Numl: A Strongly Typed Language for Numerical Accuracy[†]

Matthieu Martel

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

Abstract. It is well-known that numerical computations may sometimes lead to wrong results because of roundoff errors. We propose an ML-like type system (strong, implicit, polymorphic) for numerical computations in finite precision, in which the type of an expression carries information on its accuracy. We use dependent types and a type inference which, from the user point of view, acts like ML type inference. 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. The soundness of the type system is ensured by a subject reduction theorem and we show that our type system is able to type implementations of usual simple numerical algorithms.

1 Introduction

It is well-known that numerical computations may sometimes lead to wrong results because of the accumulation of roundoff errors [10]. Recently, much work has been done to detect these accuracy errors in finite precision computations [1], by static [8, 11, 24] or dynamic [9] analysis, to find the least data formats needed to ensure a certain accuracy (precision tuning) [14, 16, 23] and to optimize the accuracy by program transformation [7, 20]. All these techniques are used late in the software development cycle, 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 adoption reasons.

We propose an `ML`-like type system (strong, implicit, polymorphic [21]) for numerical computations in which the type of an arithmetic expression carries information on

[†]This work is supported by the Office for Naval Research Global under Grant NICOP N62909-18-1-2068 (Tycoon project). https://www.onr.navy.mil/en/Science-Technology/ONR-Global

Martel, M.
Numl - A Strongly Typed Language for Numerical Accuracy.
DOI: 10.5220/0008862701770200
In *OPPORTUNITIES AND CHALLENGES for European Projects (EPS Portugal 2017/2018 2017)*, pages 177-200
ISBN: 978-989-758-361-2

its accuracy. We use dependent types [22] and a type inference which, from the user point of view, acts like `ML` [17] type inference [21] 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 finite precision 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.

In our type system, unification necessitates to solve sets of constraints made of propositional logic formulas and relations between affine expressions over integers (and only integers). Indeed, these relations remain linear even if the term to be typed contains non-linear computations. As a consequence, these constraints can be easily checked by a SMT solver (we use `Z3` in practice [18]).

Let us insist on the fact 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. The soundness of our type system relies on a subject reduction theorem introduced in Section 4. Based on an instrumented operational semantics computing both the finite precision and exact results of a numerical computation, this theorem shows that the error on the result of the evaluation of some expression e is less than the error predicted by the type of e. Obviously, as any non-trivial type system, our type system is not complete and rejects certain programs that would not produce unbounded numerical errors. Our type system has been implemented in a prototype language `Numl` and we show that, in practice, our type system is expressive enough to type implementations of usual simple numerical algorithms [2] such as the ones of Section 6. Let us also mention that our type system represents a new application of dependent type theory motivated by applicative needs. Indeed, dependent types arise naturally in our context since accuracy depends on values.

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 language, `Numl`. The formal definition of the types and of the inference rules are given in Section 3. Section 3.1 introduces the type system itself while Section 3.2 introduces the types of the primitives of the language. A soundness theorem is given in Section 4. The implementation of the type system is discussed in Section 5. Sections 5.1 and 6 present the unification algorithm and Section 6 presents experimental results Section 7 discuss the special case of the IEEE754 floating-point arithmetic [1]. Section 8 describes related work and Section 9 concludes.

2 Programming with Types for Numerical Accuracy

In this section, we present informally how our type system works throughout a programming sequence in our language, `Numl`. First of all, we use real numbers $\mathsf{r}\{s, u, p\}$ where r is the value itself, and $\{s, u, p\}$ the format of r. The format of a real number is made of a sign $s \in \mathsf{Sign}$ and integers $u, p \in \mathsf{Int}$ such that u is the unit in the first place of r, written $\mathsf{ufp}(\mathsf{r})$ and p the precision (i.e. the number of digits of the number). For inputs, p is either explicitely specified by the user or set by default by the system. For

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.

outputs, p is inferred by the type system. We have $\mathsf{Sign} = \{0, \oplus, \ominus, \top\}$ and $\mathsf{sign}(\mathsf{r}) = 0$ if $\mathsf{r} = 0$, $\mathsf{sign}(\mathsf{r}) = \oplus$ if $\mathsf{r} > 0$ and $\mathsf{sign}(\mathsf{r}) = \ominus$ if $\mathsf{r} < 0$. The set Sign is equipped with the partial order relation $\prec \subseteq \mathsf{Sign} \times \mathsf{Sign}$ defined by $0 \prec \oplus$, $0 \prec \ominus$, $\oplus \prec \top$ and $\ominus \prec \top$. The ufp of a number x is

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

The term p defines the precision of r. Let $\varepsilon(\mathsf{r})$ be the absolute error on r, we assume that $\varepsilon(\mathsf{r}) < 2^{u-p+1}$. The errors on the numerical constants arising in programs are specified by the user or determined by default by the system. The errors on the computed values can be inferred by propagation of the initial errors. Similarly to Equation (1), we also define the *u*nit in the *l*ast *p*lace (ulp) used later in this article. The ulp of a number of precision p is defined by

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - p + 1 \ . \tag{2}$$

For example, the type of 1.234 is $\mathtt{real}\{+, 0, 53\}$ since $\mathsf{ufp}(1.234) = 0$ and since we assume that, by default, the real numbers have the same precision as in the IEEE754 double precision floating-point format [1] (see Figure 1). Other formats may be specified by the programmer, as in the example below. Let us also mention that our type system is independent of a given computer arithmetic. The interpreter only needs to implement the formats given by the type system, using floating-point numbers, fixed-point numbers [12], multiple precision numbers[1], etc in order to ensure that the finite precision operations are computed exactly. The special case of IEEE754 floating-point arithmetic, which introduces additional errors due to the roundoff on results of operations can also be treated by modifying slightly the equations of Section 3.

```
> 1.234 ;; (* precision of 53 bits by default *)
- : real{+,0,53} = 1.234000000000000

> 1.234{4};; (* precision of 4 bits specified by the user *)
- : real{+,0,4} = 1.2
```

Notice that, in `Numl`, 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.

Note that accuracy is not a property of a number but a number that states how closely a particular finite-precision number matches some ideal true value. For example, using the basis $\beta = 10$ for the sake of simplicity, the floating-point value 3.149 represents π

[1] `https://gmplib.org/`

with an accuracy of 3. It itself has a precision of 4. It represents the real number 3.14903 with an accuracy of 4. As in ML, our type system admits parameterized types [21].

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

> verbose true ;;
- : unit = ()

> f ;;
- : real{'a,'b,'c} -> real{(SignPlus 'a 'b 1 0),((max 'b 0) +_ (sigma+ 'a 1)),
((((max 'b 0) +_ (sigma+ 'a 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 `real`$\{$`'a,'b,'c`$\}$, where `'a`, `'b` and `'c` are three type variables. The return type of the function `f` is `Real`$\{e_0,e_1,e_2\}$ 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 (just like higher order values are not explicitly displayed in ML implementations) but we may enforce the system to print them. In Numl, we write $+$, $-$, $\star$ and $/$ for the operators over real numbers. Integer expressions have type `int` and we write $+_{\_}$, $-_{\_}$, $\star_{\_}$ and $/_{\_}$ for the elementary operators over integers. 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 ;;
- : real{+,1,53} = 2.234000000000000

> f 1.234{4} ;;
- : real{+,1,5} = 2.2
```

If the interpreter detects 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 ;;
- : real{+,50,54} = 1.0

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

Last but not least, our type system accepts recursive functions. For example, we have:

```
> let rec g x = if x < 1.0 then x else g (x * 0.07) ;;
val g : real{+,0,53} -> real{+,0,53} = <fun>

