Strongly Typed Numerical Computations

Matthieu Martel

Laboratoire de Mathématiques et Physique (LAMPS)
Université de Perpignan Via Domitia
52 avenue Paul Alduy, France
matthieu.martel@univ-perp.fr

1 Introduction

It is well-known that numerical computations may sometimes lead to wrong results because of the accumulation of roundoff erros. Recently, much work has been done to detect these accuracy errors in floating-point computations [1], by static [2–4] or dynamic [5] analysis, to find the least data formats needed to ensure a certain accuracy (precision tuning) [6–8] and to optimize the accuracy by program transformation [9, 10]. All these techniques are used late in the software development process, once the programs are entirely written.

In this article, we aim at exploring a different direction. We aim at detecting and correcting numerical accuracy errors at software development time, i.e. during the programming phase. From a software engineering point of view, the advantages of our approach are many since it is well-known that late bug detection is time and money consuming. We also aim at using intensively used techniques recognized for their ability to discard run-time errors. This choice is motivated by efficiency reasons as well as for end-user adoptability reasons.

We propose a ML-like type system (strong, implicit, polymorphic [11]) for floating-point computations in which the type of an arithmetic expression carries information on its accuracy. We use dependent types [12] and a type inference algorithm which, from the user point of view, acts like ML [13] type inference algorithm [11] even if it slightly differs in its implementation. While type systems have been widely used to prevent a large variety of software bugs, to our knowledge, no type system has been targeted to address numerical accuracy issues in floating-point computations. Basically, our type system accepts expressions for which it may ensure a certain accuracy on the result of the evaluation and it rejects expressions for which a minimal accuracy on the result of the evaluation cannot be inferred. Note that we use a dependent type system. Consequently, the type corresponding to a function of some argument x depends on the type of x itself. Obviously, for undecidability reasons, any non-trivial type system rejects certain programs that would not produce run-time errors. We show that our type system does is able to type usual implementations of usual simple numerical algorithms [14] such as the trapeze rule, Simpsons rule, Horners scheme, Euler, Runge-Kutta, Newton Raphson algorithms, etc.

This article is organized as follows. Section 2 introduces informally our type system and shows how it is used in our implementation of a ML-like programming

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadruple precision	113	15	-16382	$+16383$

Fig. 1. Basic binary IEEE754 formats.

language. The formal definition of the types and of the type inference algorithm is given in Section 3. A correctness theorem is given in Section **??**. Finally, some use cases are described in Section **??** and Section **??** concludes.

2 Programming with Types for Accuracy

In this section, we present informally how our type system works. First of all, a floating-point number f in base β is defined by

$$f = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the significand, $0 \leq d_i < \beta$, $0 \leq i \leq p - 1$, p is the precision and e is the exponent, $e_{min} \leq e \leq e_{max}$. The IEEE754 Standard [1] specifies a few values for p, e_{min} and e_{max} which are summarized in Figure 1. We consider only binary formats, i.e. we always consider that $\beta = 2$.

Basically, the type of a floating-point number f is $\mathtt{Float}\{\mathtt{s}, \mathtt{u}, \mathtt{p}\}$, where u and p are respectively the $u$nit in the $f$irst $p$lace (ufp) of f and the precision p. The sign s may be either positive, negative, null or unknown, respectively denoted $+$, $-$, 0 and $\top$. The ufp of a number x is

$$\mathsf{ufp}(f) = \min\left\{i \in \mathbb{N} \ : \ 2^{i+1} > f\right\} = \lfloor \log_2(f) \rfloor \ . \tag{2}$$

Similiarly, we also define the $u$nit in the $l$ast $p$lace (ulp) used later in this article. The ulp of a floating-point number of precision p (i.e. which significand has size p) is defined by

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - p + 1 \ . \tag{3}$$

For example, the type of 1.234 is $\mathtt{Float}\{+, 0, 53\}$ since $\mathsf{ufp}(1.234) = 0$ and since we assume that, by default, the floating-point numbers are in double precision (see Figure 1). Other formats may be specified by the programmer, as in the example below.

```
> 1.234 ;;
- : Float{+,0,53} = 1.2340000000000000 +/- 1.11022302463e-16

> 1.234{4} ;;
- : Float{+,0,4} = 1.23 +/- 0.0625
```

