+b), \mathcal{M}_+(c+d)]$ where $\mathcal{M}$ is the rounding function corresponding to the target architecture (or a suitable approximation of).

The experiments (to be described in a forthcoming article) show that this kind of analysis behaves poorly on floating-point code. The figure of about ten percent of the lines (which use floating-point operations) of a code being signaled as potential run-time errors (over-pessimistic warnings about the possibility of getting to an erroneous state, like overflow, division by zero etc.) is not uncommon. Using this semantics though, we are able to find real "subtle" bugs such as for the program: `if (x>0) y=1/x*x`, where there might be a division by zero error[6] (for instance when $x = 2^{2-2^K}$). Also this abstract semantics is sufficient to get good estimates of the 20th iteration of program (B) of example 2. It is of order $[\rho_0 - 2^{-23}, \rho_0 + 2^{-23}]^{20}$ i.e. about $[6.61067063328 * 10^{-5}, 6.61072163724 * 10^{-5}]$. Also the numerical bug of the Patriot as explained in the introduction would certainly have been found by such interval analyzers, with correct floating-point semantics.

But interval semantics is always very conservative and pessimistic: it might even incorporate the error of computation of the analyzer itself ($\mathcal{M}'$)! Secondly, it aggregates in the abstract value both the magnitude of the expected result and the inaccuracy error. Also it does not take care of dependencies between the values and especially between the errors. For instance, $x - x$ will always lead in such abstractions to a strictly positive error except if x is a singleton interval (i.e. a constant). What we really need is a relational abstraction at least on inaccuracy values.

3.1 Domain of affine forms

The idea here is to trace instructions (or locations in the program) that create round-off errors. We associate with each location and variable the way this control point makes the variable lose precision. This is loosely based on ideas from affine arithmetic [VACS94] (used in simulation of programs, not in static analysis).

[5] One could actually use multi-precision numbers instead of IEEE 754 `double` or `extended double` types for representing intervals in the static analyzer.

[6] This is an example taken from a seminar by Alain Deutsch in 1998.

The abstract values (notwithstanding error values) are, $x = a_0 + a_1\epsilon_1 + \cdots + a_n\epsilon_n$, the ϵ_i are variables, intended to represent random values with range $]-ulp(1), ulp(1)[$[7], associated with each location (describing the loss of precision at that point), the a_i being in an abstract domain $\mathcal{A}$ (for example real or floating-point intervals) abstracting $\wp(\mathbb{R} \cup \{\infty, -\infty\})$, through (for instance) a Galois Connection [CC92a] $\wp(\mathbb{R} \cup \{\infty, -\infty\}) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \mathcal{A}$. Basically a_0 should be an abstraction of the intended result if the program was manipulating real numbers, and the a_i $(i \geq 1)$ represent abstractions of each small error due to the "ith operation" in the program.

Let us make this more precise by setting first $\mathcal{L}$, the set of all locations in the programs to be analyzed (i.e. all elements in $Ctrl$ in the concrete semantics given in section 2.3) $\mathcal{L} = \{\epsilon_i \mid i \in L\}$ that we will identify in the sequel with a subset of $\mathbb{N}$. The affine forms domain, parameterized by $\mathcal{A}$ and $\mathcal{L}$ is the domain $\mathcal{D}$ defined as follows, $\mathcal{D} = \{a_0 + \sum_{i \in L} a_i\epsilon_i \mid a_0, a_i \in \mathcal{A}, i \in L\}$ and the order is defined component-wise, $a_0 + \sum_{i \in L} a_i\epsilon_i \leq a'_0 + \sum_{i \in L} a'_i\epsilon_i$ if $a_0 \leq_{\mathcal{A}} a'_0$ and $a_i \leq_{\mathcal{A}} a'_i$ for all $i \in L$.

Therefore, if $\mathcal{A}$ is a lattice, then $\mathcal{D}$ is a lattice with component-wise operations. Similarly, widenings and narrowings [CC92b] defined in $\mathcal{A}$ can be extended in a component-wise manner to generate widenings and narrowings on $\mathcal{D}$. In general the classical widenings on intervals are not very subtle. We use the following family of widenings here: $[a, b]\nabla_k[c, d] = [e, f]$ with $\begin{cases} e = c - 2^k(a-c) & \text{if } c < a \\ e = a & otherwise \end{cases}$ and $\begin{cases} f = d + 2^k(d-b) & \text{if } d > b \\ f = a & otherwise \end{cases}$. This is only a widening if we suppose that the boundaries of our intervals use a finite precision arithmetic (bounded multi-precision for instance).

We can now define a concretization function $\Gamma : \mathcal{D} \to \wp(\mathbb{R} \cup \{\infty, -\infty\})$ by $\Gamma(a_0 + \sum_{i \in L} a_i\epsilon_i) = \gamma(a_0) + \sum_{i \in L} \gamma(a_i) *]-ulp(1), ulp(1)[$.

The problem is that there is no way we can hope for a very strong correctness condition (we will give it in detail in section 3.3) for an analysis based on $\mathcal{D}$ with respect to the concrete semantics given in section 2.3, because we have not specified in the concrete semantics what the "real" result should be. Therefore there is no best choice to what a_0 should be (hence the same problem holds for the a_i, $i \in L$)[8]. We should in fact have designed a non-standard concrete semantics that remembers the inaccuracy of the computations, which we shall see now.

3.2 A non-standard semantics

We slightly change the semantics of section 2.3 so that environments are now of the form ρ : `Var` $\to (Val \times Val)$ (we will write $\rho = (\underline{\rho}, \overline{\rho})$). So $\rho(X) = (\underline{X}, \overline{X})$ where X is any variable, and the intended meaning is that $\underline{X}$ is the semantics we had in section 2.3 and $\overline{X}$ is the intended "real" computation (i.e. using real numbers and not floating point numbers). For the sake of simplicity, we only carry on the concrete and abstract semantics without dealing with NaN nor run-time errors, hence dropping the $\mathcal{E}$ part of the semantics. For expressions for instance, we find the new concrete operators:

- $a +^f b = (\mathcal{M}(a+b), a+b)$
- $a *^f b = (\mathcal{M}(ab), ab)$
- $a -^f b = (\mathcal{M}(a-b), a-b)$
- $a /^f b = (\mathcal{M}(\frac{a}{b}), \frac{a}{b})$ if $b \neq 0$.