> g 1.0 ;;
- : real{+,0,53} = 0.07000000000000

> g 2.0 ;;
Error: This expression has type real{+,1,53} but an expression was
expected of type real{+,0,53}
```

In the above session, the type system unifies the return type of the function with the type of the conditional. The types of the `then` and `else` branches also need to be unified. Then the return type is `real`$\{$`+,0,53`$\}$ which corresponds to the type of the

value `1.0` used in the `then` branch. The type system also unifies the return type with the type of the argument since the function is recursive. Finally, we obtain that the type of `g` is `real{+,0,53} -> real{+,0,53}`. As a consequence, we cannot call `g` with an argument whose ufp is greater than $\mathsf{ufp}(1.0) = 0$. To overcome this limitation, we introduce new comparison operations for real numbers. While the standard comparison operator < has type `'a -> 'a -> bool`, the operator `<{s,u,p}` has type `real{s,u,p} -> real{s,u,p} -> bool`. In other words, the compared value are cast in the format $\{s, u, p\}$ before performing the comparison. Now we can write the code:

```
> let rec g x = if x <{*,10,15} 1.0 then x else g (x * 0.07) ;;
val g : real{*,10,15} -> real{*,10,15} = <fun>

> g 2.0 ;;
- : real{*,10,15} = 0.1

> g 456.7 ;;
- : real{*,10,15} = 0.1

> g 4567.8 ;;
Error: This expression has type real{+,12,53} but an expression was
expected of type real{*,10,15}
```

Interestingly, unstable functions (for which the initial errors grow with the number of iterations) are not typable. This is a desirable property of our system.

```
> let rec h n = if (n=0) then 1.0 else 3.33 * (h (n -_ 1)) ;;
Error: This expression has type real{+,-1,-1} but an expression was
expected of type real{+,-3,-1}
```

Stable computations should be always accepted by our type system. Obviously, this is not the case and, as any non-trivial type system, our type system rejects some correct programs. The challenge is then to accept enough programs to be useful from an end-user point of view. We end this section by showing another example representative of what our type system accepts. More examples are given later in this article, in Section 6. The example below deals with the implementation of the Taylor series $\frac{1}{1-x} = \sum_{n\geq 0} x^n$. The implementation gives rise to a simple recursion, as shown in the programming session below.

```
> let rec taylor x{*,-1,25} xn i n = if (i > n) then 0.0{*,10,20}
                                    else xn + (taylor x (x * xn) (i +_ 1) n) ;;

val taylor : real{*,-1,25} -> real{*,10,20} -> int -> int -> real{*,10,20} = <fun>