Let us remark that in our implementation the type information is used by the pretty printer to display only the correct digits of a number and a bound on the roundoff error. As in ML, our type system admits parameterized types [11].

```
> let f = fun x -> x + 1.0 ;;
val f : Float{'a,'b,'c} -> Float{<expr>,<expr>,<expr>} = <fun>

> verbose true ;;
- : unit = ()

> f ;;
- : Float{'a,'b,'c} -> Float{((max 'b 0) +_ (sigma+ 'd 1)),
                              ((max 'b 0) +_ (sigma+ 'e 1)),
                              ((((max 'b 0) +_ 1) -_ (max ('b -_ 'c) -53))
                               -_ (iota ('b -_ 'c) -53))} = <fun>
```

In the example above, the type of `f` is a function of an argument whose parameterized type is `Float{'a, 'b, 'c}`, where `'a`, `'b` and `'c` are two type variables. The return type of `f` is `Float{`e_0`,`e_1`,`e_2`}` where e_0, e_1 and e_2 are arithmetic expressions containing the variables `'a`, `'b` and `'c`. By default these expressions are not displayed by the system (as higher order values are not explicitely displayed in ML implementations) but we may enforce the system to print them. In our implementation we write `+`, `-`, `*` and `/` the floating-point operators and `+_`, `-_`, `*_` and `/_` their integer versions. The expressions arising in the type of `f` are explained in Section 3. As shown below, various applications of `f` yield results of various types, depending on the type of the argument.

```
> f 1.234 ;;
- : Float{+,1,53} = 2.2340000000000000 +/- 2.22044604925e-16

> f 1.234{4} ;;
- : Float{+,1,5} = 2.23 +/- 0.0625
```

If the type system infers that the result of some computation has no significant digit, then an error is raised. For example, it is well-known that in IEEE754 double precision $(10^{16}+1)-10^{16}=0$. Our type system rejects this computation.

```
> (1.0e15 + 1.0) - 1.0e15 ;;
- : Float{+,50,54} = 1.0 +/- 0.0625

> (1.0e16 + 1.0) - 1.0e16 ;;
Error: The computed value has no significant digit. Its ufp is 0 but the ulp of the certified
value is 1
```

Last but not least, our type system has to accept recursive functions.

3 The Type System

$$
\begin{array}{rl}
e ::= & \mathtt{f}\{u,p\} \in \mathsf{F}_{u,p} \mid \mathtt{i} \in \mathsf{I} \mid \mathtt{b} \in \mathsf{B} \mid \mathtt{id} \in \mathsf{V} \\
 & \mid \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \lambda x.e \mid e_0\ e_1 \mid t \\
\\
t ::= & \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{Float}\{\mathtt{e}_0,\mathtt{e}_1,\ \} \mid \alpha \mid \Pi.x : e_0.e_1
\end{array} \tag{4}
$$

The terms of the language are defined in Equation (4). The e terms correspond to statements. Constants are integers `i`, booleans `b` and floating-point numbers $\mathtt{f}\{u,p\}$ where `f` is the value itself and $u,p \in \mathsf{I}$ the format of `f`. Identifiers are denoted `id`. The language also admits conditionals, functions $\lambda x.e$ and application $e_0\ e_1$. The t terms of the grammar corresponds to types expressions which are mutually recursive with statements. The types of constants are `int`, `bool` and $\mathtt{Float}\{\mathtt{e}_0,\mathtt{e}_1,\mathtt{e}_2\}$ where e_0, e_1 and e_2 are general expressions denoting the format of the floating-point number. Type variables are denoted α and

$$\frac{}{\Gamma \vdash \mathtt{i} : \mathtt{int}}\ (\textsc{Int}) \qquad \frac{}{\Gamma \vdash \mathtt{b} : \mathtt{bool}}\ (\textsc{Bool})$$

$$\frac{\mathsf{ufp}(\mathtt{f}) \leq u}{\Gamma \vdash \mathtt{f\{u,p\}} : \mathtt{Float\{s,u,p\}}}\ (\textsc{Float}) \qquad \frac{\mathtt{id} : t \in \Gamma}{\Gamma \vdash \mathtt{id} : t}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e_0 : \mathtt{bool} \qquad \Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t = t_1 \sqcup t_2}{\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : t}\ (\textsc{Cond})$$