[7]Notice that if we assume the default rounding mode, we could actually use a smaller interval i.e. $[-ulp(1)/2, ulp(1)/2]$.

[8]Mathematically, suppose we have a corresponding abstraction $A : \wp(Val) \to \mathcal{D}$ making (A, Γ) into a Galois connection. Suppose for instance that $\mathcal{A}$ is the interval domain, and consider $\Gamma(u_1 = [0,0] + [1,1]\epsilon_1) =]-ulp(1), ulp(1)[$ and $\Gamma(u_2 = [0,0] + [1,1]\epsilon_2) =]-ulp(1), ulp(1)[$ as well. But $\Gamma(u_1 \cap u_2) = \Gamma([0,0]) = [0,0]$ is not equal to $\Gamma(u_1) \cap \Gamma(u_2) =]-ulp(1), ulp(1)[$, so there cannot be a left-adjoint to Γ.

The rest of the semantics is pretty much the same, "executing in parallel" the program with floating-point operations, and the program with operations in $\mathbb{R}$, but without observing precisely the steps in the "real" computation. For instance, we have a transition from $(x = \texttt{Expr}; Prgm, (\underline{\rho}, \overline{\rho}))$ to $(Prgm, ([\![x = \texttt{Expr}]\!]^f \underline{\rho}, [\![x = \texttt{Expr}]\!]\overline{\rho}))$ where $[\![.]\!]^f$ is the "floating-point" semantics given in section 2.3, and $[\![.]\!]$ is a similar semantics, but with operations and numbers in $\mathbb{R}$ (the "ideal semantics"). The more difficult part of the semantics is tests (also `while` loops of course since they include tests). The problem is that a test in the floating-point semantics might well not give the same result as the test in the real number semantics (as in example 1), leading to a different flow of execution in the two semantics. We choose in that case to stop computing the real number semantics: for instance there is a transition from $(if\ (x < 0)\ x = x + 1\ else\ x = x - 1, (x \leftarrow -10^{-37}, x \leftarrow 0))$ to $(x = x + 1, (x \leftarrow -10^{-37}, \perp))$ and then to $(\emptyset, (x \leftarrow 1, \perp))$[9]. This actually corresponds to a synchronized product [Arn92] (with synchronization between two transitions being only allowed when the two have the same labels) of the transition system corresponding to the floating-point semantics with another, corresponding to an "observer", which is the real number semantics.

From this semantics, we can construct an even more detailed semantics which will be our final non-standard semantics, and which goes (briefly) as follows. We define inductively the notion of "inaccuracy" coming from a location (i.e. a transition) that we identify with a "formal" variable ϵ_i. Consider a trace s of execution from an initial environment $(\underline{\rho}, \overline{\rho})$. Suppose this trace goes through locations ϵ_1 to ϵ_{j-1} and that variables x and y are computed on this trace; we suppose we can write formally (this is the induction step of the definition) $x = x_0 + \sum_{i=1}^{j-1} x_i \epsilon_i$ and $y = y_0 + \sum_{i=1}^{j-1} y_i \epsilon_i$ where ϵ_i is a formal variable of magnitude $\epsilon = ulp(1)$. x_0 (respectively y_0) is the value computed with the semantics of real numbers on s from $\overline{\rho}$. x_i is the magnitude (divided by ϵ) of the error of the computed result in the semantics of floating-point numbers starting at $\underline{\rho}$, due to the rounding operation at instruction ϵ_i. Then suppose we extend the trace s with operation $z = x.y$. This derived semantics computes $z = z_0 + \sum_{i=1}^{j} z_i \epsilon_i$ with z_i being a function of the x_k and y_l ($1 \le k \le j-1$ and $1 \le l \le j-1$). This semantics is left for the full version of the paper. Now we are going to abstract the coefficients z_i.

3.3 Abstract Semantics

We particularize $\mathcal{D}$ with $\mathcal{A}$ being the interval lattice (but this is easy to generalize on any non-relational abstract domain) and $\mathcal{L}$ being the set of locations (identified again with a subset of $\mathbb{N}$). We will only deal here with a forward abstract semantics that we call $[\![.]\!]^a$. Of course, having a backward abstract semantics would enable us to gain more precision during analysis, using iterates of forward and backward iterates [CC92a], but this is outside the scope of the paper.

The semantics of expressions is defined using operations $+^a$, $-^a$, $*^a$, inv^a, $\sqrt{\ }^a$ as follows: let $x = [a_0, b_0] + \sum_{i=1}^{n} [a_i, b_i]\epsilon_i$, $y = [c_0, d_0] + \sum_{i=1}^{n} [c_i, d_i]\epsilon_i$ be two affine forms. We are trying to find a good abstraction for an operation . on x and y at location j, giving the result $z = z_0 + \sum_{i=1}^{n} z_i \epsilon_i$. The abstract semantic functions are (using $\oplus$, $\otimes$ etc. of section 3, where we assume $\mathcal{M}' = \mathcal{M}$):

$$x +^a y \stackrel{def}{=} ([a_0, b_0] \oplus [c_0, d_0]) + \sum_{i=1}^{j-1} ([a_i, b_i] \oplus [c_i, d_i])\, \epsilon_i \ + (\alpha \circ \Gamma(x) \oplus \alpha \circ \Gamma(y))\epsilon_j$$

This merely translates the fact that the "real value" of the sum should be in the sum of the (floating-point) intervals containing the real values of x and y. The errors from ϵ_i must be over-approximated by the sum of the errors for computing x and y at ϵ_i. The last term (factor of ϵ_j) is due to the rounding of the "real" sum operation at ϵ_j. Because of the IEEE-754 standard, its magnitude is at most $ulp(z)$ where z is the floating-point sum of x and y. It is easy to see that it