> taylor 0.2 1.0 0 5;;
- : real{*,10,20} = 1.249
```

Obviously, our type system computes the propagation of the errors due to finite precision but does not take care of the method error intrinsic to the implemented algorithm (the Taylor series instead of the exact formula $\frac{1}{1-x}$ in our case.) All the programming sessions introduced above as well as the additional examples of Section 6 are fully interactive in our system, `Numl`, i.e. the type judgments are obtained instantaneously (about 0.01 second in average following our measurements) including the most complicated ones.

3 The Type System

In this section, we introduce the formal definition of our type system for numerical accuracy. First, in Section 3.1, we define the syntax of expressions and types and we introduce a set of inference rules. Then we define in Section 3.2 the types of the primitives for the operators among real numbers (addition, product, etc.) These types are crucial in our system since they encode the propagation of the numerical accuracy information.

3.1 Expressions, Types and Inference Rules

In this section, we introduce the expressions, types and typing rules for our language. For the sake of simplicity, the syntax introduced hereafter uses notations *à la* lambda calculus instead of the `ML`-like syntax employed in Section 2. In our system, expressions and types are mutually dependent. They are defined inductively using the grammar of Equation (3).

$$\begin{aligned} \mathsf{Expr} \ni e ::= {} & \mathtt{r}\{s,u,p\} \in \mathsf{Real}_{u,p} \mid \mathtt{i} \in \mathsf{Int} \mid \mathtt{b} \in \mathsf{Bool} \mid \mathtt{id} \in \mathsf{Id} \\ & \mid \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \lambda x.e \mid e_0\ e_1 \mid \mathtt{rec}\ f\ x.e \mid t \\ \mathsf{Typ} \ni t ::= {} & \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{real}\{i_0,i_1,i_2\} \mid \alpha \mid \Pi x : e_0.e_1 \\ \mathsf{IExp} \ni i ::= {} & \mid \mathtt{int} \mid \mathtt{op} \in \mathsf{Id_I} \mid \alpha \mid i_0\ i_1 \end{aligned} \tag{3}$$

In Equation (3), the e terms correspond to expressions. Constants are integers $\mathtt{i} \in \mathsf{Int}$, booleans $\mathtt{b} \in \mathsf{Bool}$ and real numbers $\mathtt{r}\{s,u,p\}$ where r is the value itself, $s \in \mathsf{Sign}$ is the sign as defined in Section 2 and $u, p \in \mathsf{Int}$ the ufp (see Equation (1)) and precision of $\mathtt{r}$. For inputs, the precision p is given by the user by means of annotations or chosen by default by the system. Then p is inferred for the outputs of programs. The term p defines the precision of r. Let $\varepsilon(\mathsf{r})$ be the absolute error on r, we assume that

$$\varepsilon(\mathsf{r}) < 2^{u-p+1} \ . \tag{4}$$

The errors on the numerical constants arising in programs are specified by the user or determined by default by the system. The errors on the computed values can be inferred by propagation of the initial errors.

In Equation (3), identifiers belong to the set Id and we assume a set of pre-defined identifiers $+, -, \times, \leq, =, \ldots$ related to primitives for the logical and arithmetic operations. We write $+$, $-$, $\times$ and $\div$ the operations on real numbers and $+_{\_}$, $-_{\_}$, $\times_{\_}$ and $\div_{\_}$ the operations among integers. The language also admits conditionals, functions $\lambda x.e$, applications $e_0\ e_1$ and recursive functions $\mathtt{rec}\ f\ x.e$ where f is the name of the function, x the parameter and e the body. The language of expressions also includes type expressions t defined by the second production of the grammar of Equation (3).

The definition of expressions and type is mutually recursive. Type variables are denoted α, β, ... and $\Pi x : e_0.e_1$ is used to introduce dependent types [22]. Let us notice that our language does not explicitly contain function types $t_0 \to t_1$ since they are encoded by means of dependent types. Let $\equiv$ denote the syntactic equivalence, we have

$$t_0 \to t_1 \equiv \Pi x : t_0.t_1 \quad \text{with } x \text{ not free in } t_1 \ . \tag{5}$$

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

$$\frac{\mathsf{sign}(\mathtt{r}) \prec s \quad \mathsf{ufp}(\mathtt{r}) \leq u}{\Gamma \vdash \mathtt{r\{s,u,p\}} : \mathtt{real\{s,u,p\}}} \ (\textsc{Real}) \qquad \frac{\Gamma(\mathtt{id}) = t}{\Gamma \vdash \mathtt{id} : t} \ (\textsc{Id})$$

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

$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : \Pi x : t_1.t_2} \ (\textsc{Abs}) \qquad \frac{\Gamma, x : t_1, f : \Pi.y : t_1.t_2 \vdash e : t_2 \quad y \text{ not free in } t_2}{\Gamma \vdash \mathtt{rec}\ f\ x.e : \Pi x : t_1.t_2} \ (\textsc{Rec})$$

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

Fig. 2. Typing rules for our language.

For convenience, we also write $\lambda x_0.x_1 \ldots x_n.e$ instead of $\lambda x_0.\lambda x_1 \ldots \lambda x_n.e$ and $\Pi x_0 : t_0.x_1 : t_1 \ldots x_n : t_n.e$ instead of $\Pi x_0 : t_0.\Pi x_1 : t_1 \ldots \Pi x_n : t_n.e$.

The types of constants are `int`, `bool` and $\mathtt{real}\{i_0, i_1, i_2\}$ where i_0, i_1 and i_2 are integer expressions denoting the format of the real number. Integer expressions of $\mathsf{IExpr} \subseteq \mathsf{Expr}$ are a subset of expressions made of integer numbers, integer primitives of $\mathsf{Id_I} \subseteq \mathsf{Id}$ (such as $+_{\_}$, $\times_{\_}$, etc.), type variables and applications. Note that this definition restricts significantly the set of expressions which may be written inside `real` types.

The typing rules for our system are given in Figure 2. These rules are mostly classical. The type judgment $\Gamma \vdash e : t$ means that in the type environment Γ, the expression e has type t. A type environment $\Gamma : \mathsf{Id} \to \mathsf{Typ}$ maps identifiers to types. We write $\Gamma\ x : t$ the environment Γ in which the variable x has type t. The typing rules (Int) and (Bool) are trivial. Rule (Real) states that the type of a real number $\mathtt{r\{s,u,p\}}$ is $\mathtt{real\{s,u,p\}}$ assuming that the actual sign of $\mathtt{r}$ is less than s and that the ufp of $\mathtt{r}$ is less than u. Following Rule (Id), an identifier $\mathtt{id}$ has type t if $\Gamma(\mathtt{id}) = t$. Rules (Cond), (Abs) and (Rec) are standard rules for conditionals and abstractions respectively. The rule for application, (App), requires that the first expression e_1 has type $\Pi x : t_0.t_1$ (which is equivalent to $t_0 \to t_1$ if x is not free in t_1) and that the argument e_2 has some type $t_2 \sqsubseteq t_0$. The sub-typing relation $\sqsubset$ is introduced for real numbers. Intuitively, we want to allow the argument of some function to have a smaller ulp than what we would require if we used $t_0 = t_2$ in Rule (App), provided that the precision p remains as good with t_2 as with t_0. This relaxation allows to type more terms without invalidating the type judgments. Formally, the relation $\sqsubseteq$ is defined by

$$\mathtt{real\{s_1,u_1,p_1\}} \sqsubseteq \mathtt{real\{s_2,u_2,p_2\}} \iff s_1 \sqsubseteq s_2 \wedge u_2 \geq u_1 \wedge p_2 \leq u_2 - u_1 + p_1 \ . \tag{6}$$

In other words, the sub-typing relation of Equation (6) states that it is always correct to add zeros before the first significant digit of a number, as illustrated in Figure 3.