$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : \Pi x : t_1.t_2}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash e_1 : \Pi x : t_0.t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 \sqsubseteq t_1}{\Gamma \vdash e_1\ e_2 : t_2[x \mapsto e_2]}\ (\textsc{App})$$

Fig. 2. Inference rules for our type system.

$\Pi.x : e_0.e_1$ represents ???. Let us note that abstractions $\lambda x.e$ are defined in the language for convenience. They are syntactic sugar for $\Pi x.$?????? The arithmetic and logic operators are viewed as functional constants of the language.

$$\mathtt{Float}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \sqsubseteq \mathtt{Float}\{\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2\} \iff s_1 \sqsubseteq s_2 \wedge u_2 \geq u_1 \wedge p_2 \geq u_1 - u_2 + p_1 \tag{5}$$

In Rule (App), the type T_2 of the argument is less than the return type T_1 of the function and the type of the whole expression is the the return type T_1 of the function.

$$\begin{array}{r} + : \Pi u_1 : \mathtt{int}.\Pi p_1 : \mathtt{int}.\Pi u_2 : \mathtt{int}.\Pi p_2 : \mathtt{int}. \\ \mathtt{Float}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \to \mathtt{Float}\{\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2\} \to \mathtt{Float}\{??,\mathtt{u}_1+\mathtt{u}_2+1,\mathtt{u}_1+\mathtt{u}_2+1-\max\big(\mathtt{u}_1-\mathtt{p}_1,\mathtt{u}_2-\mathtt{p}_2\big) - \iota \end{array}$$

$$\begin{array}{l} + : \Pi \mathtt{u}_1 : \mathtt{int}, \mathtt{p}_1 : \mathtt{int}, \mathtt{u}_2 : \mathtt{int}, \mathtt{p}_2 : \mathtt{int}. \\ \qquad \mathtt{Float}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \to \mathtt{Float}\{\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2\} \to \mathtt{Float}\{\mathcal{S}_+(\mathtt{s}_1,\mathtt{s}_2),\mathcal{U}_+(\mathtt{u}_1,\mathtt{u}_2),\mathcal{P}_+(\mathtt{u}_1,\mathtt{u}_2)\} \end{array}$$

References

1. ANSI/IEEE: IEEE Standard for Binary Floating-point Arithmetic. (2008)
2. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Symposium on Principles of Programming Languages, POPL '14, ACM (2014) 235–248
3. Goubault, E.: Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In: SAS. Volume 7935 of LNCS., Springer (2013) 1–3
4. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: Formal Methods. Volume 9109 of LNCS., Springer (2015) 532–550
5. Denis, C., de Oliveira Castro, P., Petit, E.: Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In: 23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, IEEE Computer Society (2016) 55–62

6. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., LeGendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: Supercomputing, ICS'13, ACM (2013) 369–378
7. Martel, M.: Floating-point format inference in mixed-precision. In: NASA Formal Methods. Volume 10227 of Lecture Notes in Computer Science. (2017) 230–246
8. Rubio-Gonzalez, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: Int. Conf. for High Performance Computing, Networking, Storage and Analysis, ACM (2013) 27:1–27:12
9. Damouche, N., Martel, M., Chapoutot, A.: Intra-procedural optimization of the numerical accuracy of programs. In: Formal Methods for Industrial Critical Systems, FMICS. Volume 9128 of LNCS., Springer (2015) 31–46
10. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI, ACM (2015) 1–11
11. Pierce, B.C.: Types and programming languages. MIT Press (2002)
12. Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2004)
13. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. MIT Press (1997)
14. Atkinson, K.: An Introduction to Numerical Analysis, 2nd Edition. Wiley (1989)