[9] We say in that case that this test is "unstable".

is less or equal than $(\alpha \circ \Gamma(x) \oplus \alpha \circ \Gamma(y))\epsilon_j$. The other rules are:

$$
\begin{array}{rcl}
x -^a y & \stackrel{def}{=} & ([a_0, b_0] \ominus [c_0, d_0]) + \sum_{i=1}^{j-1} ([a_i, b_i] \ominus [c_i, d_i])\, \epsilon_i \;\; + (\alpha \circ \Gamma(x) \ominus \alpha \circ \Gamma(y))\epsilon_j \\
x \times^a y & \stackrel{def}{=} & [a_0, b_0] \otimes [c_0, d_0] + \sum_{i=1}^{j-1} (([a_i, b_i] \otimes \alpha \circ \Gamma(y)) \oplus (\alpha \circ \Gamma(x) \otimes [c_i, d_i]))\, \epsilon_i \\
& & \qquad\qquad\qquad + (\alpha \circ \Gamma(x) \otimes \alpha \circ \Gamma(y))\epsilon_j \\
inv(x)^a & \stackrel{def}{=} & inv^o([a_0, b_0]) - inv^o(\alpha \circ \Gamma(x) \otimes \alpha \circ \Gamma(x)) \otimes \sum_{i=1}^{j-1} [a_i, b_i]\epsilon_i + inv^o(\alpha \circ \Gamma(x))\epsilon_j \\
\sqrt{x}^a & \stackrel{def}{=} & \sqrt{[a_0, b_0]}^o + inv^o([2, 2] \otimes \sqrt{\alpha \circ \Gamma(x)}^o) \otimes \sum_{i=1}^{j-1} [a_i, b_i]\epsilon_i + \sqrt{\alpha \circ \Gamma(x)}^o \epsilon_j
\end{array}
$$

The correctness of this semantics with respect to the semantics of section 2.3 is expressed as follows; let x and y be two affine forms, and let $.^f$ (respectively h^f) be any of the operations $+^f$, $-^f$, $*^f$, (respectively inv^f, $\sqrt{}^f$) then $\Gamma(x) .^f \Gamma(y) \subseteq \Gamma(x .^a y)$ (respectively $\Gamma(h^f(x)) \subseteq h^a(\Gamma(x))$).

Addition and subtraction rules are easy, even if they are too approximate in fact because we always say that the operation might create an inaccuracy of up to one *ulp* around the result (which is at most, by the IEEE 754 norm of $ulp(\mid x.y \mid) \leq \mid x.y \mid ulp(1)$).

Let us show for instance the correctness of $\times^a$. Take $U = U_0 + \sum_{i=1}^{j-1} U_i \epsilon_i$ and $V = V_0 + \sum_{i=1}^{j-1} V_i \epsilon_i$ two affine forms, with $U_i = [a_i, b_i]$ and $V_i = [c_i, d_i]$. Let $u \in \Gamma(U)$ and $v \in \Gamma(V)$. We write $u = u_0 + \sum_{i=1}^{j-1} u_i \epsilon$ (where $\epsilon = ulp(1)$) and $v = v_0 + \sum_{i=1}^{j-1} v_i \epsilon$ where $u_i \in \gamma(U_i)$ and $v_i \in \gamma(V_i)$. We consider $u \times^f v = \mathcal{M}(uv)$. $uv = u_0 v_0 + \sum_{i=1}^{j-1} u v_i \epsilon + \sum_{i=1}^{j-1} v u_i \epsilon$. So

$$
\begin{array}{rcl}
\mathcal{M}(u \times v) & \leq & uv + \mid uv \mid \epsilon \\
& \leq & u_0 v_0 + \sum_{i=1}^{j-1} (u v_i + v u_i)\epsilon + \mid uv \mid \epsilon \\
& \leq & max([a_0, b_0] \otimes [c_0, d_0]) + \\
& & \quad \sum_{i=1}^{j-1} (max(\Gamma(U) \otimes [c_i, d_i] + [a_i, b_i] \otimes \Gamma(V)))\epsilon + max(\mid \Gamma(U)\Gamma(V) \mid)\epsilon
\end{array}
$$

Which shows one part of the inclusion. The rest is left to the reader.

For the formula for *inv*, the abstraction is correct since $x \rightarrow \frac{1}{x}$ is concave on its domain of definition, so it can be safely approximated on an interval $[a, b]$ by (for $x \in [a, b]$, $x + \delta \in [a, b]$):

$$
\frac{1}{x} - \frac{1}{a^2}\delta \leq \frac{1}{x + \delta} \leq \begin{cases} \frac{1}{a} - \frac{1}{ab}\delta & \text{if } ab > 0 \\ \frac{1}{b} - \frac{1}{ab}\delta & \text{if } ab < 0 \end{cases}
$$

Same proof with square root, but this time the function is convex so we approximate it on $[a, b]$ by:

$$
\sqrt{a} + \frac{1}{\sqrt{a} + \sqrt{b}}\delta \leq \sqrt{x + \delta} \leq \sqrt{x} + \frac{1}{2\sqrt{x}}\delta
$$

Of course, there are other ways to give lower and upper bounds to these computations. The choice we have taken is that the individual coefficients of the ϵ_i should reflect the magnitude of the error coming from the computation at ϵ_i in the total error on a trace of computation. Of course this depends on the "formal derivatives" that appear in these formulae.