3.2 Types of Primitives

In this section, we introduce the types of the primitives of our language. As mentionned earlier, the arithmetic and logic operators are viewed as functional constants of the language. The type of a primitive for an arithmetic operation among integers $*_{\_} \in$

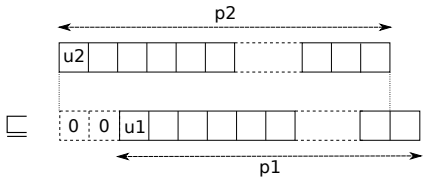

Fig. 3. The sub-typing relation $\sqsubseteq$ of Equation (6).

$\{+_{\_}, -_{\_}, \times_{\_}, \div_{\_}\}$ is

$$t_{*\_} = \Pi x : \mathtt{int}.y : \mathtt{int}.\mathtt{int} \ . \tag{7}$$

The type of comparison operators $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$ are polymorphic with the restriction that they reject the type $\mathtt{real}\{\mathtt{s}, \mathtt{u}, \mathtt{p}\}$ which necessitates special comparison operators:

$$t_{\bowtie} = \Pi x : \alpha.y : \alpha.\mathtt{bool} \quad \alpha \neq \mathtt{real}\{\mathtt{s}, \mathtt{u}, \mathtt{p}\} \ . \tag{8}$$

For real numbers, we use comparisons at a given accuracy defined by the operators $\bowtie_{\{u,p\}} \in \{<_{\{u,p\}}, >_{\{u,p\}}\}$. We have

$$t_{\bowtie_{\{u,p\}}} = \Pi \mathtt{s} : \mathtt{int}, \mathtt{u} : \mathtt{int}, \mathtt{p} : \mathtt{int}.\mathtt{real}\{\mathtt{s}, \mathtt{u}, \mathtt{p}+1\} \to \mathtt{real}\{\mathtt{s}, \mathtt{u}, \mathtt{p}+1\} \to \mathtt{bool} \ .$$

Notice that the operands of a comparison $\bowtie_{\{u,p\}}$ must have $p+1$ bits of accuracy. This is to avoid unstable tests, as detailed in the proof of Lemma 3 in Section 4. An unstable test is a comparison between two approximate values such that the result of the comparison is altered by the approximation error. For instance, if we reuse an example of Section 2, in IEEE754 double precision, the condition $10^{16} + 1 = 10^{16}$ evaluates to `true`. We need to avoid such situations in our language in order to preserve our subject reduction theorem (we need the control-flow be the same in the finite precision and exact semantics). Let us also note that our language does not provide an equality relation $=_{\{u,p\}}$ for `real` values. Again this is to avoid unstable tests. Given values x and y of type $\mathtt{real}\{\mathtt{s}, \mathtt{u}, \mathtt{p}\}$, the programmer is invited to use $|x - y| < 2^{u-p+1}$ instead of $x = y$ in order to get rid of the perturbations of the finite precision arithmetic.

The types of primitives for real arithmetic operators are fundamental in our system since they encode the propagation of the numerical accuracy information. They are defined in figures 4 and 5. The type t_* of some operation $* \in \{+, -, \times, \div\}$ is a pi-type with takes six arguments $\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2$ and $\mathtt{p}_2$ of type `int` corresponding to the sign, ufp and precision of the two operands of $*$ and which produces a type

$$\mathtt{real}\{\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1\} \to \mathtt{real}\{\mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2\} \to \mathtt{real}\{\mathcal{S}_*(\mathtt{s}_1, \mathtt{s}_2), \mathcal{U}_*(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2), \mathcal{P}_*(\mathtt{u}_1, \mathtt{p}_1, \mathtt{u}_2, \mathtt{p}_2)\} \tag{9}$$

where $\mathcal{S}_*$, $\mathcal{U}_*$ and $\mathcal{P}_*$ are functions which compute the sign, ufp and precision of the result of the operation $*$ in function of $\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2$ and $\mathtt{p}_2$. These functions extend the functions used in [16].

The functions $\mathcal{S}_*$ determine the sign of the result of an operation in function of the signs of the operands and, for additions and subtractions, in function of the ufp of the operands. The functions $\mathcal{U}_*$ compute the ufp of the result. Notice that $\mathcal{U}_+$ and $\mathcal{U}_-$ use

$$
\begin{aligned}
\mathtt{t}_* = \Pi \mathtt{s}_1 : \mathtt{int}, \mathtt{u}_1 : \mathtt{int}, \mathtt{p}_1 : \mathtt{int}, \mathtt{s}_2 : \mathtt{int}, \mathtt{u}_2 : \mathtt{int}, \mathtt{p}_2 : \mathtt{int}.\\
\mathtt{real}\{\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1\} \rightarrow \mathtt{real}\{\mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2\}\\
\rightarrow \mathtt{real}\{\mathcal{S}_*(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2), \mathcal{U}_*(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2), \mathcal{P}_*(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}_+(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2)) &= \max(\mathtt{u}_1, \mathtt{u}_2) + \sigma_+(\mathtt{s}_1, \mathtt{s}_2)\\
\mathcal{P}_+(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) &= \max(\mathtt{u}_1, \mathtt{u}_2) + \sigma_+(\mathtt{s}_1, \mathtt{s}_2) -\\
&\quad \max(\mathtt{u}_1 - \mathtt{p}_1, \mathtt{u}_2 - \mathtt{p}_2) - \iota(\mathtt{u}_1 - \mathtt{p}_1, \mathtt{u}_2 - \mathtt{p}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}_-(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2)) &= \max(\mathtt{u}_1, \mathtt{u}_2) + \sigma_-(\mathtt{s}_1, \mathtt{s}_2)\\
\mathcal{P}_-(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) &= \max(\mathtt{u}_1, \mathtt{u}_2) + \sigma_-(\mathtt{s}_1, \mathtt{s}_2) -\\
&\quad \max(\mathtt{u}_1 - \mathtt{p}_1, \mathtt{u}_2 - \mathtt{p}_2) - \iota(\mathtt{u}_1 - \mathtt{p}_1, \mathtt{u}_2 - \mathtt{p}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}_\times(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2)) &= \mathtt{u}_1 + \mathtt{u}_2 + 1\\
\mathcal{P}_\times(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) &= \mathtt{u}_1 + \mathtt{u}_2 + 1 -\\
&\quad \max(\mathtt{u}_1 + \mathtt{u}_2 + 1 - \mathtt{p}_1, \mathtt{u}_1 + \mathtt{u}_2 + 1 - \mathtt{p}_2) - \iota(\mathtt{p}_1, \mathtt{p}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}_\div(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2)) &= \mathtt{u}_1 - \mathtt{u}_2 + 1\\
\mathcal{P}_\div(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) &= \mathcal{P}_\times(\mathtt{u}_1, \mathtt{p}_1, \mathtt{u}_2, \mathtt{p}_2) - 1 \qquad \iota(x, y) = \begin{cases} 1 \text{ if } x = y, \\ 0 \text{ otherwise.} \end{cases}
\end{aligned}
$$

Fig. 4. Types of the primitives corresponding to the elementary arithmetic operations $* \in \{+, -, \times, \div\}$. The functions $\mathcal{S}_*$ and σ_* are defined in Figure 5.

the functions σ_+ and σ_-, respectively. These functions are defined in the bottom right corner of Figure 5 to increment the ufp of the result of some addition or subtraction in the relevant cases only. For example if a and b are two positive real numbers then $\mathsf{ufp}(a+b)$ is possibly $\max\big(\mathsf{ufp}(a), \mathsf{ufp}(b)\big) + 1$ but if $a > 0$ and $b < 0$ then $\mathsf{ufp}(a+b)$ is not greater than $\max\big(\mathsf{ufp}(a), \mathsf{ufp}(b)\big)$. The functions $\mathcal{P}_*$ compute the precision of the result. Basically, they compute the number of bits between the ufp and the ulp of the result.

We end this section by exhibiting some properties of the functions $\mathcal{P}_*$. Let $\varepsilon(x)$ denote the error on $x \in \mathsf{Real}_{u,p}$. We have $\varepsilon(x) < 2^{u-p+1} = \mathsf{ulp}(x)$. Let us start with addition. Lemma 1 relates the accuracy of the operands to the accuracy of the result of an addition between two values x and y. Lemma 2 is similar to Lemma 1 for product.