The correctness of this semantics with respect to the semantics of section 3.2 is as follows. On the set of paths that $Prgm$ can execute from a set of initial environments, x (respectively y) has real value in $[a_0, b_0]$ (respectively $[c_0, d_0]$) and errors coming from location ϵ_i ($i \in \{1, \cdots, j - 1\}$) are in $[a_i, b_i]$ (respectively $[c_i, d_i]$), then on the set of paths that $Prgm; z = x.y$ can execute from the same environment, z has real value in z_0, and the error coming from location ϵ_i is in z_i plus the error in the computation of the operation . itself (represented by ϵ_j). The effect of adding the operation $z = x.y$ on the magnitude of the error coming from location ϵ_i ($i < j$) is reflected by the derivative of the operation in question.

We have not spoken of the abstract semantics of constants and tests. Constants are easy to abstract, the IEEE754-1985 rules dealing with constants are very precise and we can determine for sure whether we lose precision or not. Tests are more complex. We use local decreasing iterations as in [Gra92]. For instance suppose we want to interpret $x == y$. The corresponding abstract operator $==^a$ will be the greatest fixed point of the functional F on affine forms, which to every pair of affine forms ($x = x_0 + \sum_i x_i\epsilon_i, y = y_0 + \sum_i y_i\epsilon_i$) associates ($x' = x'_0 + \sum_i x'_i\epsilon_i, y' = y'_0 + \sum_i y'_i\epsilon_i$) with (each component of the functional is in $\mathcal{A}$, i.e. here the lattice of intervals)

$$\left\{\begin{array}{lcl} x'_0 & = & x_0 \cap (\Gamma(y -^a \sum_i x_i\epsilon_i)) \\ x'_i & = & x_i \\ y'_0 & = & y_0 \cap (\Gamma(x -^a \sum_i y_i\epsilon_i)) \\ y'_i & = & y_i \end{array}\right.$$

This is not the best abstraction (on the coefficients of ϵ_i) but it is enough to show that some tests might be unstable (when the order of magnitude of the x_i or y_i is not negligible with respect to x_0 and y_0 respectively).

4 An example

Let us decorate now the different floating-point operations for program (A):

```
x=1;
y=(sqrt(5)!1! -1 !2!)/2 !3!; !4!

for(i=1;i<20;i++){
  z=x;
  x=y;
  y=z-y; !5!
}
```

The semantics using affine forms[10] goes as follows; first for the locations before the loop:

$$\begin{array}{rcl} !1! : \sqrt{5} & = & [2.236068, 2.236069] + [2.236068, 2.236069]\epsilon_1 \\ !2! : \sqrt{5}-1 & = & [1.236068, 1.236069] + [2.236068, 2.236069]\epsilon_1 + [1.2360676, 1.2360684]\epsilon_2 \\ !3! : \dfrac{1}{2} & = & [0.5, 0.5] + [0.5, 0.5]\epsilon_3 \\ !4! : \dfrac{\sqrt{5}-1}{2} & = & [0.618033936, 0.618034058] + [1.118033936, 1.118034058]\epsilon_1 \\ & & +[0.6180338176, 0.6180342272]\epsilon_2 + [0.6180340736, 0.6180342784]\epsilon_3 \\ & & +[0.6180340736, 0.6180342784]\epsilon_4 \\ & = & y \end{array}$$

(Notice that $\frac{1}{2}$ is blindly over-approximated. This could be done exactly in a more refined semantics). We can then look at the abstract values on the first unfolding of the loop. For instance, in the first loop we find the abstract value for y to be:

$$[0.381966004, 0.381966126]+[-1.118033935, -1.118033813]\epsilon_1+[-0.6180341760, -0.6180337664]\epsilon_2+$$
$$[-0.6180342272, -0.6180340224]\epsilon_3 + [-0.6180342272, -0.6180340224]\epsilon_4+$$
$$[0.3819656448, 0.3819658240]\epsilon_5$$

[10] This comes from a library programmed in C by Nicolas Regal in 1999 [Reg99].

(concretization is [0.38196568,0.38196583]). Then,

$$[0.236067780, 0.236068024] + [2.236067872, 2.236068116]\epsilon_1 + [1.2360676352, 1.2360684544]\epsilon_2+$$
$$[1.2360681472, 1.2360685568]\epsilon_3 + [1.2360681472, 1.2360685568]\epsilon_4+$$
$$[-0.1458974464, -0.1458969344]\epsilon_5$$

(concretization is [0.23606846,0.23606873]) Then,

$$[0.145897995, 0.145898346]+[-3.354102050, -3.354101562]\epsilon_1+[-1.854102528, -1.8541012992]\epsilon_2+$$
$$[-1.854102528, -1.8541021184]\epsilon_3 + [-1.854102528, -1.8541021184]\epsilon_4+$$
$$[0.2917938944, 0.2917948160]\epsilon_5$$

(concretization is [0.14589696,0.14589734]) Then,

$$[0.090169434, 0.090170029] + [5.590169434, 5.590170411]\epsilon_1 + [3.090168832, 3.0901712896]\epsilon_2+$$
$$[3.0901702656, 3.0901714944]\epsilon_3 + [3.0901702656, 3.0901714944]\epsilon_4+$$
$$[-0.2016236672, -0.2016221184]\epsilon_5$$

(concretization is [0.09017118,0.09017179]) Then again (the fifth time we go around the loop):

$$[0.055727959, 0.055728913]+[-8.944272460, -8.944270507]\epsilon_1+[-4.9442738176, -4.9442693120]\epsilon_2+$$
$$[-4.9442742272, -4.9442717696]\epsilon_3 + [-4.9442742272, -4.9442717696]\epsilon_4+$$
$$[0.2573515008, 0.2573540352]\epsilon_5$$

(concretization is [0.05572883,0.05573179]). We see that the coefficients of the ϵ_i up to $i = 4$ get bigger and bigger as the expected value gets smaller and smaller. The subtraction in control point 5 does not lose much precision as such. This means the loop magnifies the initial error of computation of y at each turn. This is an example of bad-conditioning. For the well-conditioned example computing $(-1/3)^n$, the computation with affine forms would show there is no problem.

Of course, in general we cannot unfold loops like that in a static analyzer. After some number of un-foldings, we use our widening operator, which would predict a huge potential loss of precision. We need better widening operators in general.

5 Improvements

5.1 Affine interval transformations

The idea is to consider that the semantics creates dependencies between the a_i coefficients (due to an inaccuracy at location i) that we can approximate by linear dependencies. This choice is motivated by the fact that a great deal of numerical codes compute affine operations (also quadratic sometimes in finite elements methods). It is also motivated by the fact that we know in general how to linearize errors, and we know how to manipulate affine constraints (which are used for instance in [Kar76] in static analysis).

We call T an affine transformation on the space generated by $\{\epsilon_1, \cdots, \epsilon_n\}$ if there exists a $n \times n$ matrix A, and an n-dimensional vector B such that for all vectors X, $T(X) = AX + B$. We can represent such a transformation by the pair (A, B). We abstract a set of affine transformations by abstracting all its elements in A and B by an element of $\mathcal{A}$. In fact the semantics of section 3.2 gives a set of such transformations each over-approximating the effect of each trace.

For instance, setting $\mathcal{A}$ to be the interval domain,

$$\alpha\left(\left\{\left(\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix}\right), \left(\begin{pmatrix} 1 & 3 \\ 0 & 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 5 \end{pmatrix}\right)\right\}\right) = \left(\begin{pmatrix} [1,1] & [2,3] \\ [0,0] & [1,2] \end{pmatrix}, \begin{pmatrix} [3,3] \\ [4,5] \end{pmatrix}\right)$$

So the abstract domain of affine error dependence $\mathcal{T}$ is isomorphic to $\mathcal{A}^{n^2+n}$ with componentwise ordering. This means that as for affine intervals, if $\mathcal{A}$ is a lattice, then $\mathcal{T}$ is a lattice with intersection and union computed pointwise etc.

The concretization function G goes from $\mathcal{T}$ to the set of all (concrete) affine transformations. $G(A' = (a'_{i,j})_{1 \geq i,j \leq n}, B' = (b'_j)_{1 \leq j \leq n})$ is the set of affine transformations $(A = (a_{i,j})_{1 \geq i,j \leq n}, B = (b_j)_{1 \leq j \leq n})$ with $a_{i,j} \in \gamma_{\mathcal{A}}(a'_{i,j})$ (for all $1 \leq i,j \leq n$) and $b_j \in \gamma_{\mathcal{A}}(b'_j)$ (for all $1 \leq j \leq n$).

What is important to see now is that these abstract affine transformations act on elements of $\mathcal{D}$, because we can use the semantics of the operations $+$, $*$ in $\mathcal{A}$ to compute a safe approximation of $\{AX + B / (A,B) \subseteq G(A', B'), X \in G(X')\}$.

For instance, the instruction $x + y = z$ at instruction j will be written in matrix form as:

$$
\begin{array}{c}
 \\ \epsilon_0(x) \\ \dots \\ \dots \\ \epsilon_n(x) \\ \epsilon_0(y) \\ \dots \\ \dots \\ \epsilon_n(y) \\ \epsilon_0(z) \\ \dots \\ \dots \\ \epsilon_n(z) \\ \epsilon_j(z)
\end{array}
\begin{array}{c}
\begin{array}{ccccccccc}
\epsilon_0(x) & \dots & \dots & \epsilon_n(x) & \epsilon_0(y) & \dots & \dots & \epsilon_n(y) & \epsilon_j
\end{array} \\
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma \circ \Gamma(x) \oplus \gamma \circ \Gamma(y)
\end{pmatrix}
\end{array}
$$

We do not write the other rules since they are the transcription of what we have seen for affine interval forms, on affine interval matrices.

5.2 Principle of the improvement

Let $\mathcal{X}$ be the product of the domain $Var \rightarrow \mathcal{D}$ with $\mathcal{T}$. An abstract value in $\mathcal{X}$ is a pair $(f = \lambda x.a_0(x) + \sum_{i \in L} a_i(x)\epsilon_i, (A, B))$ which describes an abstract state at some location ϵ_j for which the value of variable x is in $\gamma(a_0(x))$ plus inaccuracy errors of order $\gamma(a_i(x))$ coming from control point i, together with the abstract affine transformation approximating the way these inaccuracy errors have been transformed by the instructions just before location ϵ_j. This extra information added to f would not be necessary if the control flow of the program we are analyzing was acyclic. It is only when we need infinite least fixed point iterations that we can benefit from the approximation of the transformation of errors that take place at certain control points (given by (A, B)) to widen the iterations. So in practice, the abstract affine transformations will be managed at some suitable widening points as defined for instance in [Bou90, Bou93] (heads of loops, return sites in case of inter-procedural analysis of mutually recursive functions).

We use this extra-information for getting better widening operators. In fact we approximate the abstract affine transformation by a transformation that multiplies by an upper approximation of the spectral radius of the transformation (A, B). We can then look at the asymptotic value: $lim_{n \rightarrow \infty} A^n X_0 + \frac{A^{n+1} - Id}{A - Id} B$.

Unfortunately, most of the 'interesting properties" that we might want to compute on $G(A', B')$ are NP-complete. Among these interesting properties are, the property of having all the transformations invertible, or the determination of the spectrum of all the transformations ("spectral portrait"). So we need to approximate further, so that we can compute in an efficient manner a good upper approximation of the spectral radius of (A, B).

"Any" norm on matrices can be used to determine an approximation of the spectral radius. If $A = (a_{i,j})_{1 \leq i,j \leq n}$, $\|A\|_1 = \sum_{i,j} \mid a_{i,j} \mid$, $\|A\|_2 = \sqrt{\sum_{i,j} \mid a_{i,j} \mid^2}, \cdots$, $\|A\|_\infty = \max \{\mid a_{i,j} \mid\}$ can all be used because, the maximal norm of its eigenvalues is $\lambda_{max} \leq ||A||$ ($n||A||$ in the last case). This is not a very precise though.