Lemma 1. *Let x and y be two values such that $\varepsilon(x) < 2^{u_1-p_1+1}$ and $\varepsilon(y) < 2^{u_2-p_2+1}$. Let $z = x + y$,*

$$
\begin{aligned}
u &= \mathcal{U}_+(s_1, u_1, s_2, u_2)\\
p &= \mathcal{P}_+(s_1, u_1, p_1, s_2, u_2, p_2) \ .
\end{aligned} \tag{10}
$$

Then $\varepsilon(z) < 2^{u-p+1}$.

Proof. The errors on addition may be bounded by $e_+ = \varepsilon(x) + \varepsilon(y)$. Then the most significant bit of the error has weight $\mathsf{ufp}(e_+)$ and the accuracy of the result is $p = \mathsf{ufp}(x+y) - \mathsf{ufp}(e_+)$. Let $u = \mathsf{ufp}(x+y) = \max(u_1, u_2) + \sigma_+(s_1, s_2) = \mathcal{U}_+(s_1, u_1, s_2, u_2)$. We need to over-approximate e_+ in order to ensure p. We have $\varepsilon(x) < 2^{u_1-p_1+1}$ and $\varepsilon(y) < 2^{u_2-p_2+1}$ and, consequently, $e_+ < 2^{u_1-p_1+1} + 2^{u_2-p_2+1}$. We introduce the function $\iota(x, y)$ also defined in Figure 4 and which is equal to 1 if $x = y$ and 0 otherwise. We have

$$
\begin{aligned}
\mathsf{ufp}(e_+) &< \max(u_1 - p_1 + 1, u_2 - p_2 + 1) + \iota(u_1 - p_1, u_2 - p_2)\\
&\leq \max(u_1 - p_1, u_2 - p_2) + \iota(u_1 - p_1, u_2 - p_2)
\end{aligned}
$$

$\mathcal{S}_+$

$s_1 \backslash s_2$	0	+	−	⊤
0	0	+	−	⊤
+	+	+	+ if $u_1 < u_2$ − if $u_2 < u_1$ ⊤ otherwise	⊤
−	−	+ if $u_2 < u_1$ − if $u_1 < u_2$ ⊤ otherwise	−	⊤
⊤	⊤	⊤	⊤	⊤

$\mathcal{S}_\times$ and $\mathcal{S}_\div$

$s_1 \backslash s_2$	0	+	−	⊤
0	0	0	0	0
+	0	+	−	⊤
−	0	−	+	⊤
⊤	0	⊤	⊤	⊤

$\mathcal{S}_-$

$s_1 \backslash s_2$	0	+	−	⊤
0	0	−	+	⊤
+	+	− if $u_1 < u_2$ + if $u_2 < u_1$ ⊤ otherwise	+	⊤
−	−	−	− if $u_2 < u_1$ + if $u_1 < u_2$ ⊤ otherwise	⊤
⊤	⊤	⊤	⊤	⊤

σ_+

	0	+	−	⊤
0	0	0	0	0
+	0	1	0	1
−	0	0	1	1
⊤	0	1	1	1

σ_-

	0	+	−	⊤
0	0	0	0	0
+	0	0	1	1
−	0	1	0	1
⊤	0	1	1	1

Fig. 5. Operators used in the types of the primitives of Figure 4.

Let us write $p = \max(u_1 - p_1, u_2 - p_2) - \iota(u_1 - p_1, u_2 - p_2) = \mathcal{P}_+(s_1, u_1, p_1 s_2, u_2, p_2)$. We conclude that $u = \mathcal{U}_+(s_1, u_1, s_2, u_2)$, $p = \mathcal{P}_+(s_1, u_1, p_1 s_2, u_2, p_2)$ and $\varepsilon(z) < 2^{u-p+1}$. □

Lemma 2. *Let x and y be two values such that $\varepsilon(x) < 2^{u_1 - p_1 + 1}$ and $\varepsilon(y) < 2^{u_2 - p_2 + 1}$. Let $z = x \times y$, $u = \mathcal{U}_\times(s_1, u_1, s_2, u_2)$ and $p = \mathcal{P}_\times(s_1, u_1, p_1, s_2, u_2, p_2)$. Then $\varepsilon(z) < 2^{u-p+1}$.*

Proof. For product, we have $p = \mathsf{ufp}(x \times y) - \mathsf{ufp}(e_\times)$ with $e_\times = x \cdot \varepsilon(y) + y \cdot \varepsilon(x) + \varepsilon(x) \cdot \varepsilon(y)$. Let $u = u_1 + u_2 + 1 = \mathcal{U}_\times(s_1, u_1, s_2, u_2)$. We have, by definition of ufp, $2^{u_1} \leq x < 2^{u_1+1}$ and $2^{u_2} \leq y < 2^{u_2+1}$. Then $e_\times$ may be bounded by

$$\begin{aligned} e_\times &< 2^{u_1+1} \cdot 2^{u_2-p_2+1} + 2^{p_2+1} \cdot 2^{u_1-p_1+1} + 2^{u_1-p_1+1} \cdot 2^{u_2-p_2+1} \\ &= 2^{u_1+u_2-p_2+2} + 2^{u_1+u_2-p_1+2} + 2^{u_1+u_2-p_1-p_2+2} \ . \end{aligned} \tag{11}$$

Since $u_1+u_2-p_1-p_2+2 < u_1+u_2-p_1+2$ and $u_1+u_2-p_1-p_2+2 < u_1+u_2-p_2+2$, we may get rid of the last term of Equation (11) and we obtain that

$$\begin{aligned} \mathsf{ufp}(e_\times) &< \max(u_1 + u_2 - p_1 + 2, u_1 + u_2 - p_2 + 2) + \iota(p_1, p_2) \\ &\leq \max(u_1 + u_2 - p_1 + 1, u_1 + u_2 - p_2 + 1) + \iota(p_1, p_2) \ . \end{aligned}$$

Let us write $p = \max(u_1+u_2-p_1+1, u_1+u_2-p_2+1) - \iota(p_1, p_2) = \mathcal{P}_\times(s_1, u_1, p_1 s_2, u_2, p_2)$. Then $u = \mathcal{U}_\times(s_1, u_1, s_2, u_2)$, $p = \mathcal{P}_\times(s_1, u_1, p_1 s_2, u_2, p_2)$ and $\varepsilon(z) < 2^{u-p+1}$. □

Note that, by reasoning on the exponents of the values, the constraints resulting from a product become linear. The equations for subtraction and division are almost identical to the equations for addition and product, respectively. Note that the result of a division has one less bit than the result of a product. This is due to the fact that, even if the operands are finite numbers, the result of the division may be irrational and possibly needs to be truncated. We conclude this section with the following theorem which summarize the properties of the types of the result of the four elementary operations.

Theorem 1. *Let x and y be two values such that $\varepsilon(x) < 2^{u_1-p_1+1}$ and $\varepsilon(y) < 2^{u_2-p_2+1}$ and let $* \in \{+,-,\times,\div\}$ be an elementary operation. Let $z = x * y$, let*

$$\begin{array}{l} u = \mathcal{U}_*(s_1,u_1,s_2,u_2) \ , \\ p = \mathcal{P}_*(s_1,u_1,p_1,s_2,u_2,p_2) \ . \end{array} \tag{12}$$

Then $\varepsilon(z) < 2^{u+p-1}$.

Proof. The cases of addition and product correspond to Lemma 1 and Lemma 2, respectively. The cases of subtraction and division are similar. □

`Numl` uses a modified Hindley-Milner type inference algorithm. Linear constraints among integers are generated (even for non linear expressions). They are solvd using a SMT solver. For space limitation reasons, the details of this algorithm are out of the scope of this articl e.

4 Soundness of the Type System

In this section, we introduce a subject reduction theorem proving the consistency of our type system. We use two operational semantics $\rightarrow_{\mathbb{F}}$ and $\rightarrow_{\mathbb{R}}$ for the finite precision and exact arithmetics, respectively. The exact semantics is used for proofs. Obviously, in practice, only the finite precision semantics is implemented. We write $\rightarrow$ whenever a reduction rule holds for both $\rightarrow_{\mathbb{F}}$ and $\rightarrow_{\mathbb{R}}$ (in this case, we assume that the same semantics $\rightarrow_{\mathbb{F}}$ or $\rightarrow_{\mathbb{R}}$ is used in the lower and upper parts of the same sequent). Both semantics are displayed in Figure 6. They concern the subset of the language of Equation (3) which do not deal with types.

$$\begin{array}{l} \mathsf{EvalExpr} \ni e ::= \ \mathtt{r}\{s,u,p\} \in \mathsf{Real}_{u,p} \mid \mathtt{i} \in \mathsf{Int} \mid \mathtt{b} \in \mathsf{Bool} \mid \mathtt{id} \in \mathsf{Id} \\ \qquad\qquad\qquad \mid \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \lambda x.e \mid e_0\ e_1 \mid \mathtt{rec}\ f\ x.e \mid e_0\ *\ e_1 \end{array} \ . \tag{13}$$

In Equation (13), $*$ denotes an arithmetic operator $* \in \{+,-,\times,\div,+_{\_},-_{\_},\times_{\_},\div_{\_}\}$. In Figure 6, Rule (FVAL) of $\rightarrow_{\mathbb{F}}$ transforms a syntactic element describing a real number $\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ in a certain format into a value $v_{\mathbb{F}}$. The finite precision value $v_{\mathbb{F}}$ is an approximation of $\mathtt{r}$ with an error less than the ulp of $\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$. In the semantics $\rightarrow_{\mathbb{R}}$, the real number $\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ simply produces the value r without any approximation by Rule (RVal). Rules (Op1) and (Op2) evaluate the operands of some binary operation and Rule (Op) performs an operation $* \in \{+,-,\times,\div,+_{\_},-_{\_},\times_{\_},\div_{\_}\}$ between two values v_0 and v_1.

$$\frac{|\mathtt{r} - v_{\mathbb{F}}| < 2^{u-p+1} \quad \mathsf{ufp}(\mathtt{r}) \leq u \quad \mathsf{sign}(v_{\mathbb{F}}) \prec s}{\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \rightarrow_{\mathbb{F}} v_{\mathbb{F}}} \text{ (FVal)} \qquad \frac{v_{\mathbb{R}} = \mathtt{r}}{\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \rightarrow_{\mathbb{R}} v_{\mathbb{R}}} \text{ (RVal)}$$