There are in fact better algorithms to calculate this spectral radius. A very famous method (iterative power) is as follows. Let $A = (a_{i,j})_{1 \leq i,j \leq n}$ be a matrix, u any non-null vector, and $(q_k)_{k \geq 0}$ the sequence, $q_0 = \frac{u}{||u||_2}$, $\cdots$ $q_k = \frac{Aq_{k-1}}{||Aq_{k-1}||_2}$, $\cdots$ q_k converges (when $k \to +\infty$) towards the greatest norm of the eigenvalues of A. Unfortunately this is not of much use in an abstract calculus since we do not know if any of the iterates are upper-approximations of the spectral radius. We have only a weaker result, about the convergence of the (q_k) sequence.

There is a better approximation, which only uses "lattice-theoretic" notions. It is called the "Gerschgorin discs" and is based on an old result by the french mathematician J. Hadamard.

Lemma 2 *Let $A = (a_{i,j})_{1 \leq i,j \leq n}$ be a matrix. The spectrum of A is contained in $D_1 \cap D_2$ (in the complex plane) with,*

- $D_1 = \cup_{1 \leq i \leq n} D_{1,i}$,
- $D_2 = \cup_{1 \leq j \leq n} D_{2,j}$,
- $D_{1,i}$ *is the circle with center* $a_{i,i}$, *radius* $r_{1,i} = \sum_{1 \leq j \leq n, j \neq i} \mid a_{i,j} \mid$,
- $D_{2,j}$ *is the circle with center* $a_{j,j}$, *radius* $r_{2,j} = \sum_{1 \leq i \leq n, i \neq j} \mid a_{i,j} \mid$.

A good approximation of the biggest absolute value of the eigenvalues of *real* A is thus $G(A)$,

$$\max \{ \mid a_{i,i} \mid + \max \{r_{1,i}, r_{2,i}\} / 1 \leq i \leq n \}$$

Consider again the example 2 and in particular program (A). The analysis using affine transformations basically discovers something that numericians are used to. At each loop the errors on x, y et z are given by the affine transformation,

$$\begin{pmatrix} \delta'_x \\ \delta'_y \\ \delta'_z \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_y \\ \delta_z \end{pmatrix} + \begin{pmatrix} 0 \\ \alpha \\ 0 \end{pmatrix}$$

with at the first iteration of the loop $\delta_y = \epsilon = 2^{-23}$, and α is the error due to the rounding of operation $-$ in the loop.

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

In fact, matrix A which comes from the semantics is slightly bigger, but is quite redundant. We find $G(A) = 2$ (instead of $1.6180\cdots$).

Then we have to notice that $\alpha \leq 2^{-24}$ (in fact it is of order 2^{-23-i}). In fact the affine transformation has two non-null eigenvalues $\rho_1 = \frac{-1+\sqrt{5}}{2}$ and $\rho_0 = \frac{-1-\sqrt{5}}{2} < -1$. Therefore the error at the 20th iteration of order $\frac{\epsilon}{\rho_1 - \rho_0}\rho_1^{20} + \frac{|\rho_1^{21}|-1}{\rho_1 - 1}\alpha$ (about $4 * 10^{-4} + 2.4 * 10^{-3}$) bigger than ρ_0^{20} (about $6.6 * 10^{-5}$).

Of course to do this we need a reduced product with at least an interval analysis on the integer variables (to determine the right number of loops). This will not be described here.

Note that program (C) has $G(A) = 1$ (instead of $1/2$ which is the exact greatest eigenvalue), hence the inaccuracy does not increase at each iteration of the loop.

6 Related work

To our knowledge, there are two main types of tools that are used to help the programmer compute with floating-point operations (see [BMMM95, Mul89] for general references).

The first type of tool uses alternative arithmetic implementations to better match the "ideal" semantics of reals. For instance, interval arithmetic [GL70, Moo79] implements a real number as an interval of floating-point numbers containing it. Multi-precision arithmetic [KJ93, Sco89] uses variable length floating-point numbers to approximate at best real number computations. Rational arithmetic [Bak75, Cle74, HCL+68, Kla93, KM83, KM88, KM89, Sch83] implements exact arithmetic for rational numbers, which can be used to approximate real numbers, for instance using continued fractions [KM85, RT73, Sei83, Vui90]. These methods do not solve the problem in all cases; interval arithmetic might lead to very imprecise (but true) results[11], multi-precision arithmetic and rational arithmetic may be very costly to perform on real scientific codes[12].

A new and promising line of research uses domain theory and fractal encoding of exact real numbers [ES98, EP97, Eda97]. We do not know yet how we can use these ideas for static analysis purposes.

Another approach is at the heart of the CADNA software [Che95]. It is known as the perturbation method (CESTAC) or stochastic arithmetic [CV88, CV92]. The idea is that round-off errors can be modeled as quasi-Gaussian distributions of some sort, and that a simple statistic test (Student test) can estimate its parameters, thus enabling to give better approximations of real number computations. This method and tool is probably one of the most favored among the code programmers, since it is quite precise. But it has lead to some criticisms, see [Kah91] for example. We review some of its central ideas more in detail in section 6.2. A similar approach (still stochastic but in a "backwards" manner) has been implemented and tested under the name "Precise" [CCT00]. We thought of using similar probabilistic ideas as a basis of a static analysis but at the moment there seems to be no way to ensure (even probabilistically) the correctness of the approach. Very important work is being carried out on the foundations of probabilistic abstract interpretation [Mon00, Mon01] which we might use for this purpose later on.

In some applications (image synthesis etc.), some algorithmic geometry has been specifically designed, like for CGAL, [BCD+99] in order to use in a very controlled manner the IEEE 754 floating-point numbers for some computations. We do not know yet how to use this for our purposes.

We know of only one example of a static analyzer (such as we are discussing in this article), that not only tries to give more precise results on one execution, or give some hint about the precision on one execution, but rather assesses a property valid for all (or a large class of) executions. This analyzer [ACFG92] actually uses abstract interpretation. We will explain the abstraction chosen more in detail in next section.

6.1 Another abstract interpretation

The abstract interpreter [ACFG92] is based on the following underlying concrete model. Floating-point numbers are identified with a pair $f = \langle m, e\rangle$ with, a mantissa $m \in M = \{m \in Z/p \in \mathbb{N}, -(10^p - 1) \leq m \leq 10^p - 1\}$, and an exponent $e \in E = \{e \in Z/q \in \mathbb{N}, -q \leq e \leq q\}$.

Abstract values are $f = \langle s, p, em, eM\rangle$ where, $s = +, -, +/-, \perp$ is an abstract sign, p is an integer representing the "number of significative digits" and $[em, eM]$ is the "interval of exponents". For instance $< +, 4, 6, 12 >$ denotes in the concrete model,

$$F' = \{0.1000 * 10^6, 0.1001 * 10^6, \cdots, 0.9999 * 10^{12}\}$$

[11] See Section 3 for examples of that phenomenon.

[12] And they do not solve the problem in general, see [CC94] where the following classical dynamical system example by J. M. Muller $U_{n+1} = 111 - \frac{1130}{U_n} + \frac{3000}{U_n * U_{n-1}}$ with $U_0 = 2$ and $U_1 = -4$ leads with floating-point numbers of all (finite) precision – this includes multi-precision arithmetic – to 100 whereas the exact value for the limit is 6. This can only be solved by exact real numbers, i.e. infinite precision arithmetic!

It is proved to form a complete lattice with a Galois connection with the set of subsets of the concrete model. Abstract operations like *Add*, *Sub*, *Mul*, *Div* are defined. Unfortunately, no interpretation of tests nor of loops is made, thus greatly restricting the precision of the analysis. The analysis is sometimes even weaker than an interval analysis of floating-point numbers, thus much weaker than our analysis.

6.2 CADNA

The CADNA library implements the CESTAC method [CV88, CV92, PV74, VA85, Vig78, Vig87, Vig93]. The underlying model is that the round-off errors are of the form $2^{-p}.\alpha_i$ where the α_i are equi-distributed independent random variables on $]-1,1[$ with uniform law. This law is justified both experimentally and by some theoretical reasons [Knu73]. The perturbation methods goes as follows. At each floating-point operation, we perturbate the round-off towards $+\infty$ or $-\infty$ with the same probability. We execute N times each of the instructions of the program (in practice, $N = 2$ or 3). The mean value "converges" towards the exact mathematical result and the Student test computes a good approximation of the standard deviation of the quasi-Gaussian distribution law of the result.

Thus this method can be used both for improving the computations, and for testing the relevance of a given execution. It is very much used in practice but some bugs are known: its estimates are sometimes too optimistic.There are several reasons for which this may happen [Kah91]. The approximation by a Gaussian law is justified by the central limit theorem but the convergence is very slow on the tail of the distribution (in particular with $N = 2$ or 3). The approximation is very bad on un-probable events (precisely the ones which lead to very costly errors). In order to justify the use of the theorem, we also have to suppose that the round-off errors are random, not correlated, continuously distributed on a small interval. But in general, the errors are only due to a few round-off errors and to singularities (which we might find out by data perturbation but probably not by perturbation of the operations). So errors are not uncorrelated random variables. Also, the distribution of errors is a discrete distribution on floating-point numbers and not on continuous real number. Finally, this is only a "first order approximation": for multiplication and division, second-order terms are not considered, but it might occur that they are not negligible.

7 Conclusion and future work

We have presented some ideas about what static analysis can try to do for programs computing with floating-point numbers. Our first concern in the DAEDALUS project is to analyze control-command software (like the Patriot software seen in the introduction) which is not numerically intensive, and for which we think we should have good chances of finding nice solutions. Numeric-intensive software, like scientific codes, are much more complex. Some of them, such as well-conditioned problems, might be amenable to static analysis. More difficult is to consider what can happen for ill-conditioned problems (for example, inverse of a matrix with very small determinant). We believe that there is little hope an automatic tool can cope with such problems, but we would like to be shown to be wrong .

In fact, most scientific codes pose new problems to static analyzers. For instance, some codes rely on Monte Carlo methods, or more generally on randomized algorithms. We believe that the static analysis of such algorithms is an interesting prospect, but goes beyond the scope of this paper (see [Mon00, Mon01] for some ideas which could constitute a good basis for future work). What might have to be considered on these codes is that the random generators that are used are only pseudo-random generators. This complexifies again the semantic problem. Also some scientific codes use parallel algorithms which make things even more complex, especially regarding the evaluation order of floating-point operations; they will depend on actual synchronizations between tasks.

Our future work will not try to tackle these very subtle problems. The first extension we wish to make is to look at inverse problems i.e. at clever backwards semantics. The aim is to solve the following problem: what precision should we have on the input so that we reach a given precision level on the output. The last point is particularly important in the field of on-board software since the wrong estimate of the precision for the input of a control/command program can be very expensive (for instance, the cost of very precise sensors).

References

[ACFG92] Y. Ameur, P. Cros, J-J. Falcon, and A. Gomez. An application of abstract interpretation to floating-point arithmetic. In *Proceedings of the Workshop on Static Analysis*, 1992.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Masson, 1992.

[Bak75] G. A. Baker. *Essentials of Padé approximants.* Academic Press, New York, 1975.

[BCD+99] Jean-Daniel Boissonat, Frédéric Cazals, Frank Da, Olivier Devillers, Sjylvain Pion, François Rebufat, Monique Teillaud, and Mariette Yvinec. Programming with CGAL: The example of triangulations. In *Proceedings of the Conference on Computational Geometry (SCG '99)*, pages 421–422, New York, N.Y., June 13–16 1999. ACM Press.

[BMMM95] J. C. Bajard, D. Michelucci, J. M. Moreau, and J. M. Muller, editors. *Real Numbers and Computers — Les Nombres réels et l'Ordinateur*, 1995.

[Bou90] Francois Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, volume 456 of *LNCS*, pages 307–323. Springer-Verlag, 1990.