$$\frac{e_0 \rightarrow e_0'}{e_0 * e_1 \rightarrow e_0' * e_1} \text{ (Op1)} \qquad \frac{e_1 \rightarrow e_1'}{v * e_1 \rightarrow v * e_1'} \text{ (Op2)} \qquad * \in \{+, -, \times, \div, +_{\_}, -_{\_}, \times_{\_}, \div_{\_}\}$$

$$\frac{v = v_0 * v_1}{v_0 * v_1 \rightarrow v} \text{ (Op)} \quad * \in \{+, -, \times, \div, +_{\_}, -_{\_}, \times_{\_}, \div_{\_}\} \qquad \mathtt{rec}\ f\ x.e \rightarrow \lambda x.e\langle \mathtt{rec}\ f\ x.e/f \rangle \text{ (Rec)}$$

$$\frac{e_0 \rightarrow e_0'}{e_0 \bowtie e_1 \rightarrow e_0' \bowtie e_1} \text{ (Cmp1)} \qquad \frac{e_1 \rightarrow e_1'}{v \bowtie e_1 \rightarrow v \bowtie e_1'} \text{ (Cmp2)} \qquad \bowtie \in \{<_{\{u,p\}}, >_{\{u,p\}}, <, >\}$$

$$\frac{b = (2^{u-p+1} \bowtie v_1^{\mathbb{F}} - v_0^{\mathbb{F}})}{v_0 \bowtie_{\{u,p\}} v_1 \rightarrow_{\mathbb{F}} b} \text{ (FCmp)} \qquad \frac{b = (v_0 \bowtie v_1)}{v_0 \bowtie_{\{u,p\}} v_1 \rightarrow_{\mathbb{R}} b} \text{ (RCmp)} \qquad \bowtie \in \{<_{\{u,p\}}, >_{\{u,p\}}\}$$

$$\frac{e_1 \rightarrow e_1'}{e_0\ e_1 \rightarrow e_0\ e_1'} \text{ (App1)} \qquad \frac{e_0 \rightarrow e_0'}{e_0\ v \rightarrow e_0'\ v} \text{ (App2)} \qquad (\lambda x.e)\ v \rightarrow e\langle v/x \rangle \text{ (Red)}$$

$$\frac{b = v_0 \bowtie v_1}{v_0 \bowtie v_1 \rightarrow b} \quad \bowtie \in \{=, \neq, <, >, \leq, \geq\} \qquad \frac{e_0 \rightarrow e_0'}{\mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightarrow \mathtt{if}\ e_0'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2} \text{ (Cond)}$$

$$\frac{v = \mathtt{true}}{\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightarrow e_1} \text{ (CondTrue)} \qquad \frac{v = \mathtt{false}}{\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightarrow e_2} \text{ (CondFalse)}$$

Fig. 6. Operational semantics for our language.

Rules (Cmp1), (Cmp2) and (ACmp) deal with comparisons. They are similar to Rules (Op1), (Op2) and (Op) described earlier. Note that the operators $<$, $>$, $=$, $\neq$ concerned by Rule (ACmp) are polymorphic except that they do not accept arguments of type `real`. Rules (FCmp) and (RCmp) are for the comparison of `real` values. Rule (FCmp) is designed to avoid unstable tests by requiring that the distance between the two compared values is greater than the ulp of the format in which the comparison is done. With this requirement, a condition cannot be invalidated by the roundoff errors. Let us also note that, with this definition, $x <_{u,p} y \not\Rightarrow y >_{u,p} x$ or $x >_{u,p} y \not\Rightarrow y <_{u,p} x$. For the semantics $\rightarrow_{\mathbb{R}}$, Rule (RCmp) simply compares the exact values.

The other rules are standard and are identical in $\rightarrow_{\mathbb{F}}$ and $\rightarrow_{\mathbb{R}}$. Rules (App1), (App2) and (Red) are for applications and Rule (Rec) is for recursive functions. We write $e\langle v/x \rangle$ the term e in which v has been substituted to the free occurrences of x. Rules (Cond), (CondTrue) and (CondFalse) are for conditionals.

The rest of this section is dedicated to our subject reduction theorem. First of all, we need to relate the traces of $\rightarrow_{\mathbb{F}}$ and $\rightarrow_{\mathbb{R}}$. We introduce new judgments

$$\Gamma \models (e_{\mathbb{F}}, e_{\mathbb{R}}) \ : \ t \ . \tag{14}$$

Intuitively, Equation (14) means that expression $e_{\mathbb{F}}$ simulates $e_{\mathbb{R}}$ up to accuracy t. In this case, $e_{\mathbb{F}}$ is syntactically equivalent to $e_{\mathbb{R}}$ up to the values which, in $e_{\mathbb{F}}$, are approximations of the values of $e_{\mathbb{R}}$. The value of the approximation is given by type t.

$$\frac{}{\Gamma \models (\mathtt{i},\mathtt{i}) : \mathtt{int}} \ (\textsc{Int}) \qquad \frac{}{\Gamma \models (\mathtt{b},\mathtt{b}) : \mathtt{bool}} \ (\textsc{Bool}) \qquad \frac{\Gamma(\mathtt{id}) = t}{\Gamma \models (\mathtt{id},\mathtt{id}) \ : \ t} \ (\textsc{Id})$$

$$\frac{\mathsf{sign}(\mathtt{r}) \prec s \quad \mathsf{ufp}(\mathtt{r}) \leq u}{\Gamma \models (\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\},\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}) \ : \ \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}} \ (\textsc{SReal}) \qquad \frac{|v_{\mathbb{R}} - v_{\mathbb{F}}| < 2^{u-p+1}}{\Gamma \models (v_{\mathbb{F}}, v_{\mathbb{R}}) \ : \ \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}} \ (\textsc{VReal})$$

$$\frac{\Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \qquad * \in \{+,-,\times,\div\}}{\Gamma \models (e_{1\mathbb{F}} * e_{2\mathbb{F}}, e_{1\mathbb{R}} * e_{2\mathbb{R}}) : \mathtt{real}\{\mathcal{S}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{U}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{P}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1,\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2)\}} \ (\textsc{ROp})$$

$$\frac{\Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_1,\mathtt{u},\mathtt{p}+1\} \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_1,\mathtt{u},\mathtt{p}+1\} \qquad * \in \{<,>\}}{\Gamma \models (e_{1\mathbb{F}} \bowtie_{u,p} e_{2\mathbb{F}}, e_{1\mathbb{R}} \bowtie_{u,p} e_{2\mathbb{R}}) : \mathtt{bool}} \ (\textsc{RCmp})$$

$$\frac{\Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) : \mathtt{int} \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) : \mathtt{int} \qquad *_{\_} \in \{+_{\_}, -_{\_}, \times_{\_}, \div_{\_}\}}{\Gamma \models (e_{1\mathbb{F}} *_{\_} e_{2\mathbb{F}}, e_{1\mathbb{R}} *_{\_} e_{2\mathbb{R}}) : \mathtt{int}} \ (\textsc{IntOp})$$

$$\frac{\Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) : t \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) : t \qquad t \neq \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \qquad \bowtie \in \{=,\neq,<,>,\leq,\geq\}}{\Gamma \models (e_{1\mathbb{F}} \bowtie e_{2\mathbb{F}}, e_{1\mathbb{R}} \bowtie e_{2\mathbb{R}}) : \mathtt{bool}} \ (\textsc{ACmp})$$

$$\frac{\Gamma \models (e_{0\mathbb{F}}, e_{0\mathbb{R}}) \ : \ \mathtt{bool} \qquad \Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) \ : \ t_1 \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) \ : \ t_2 \qquad t = t_1 \sqcup t_2}{\Gamma \models (\mathtt{if}\ e_{0\mathbb{F}}\ \mathtt{then}\ e_{\mathbb{F}}1\ \mathtt{else}\ e_{2\mathbb{F}}, \mathtt{if}\ e_{0\mathbb{R}}\ \mathtt{then}\ e_{1\mathbb{R}}\ \mathtt{else}\ e_{2\mathbb{R}}) \ : \ t} \ (\textsc{Cond})$$

$$\frac{\Gamma, x : t_1 \models (e_{\mathbb{F}}, e_{\mathbb{R}}) : t_2}{\Gamma \models (\lambda x.e_{\mathbb{F}}, \lambda x.e_{\mathbb{R}}) : \Pi x : t_1.t_2} \ (\textsc{Abs}) \qquad \frac{\Gamma, x : t_1, f : \Pi.y : t_1.t_2 \models (e_{\mathbb{F}}, e_{\mathbb{R}}) : t_2}{\Gamma \models (\mathtt{rec}\ f\ x.e_{\mathbb{F}}, \mathtt{rec}\ f\ x.e_{\mathbb{R}}) : \Pi x : t_1.t_2} \ (\textsc{Rec})$$

$$\frac{\Gamma \models (e_{1\mathbb{F}}, e_{1\mathbb{R}}) : \Pi x : t_0.t_1 \qquad \Gamma \models (e_{2\mathbb{F}}, e_{2\mathbb{R}}) : t_2 \qquad t_2 \sqsubseteq t_0}{\Gamma \models (e_{1\mathbb{F}}\ e_{2\mathbb{F}}, e_{1\mathbb{R}}\ e_{2\mathbb{R}}) : t_1[x \mapsto e_2]} \ (\textsc{App})$$

Fig. 7. Simulation relation $\models$ used in our subject reduction theorem.