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735, 1993.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[CC94] F. Chaitin-Chatelin. Le calcul sur ordinateur à précision finie, théorie et état de l'art. Technical Report TR/PA/94/05, CERFACS, 1994.

[CCT00] F. Chaitin-Chatelin and E. Traviesas. Precise, a toolbox for assessing the quality of numerical methods and software. In *16th IMACS World Congress*, 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Che95] J. M. Chesneaux. *L'Arithmétique Stochastique et le Logiciel CADNA.* Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, November 1995.

[Cle74] C. W. Clenshaw. Rational approximations for special functions. In D. J. Evans, editor, *Software for Numerical Mathematics.* Academic Press, New York, 1974.

[CV88] J. M. Chesneaux and J. Vignes. Sur la robustesse de la méthode cestac. *Comptes Rendus de l'Académie des Sciences, Paris*, 307(1):855–860, 1988.

[CV92] J. M. Chesneaux and J. Vignes. Les fondements de l'arithétique stochastique. *Comptes-Rendus de l'Académie des Sciences, Paris*, 1(315):1435–1440, 1992.

[Eda97] A. Edalat. Domains for computation in mathematics, physics and exact real arithmetic, 1997.

[EP97] A. Edalat and P. J. Potts. Exact real computer arithmetic. Technical report, Department of Computing Technical Report DOC 97/9, Imperial College, 1997.

[ES98] A. Edalat and P. Sünderhauf. A domain-theoretic approach to real number computation, 1998.

[GL70] Donald I. Good and Ralph L. London. Computer interval arithmetic: Definition and proof of correct implementation. *Journal of the Association for Computing Machinery*, 17(4):603–612, October 1970.

[Gol91] D. Goldberg. What every computer-scientist should know about computer arithmetic. *Computing Surveys*, 1991.

[Gra92] P. Granger. Improving the results of static analyses of programs by local decreasing iterations. *Lecture Notes in Computer Science*, 652, 1992.

[HCL+68] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall. *Computer Approximations*. Wiley, New York, 1968.

[Kah91] W. Kahan. The improbability of probabilistic error analyses for numerical computations. Technical report, University of Berkeley, 1991.

[Kah96] W. Kahan. Lecture notes on the status of ieee standard 754 for binary floating-point arithmetic. Technical report, Berkeley University, 1996.

[Kar76] M. Karr. Affine relationships between variables of a program. *Acta Informatica*, (6):133–151, 1976.

[KJ93] W. Krandick and J. R. Johnson. Efficient multiprecision floating point multiplication with optimal directional rounding. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 228–233, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[Kla93] S. Kla. *Calcul parallèle et en ligne des fonctions arithmétiques*. PhD thesis, Ecole Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France, February 1993.

[KM83] P. Kornerup and D. W. Matula. Finite-precision rational arithmetic: an arithmetic unit. *IEEE Transactions on Computers*, C-32:378–388, 1983.

[KM85] P. Kornerup and D. W. Matula. Finite precision lexicographic continued fraction number systems. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, Urbana, USA, 1985. IEEE Computer Society Press, Los Alamitos, CA. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[KM88] P. Kornerup and D. W. Matula. An on-line arithmetic unit for bit-pipelined rational arithmetic. *Journal of Parallel and distributed Computing*, Special Issue on Parallelism in Computer Arithmetic(5), 1988.

[KM89] P. Kornerup and D. W. Matula. Exploiting redundancy in bit-pipelined rational arithmetic. In M. D. Ercegovac and E. E. Swartzlander, editors, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 119–126, Santa Monica, USA, 1989. IEEE Computer Society Press, Los Alamitos, CA.

[Knu73] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[Mon00] David Monniaux. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)*, number 1824 in Lecture Notes in Computer Science. Springer-Verlag, 2000. © Springer-Verlag.

[Mon01] David Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs (extended abstract). In *28th Symposium on Principles of Programming Languages (POPL '01)*. Association for Computer Machinery, 2001. To appear.

[Moo79] Ramon E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[Mul89] J. M. Muller. *Arithmétique des Ordinateurs*. Masson, Paris, 1989.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus, 1981.

[PV74] M. La Porte and J. Vignes. Error analysis in computing. In *Information Processing 74*. North-Holland, 1974.

[Reg99] N. Regal. Petite analyse en nombres flottants. Technical Report DTA/LETI/DEIN/SLA/99-055, CEA, 1999.

[RT73] J. E. Robertson and K. S. Trivedi. The status of investigations into computer hardware design based on the use of continued fractions. *IEEE Transactions on Computers*, C-22:555–560, 1973.

[Sch83] C. W. Schelin. Calculator function approximation. *American Mathematical Monthly*, 90(5), May 1983.

[Sco89] M. Scott. Fast rounding in multiprecision floating-slash arithmetic. *IEEE Transactions on Computers*, 38:1049–1052, 1989.

[Sei83] R. B. Seidensticker. Continued fractions for high-speed and high-accuracy computer arithmetic. In *Proceedings of the 6th IEEE Symposium on Computer Arithmetic*, Aarhus, Denmark, 1983. IEEE Computer Society Press, Los Alamitos, CA.

[VA85] J. Vignes and R. Alt. An efficient stochastic method for round-off error analysis. In Miranker Willard and Toupin, editors, *Accurate Scientific Computations, Lecture notes in Computer Science 235*. Springer-Verlag, 1985.

[VACS94] M. Vincius, A. Andrade, J. L. D. Comba, and J. Stolfi. Affine arithmetic, March 1994.

[Vig78] J. Vignes. New methods for evaluating the validity of mathematical software. *Math. Comp. Simul. IMACS*, 20:227–249, 1978.

[Vig87] J. Vignes. Contrôle et estimation stochastique des arrondis de calcul. *AFCET Interfaces*, 54:3–10, 1987.

[Vig93] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comp. Simul.*, 35:233–261, 1993.

[Vui90] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8), 1990.