Formally, $\models$ is defined in Figure 7. These rules are similar to the typing rules of Figure 2 excepted that they operate on pairs $(e_{\mathbb{F}}, e_{\mathbb{R}})$. They are also designed for the language of Equation (13) and, consequently, deal with the elementary arithmetic operations $+$, $-$, $\times$ and $\div$ as well as the comparison operators. The difference between the rules of Figure 2 and Figure 7 is in Rule (VReal) which states that a `real` value $v_{\mathbb{R}}$ is correctly simulated by a value $v_{\mathbb{F}}$ up to accuracy $\mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ if $|v_{\mathbb{R}} - v_{\mathbb{F}}| < 2^{u-p+1}$. It is easy to show, by examination of the rules of Figure 2 and Figure 7 that

$$\Gamma \models (e_{\mathbb{F}}, e_{\mathbb{R}}) \ : \ t \implies \Gamma \vdash e_{\mathbb{F}} \ : \ t \ . \tag{15}$$

We introduce now Lemma 3 which asse rts the soundness of the type system for one reduction step. Basically, this lemma states that types are preserved by reduction and that concerning the values of type `real`, the distance between the finite precision value and the exact value is less than the ulp given by the type.

Lemma 3 (Weak subject reduction). *If* $\Gamma \models (e_{\mathbb{F}}, e_{\mathbb{R}}) \ : \ t$ *and if* $e_{\mathbb{F}} \rightarrow_{\mathbb{F}} e'_{\mathbb{F}}$ *and* $e_{\mathbb{R}} \rightarrow_{\mathbb{R}} e'_{\mathbb{R}}$ *then* $\Gamma \models (e'_{\mathbb{F}}, e'_{\mathbb{R}}) \ : t$.

Proof. By induction on the structure of expressions and case examination on the possible transition rules of Figure 6.

- If $e_{\mathbb{F}} \equiv e_{\mathbb{R}} \equiv \mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ then $\Gamma \models (\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}, \mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\}) : \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ and, from the reduction rules (FVal) and (RVal) of Figure 6, $\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \rightarrow_{\mathbb{F}} v_{\mathbb{F}}$ and $\mathtt{r}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \rightarrow_{\mathbb{R}} v_{\mathbb{R}}$ with $|v_{\mathbb{F}} - v_{\mathbb{R}}| < 2^{u-p+1}$. So $\Gamma \models (v_{\mathbb{F}}, v_{\mathbb{R}}) : \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$.
- If $e_{\mathbb{F}} \equiv e_{0\mathbb{F}} * e_{1\mathbb{F}}$ and $e_{\mathbb{R}} \equiv e_{0\mathbb{R}} * e_{1\mathbb{R}}$ then several cases must be distinguished.
  - If $e_{\mathbb{F}} \equiv v_{0\mathbb{F}} * v_{1\mathbb{F}}$ and $e_{\mathbb{R}} \equiv v_{0\mathbb{R}} * v_{1\mathbb{R}}$ then, by induction hypothesis, $\Gamma \models (v_{0\mathbb{F}}, v_{0\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_0,\mathtt{u}_0,\mathtt{p}_0\}$, $\Gamma \models (v_{1\mathbb{F}}, v_{1\mathbb{R}}) : \mathtt{real}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\}$ and, consequently, from Rule (VREAL),

$$|v_{0\mathbb{R}} - v_{0\mathbb{F}}| < 2^{u_0-p_0+1} \quad \text{and} \quad |v_{1\mathbb{R}} - v_{1\mathbb{F}}| < 2^{u_1-p_1+1} \ . \tag{16}$$

    Following Figure 4, the type t of e is

$$\begin{aligned} t = \big(&\Pi \mathtt{s}_1 : \mathtt{int}, \mathtt{u}_1 : \mathtt{int}, \mathtt{p}_1 : \mathtt{int}, \mathtt{s}_2 : \mathtt{int}, \mathtt{u}_2 : \mathtt{int}, \mathtt{p}_2 : \mathtt{int}. \\ &\quad \mathtt{real}\{\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1\} \rightarrow \mathtt{real}\{\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2\} \rightarrow \\ &\quad \rightarrow \mathtt{real}\{\mathcal{S}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{U}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{P}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1,\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2)\} \\ &\big)\ \mathtt{s}_1\ \mathtt{u}_1\ \mathtt{p}_1\ \mathtt{s}_2\ \mathtt{u}_2\ \mathtt{p}_2 \ , \\ = \ &\mathtt{real}\{\mathcal{S}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{U}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{s}_2,\mathtt{u}_2), \mathcal{P}_*(\mathtt{s}_1,\mathtt{u}_1,\mathtt{p}_1,\mathtt{s}_2,\mathtt{u}_2,\mathtt{p}_2)\} \\ = \ &\mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\} \end{aligned}$$

    By Rule (OP), $e \rightarrow_{\mathbb{F}} v_{\mathbb{F}}$ and $e \rightarrow_{\mathbb{R}} v_{\mathbb{R}}$ and, by Theorem 1, with the assumptions of Equation (16), we know that $|v_{\mathbb{R}} - v_{\mathbb{F}}| < 2^{u-p+1}$. Consequently, $\Gamma \models (v_{\mathbb{F}}, v_{\mathbb{R}}) : \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$.
  - If $e_{\mathbb{F}} \equiv v_{0\mathbb{F}} * v_{1\mathbb{F}}$ and $e_{\mathbb{R}} \equiv v_{0\mathbb{R}} * v_{1\mathbb{R}}$ with $\Gamma \models (v_0, v_1) : \mathtt{int}$ then, by Rule (Op), $e \rightarrow (v, v)$ and, by Equation (7), $\Gamma \vdash v : \mathtt{int}$. If $e \equiv e_0 * e_1$ then, by Rule (Op1), $e \rightarrow e_0 * e_1'$ and we conclude by induction hypothesis. The case $e \equiv e_0 * v_1$ is similar to the former one.
- If $e_{\mathbb{F}} \equiv e_{0\mathbb{F}} \bowtie_{u,p} e_{1\mathbb{F}}$ and $e_{\mathbb{R}} \equiv e_{0\mathbb{R}} \bowtie_{u,p} e_{1\mathbb{R}}$ then several cases have to be examined.
  - If $e_{\mathbb{F}} \equiv v_{0\mathbb{F}} \bowtie_{u,p} v_{1\mathbb{F}}$ and $e_{\mathbb{R}} \equiv v_{0\mathbb{R}} \bowtie_{u,p} v_{1\mathbb{R}}$ then by rules (FCmp) and (RCmp) $e_{\mathbb{F}} \rightarrow_{\mathbb{F}} b_{\mathbb{F}}$, $e_{\mathbb{R}} \rightarrow_{\mathbb{R}} b_{\mathbb{R}}$ with $b_{\mathbb{F}} = v_{0\mathbb{F}} - v_{1\mathbb{F}} \bowtie_{\{u,p\}} 2^{u-p+1}$ and $b_{\mathbb{R}} = v_{0\mathbb{R}} - v_{1\mathbb{R}} \bowtie_{\{u,p\}} 0$. By rule (RCmp) of Figure 7, $\Gamma \models (v_{0\mathbb{F}}, v_{1\mathbb{F}}) : \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ and $\Gamma \models (v_{0\mathbb{R}}, v_{1\mathbb{R}}) : \mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$. Consequently, $|v_{0\mathbb{R}} - v_{0\mathbb{F}}| < 2^{u-p+1}$ and $|v_{1\mathbb{R}} - v_{1\mathbb{F}}| < 2^{u-p+1}$. By combining the former equations, we obtain that $|(v_{0\mathbb{R}} - v_{1\mathbb{R}}) - (v_{0\mathbb{F}} - v_{1\mathbb{F}})| < 2^{u-p}$. Consequently, $b_{\mathbb{F}} = b_{\mathbb{R}}$ and we conclude that $\Gamma \models (b_{\mathbb{F}}, b_{\mathbb{R}}) : \mathtt{bool}$.
  - The other cases for $e_{\mathbb{F}} \equiv e_{0\mathbb{F}} \bowtie_{u,p} e_{1\mathbb{F}}$ are similar to the cases $e_{\mathbb{F}} \equiv v_{0\mathbb{F}} * v_{1\mathbb{F}}$ examined previously.
- The other cases simply follow the structure of the terms, by application of the induction hypothesis. □

Let $\rightarrow_{\mathbb{F}}^*$ (resp. $\rightarrow_{\mathbb{R}}^*$) denote the reflexive transitive closure of $\rightarrow_{\mathbb{F}}$ (resp. $\rightarrow_{\mathbb{R}}$). Theorem 2 expresses the soundness of our type system for sequences of reduction of arbitrary length.

Theorem 2 (Subject reduction). *If $\Gamma \models (e_{\mathbb{F}}, e_{\mathbb{R}}) : t$ and if $e_{\mathbb{F}} \rightarrow_{\mathbb{F}}^* e_{\mathbb{F}}'$ and $e_{\mathbb{R}} \rightarrow_{\mathbb{R}}^* e_{\mathbb{R}}'$ then $\Gamma \models (e_{\mathbb{F}}', e_{\mathbb{R}}') : t$.*

Proof. By induction on the length of the reduction sequence, using Lemma 3. □

Theorem 2 asserts the soundness of our type system. It states that the evaluation of an expression of type $\mathtt{real}\{\mathtt{s},\mathtt{u},\mathtt{p}\}$ yields a result of accuracy 2^{u-p+1}.

```
let rec unifyReal s1 u1 p1 s2 u2 p2 = match (!s1,!u1,!p1) with
  (int(s1'),int(u1'),int(p1')) →
    (match (!s2,!u2,!p2) with
       (int(s2'),int(u2'),int(p2')) →
         let s = if (s1'=s2') then s1' else 2 in
         let u = max u1' u2' in
         let p = if (u1'>=u2') then min p1' (u1' - u2' + p2')
                 else min p2' (u2' - u1' + p1')
         in if (p>0) then
              (s1 := int(s) ; s2 := int(s) ; u1 := int(u) ;
               u2 := int(u) ; p1 := int(p) ; p2 := int(p))
           else raise (Error ("Type "
           ^(printExpr (TFloat(s1,u1,p1)))^" is not compatible
           with type "^(printExpr (TFloat(s2,u2,p2))) ) )
     | (TypeVar(refS,strS),TypeVar(refU,strU),
        TypeVar(refP,strP)) → refS := Some(!s1) ;
                refU := Some(!u1) ; refP := Some(!p1)
     | _ → solveLT !s1 !s2 !u1 !u2 !p1 !p2
    )
| (TypeVar(refS,strS),TypeVar(refU,strU),
  TypeVar(refP,strP)) →
    ((match !refS with
        None → refS := Some(!s2)
      | Some(s1) → unify s1 !s2) ;
     (match !refU with
        None → refU := Some(!u2)
      | Some(u1) → unify u1 !u2) ;
     (match !refP with
        None → refP := Some(!p2)
      | Some(p1) → unify p1 !p2)
    )
| _ → (match (!s2,!u2,!p2) with
         (TypeVar(refS,strS),TypeVar(refU,strU),
         TypeVar(refP,strP)) →
            similar to previous case
       | _ → if ((s1=s2) && (u1=u2) && (p1=p2)) then ()
             else solve !s1 !s2 !u1 !u2 !p1 !p2
      )
```

Fig. 8. Unification procedure for types `real`.

5 Type System Implementation

In this section, we give some details about the implementation of our type system in `Numl`. Section 5.1 deals with the unification algorithm and Section 6 presents examples of typable programs in complement to the introductory examples of Section 2.

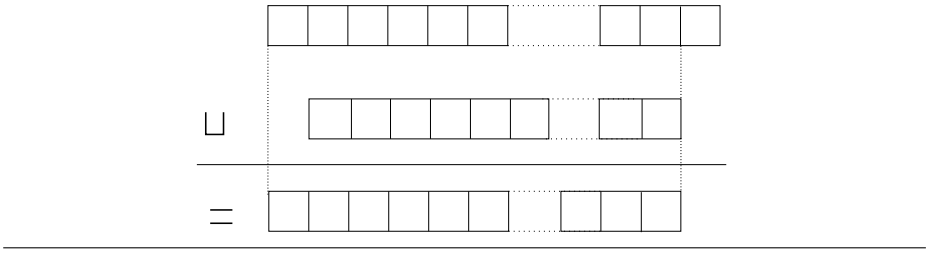

Fig. 9. The supremum operator $\sqcup$ of Equation (17).

5.1 Unification Algorithm

In this section, we describe how the type system introduced in Section 3 is implemented. Basically, we use a unification-based type inference in which type variables are represented by reference cells. The type `real` also stores the format $\{s, u, p\}$ into reference cells, so that it can be modified when unifying two terms of type `real`.

The type inference and unification algorithms are classical excepted for the unification of two `real` types, done by the function `unifyReal` displayed in Figure 8 and which requires, in certain cases, a call to a SMT solver (in practice we use `Z3` [18]). The function `unifyReal` takes as arguments the formats $\phi_1 = \{s_1, u_1, p_1\}$ and $\phi_2 = \{s_2, u_2, p_2\}$ of the types to be unified.

The function `unifyReal` calls in a mutually recursive way the function `unify` on terms. It also refers to type variables correspndig the constructor `TypeVar`. The fields of `TypeVar` are the value itself and a string corrsponding to the name of the variable. The value may be either `None` when the type variable is not constrained or some reference to an expression when a type has been given to the variable by unification. The function `solve` performs a partial evaluation of the expressions occurring in the equations, in order to simplify them, translates them for `Z3`, calls the SMT solver and then assign the values of the solution to the relevant type variables. The function `solveLT` acts just like the function `solve` but requires that the precision of the second expression is greater than or equal to the precision of the first expression instead of a strict equality. Several cases are distinguished in the function `unifyReal` of Figure 8:

- If ϕ_1 and ϕ_2 are fully instantiated, i.e. s_i, u_i and p_i, $1 \leq i \leq 2$ are integers then we assign $\phi = \phi_1 \sqcup \phi_2$ to ϕ_1 and ϕ_2. The supremum $\sqcup$ refers to the order $\sqsubseteq$ introduced in Equation (6). Formally, we have:

$$\phi_1 \sqcup \phi_2 = \{s_1 \uplus s_2, \max(u_1, u_2), p\} \text{ with } p = \begin{cases} \min(p_1, u_1 - u_2 + p_2) \text{ if } u_1 \geq u_2 \\ \min(p_2, u_2 - u_1 + p_1) \text{ otherwise} \end{cases}. \tag{17}$$

  In Equation (17), $\uplus$ computes the supremum of to values of Sign. Figure 9 illustrates the effect of the operator $\sqcup$.
- If ϕ_1 is fully instantiated and ϕ_2 is made of three type variables then ϕ_1 is assigned to ϕ_2.

– If ϕ_1 is fully instantiated and ϕ_2 is neither fully instantiated or a triple of type variables then $\phi - 2$ is made of three integer expressions containing type variables, $\phi_2 = \{e_0, e_1, e_2\}$. We have to solve the system

$$(S) \ : \ \begin{cases} s_1 = e_0 \\ u_1 = e_1 \\ p_1 = e_2 \end{cases} . \tag{18}$$

We call the SMT solver `Z3` to solve this system of equation. Recall that e_0, e_1 and e_2 come from the types of primitives introduced in Figure 4. These expressions are linear and are easy to solve for an SMT solver.
– If both ϕ_1 and ϕ_2 are made of type variables then we identify them in a pairwise manner.
– If both ϕ_1 and ϕ_2 are integer expressions then $\phi_1 = \{e_0, e_1, e_2\}$ and $\phi_2 = \{e'_0, e'_1, e'_2\}$. We have to solve the system

$$(S) \ : \ \begin{cases} e_0 = e'_0 \\ e_1 = e'_1 \\ e_2 = e'_2 \end{cases} . \tag{19}$$

Again, we call `Z3` to solve this system of linear equations.
– The other cases are symmetric to the ones detailled formerly, they are treated similarly.

For example, the equations sent to the SMT solver for the call `newton 9.0 0.0 g gprime ;;` to the `newton` function of Section 2 are given in Equation (20).

$$(S) \ : \ \begin{cases} \mathcal{S}_+('\mathtt{a}, \max('\mathtt{b}, -7) + \mathcal{S}_\times('\mathtt{a}, 1), 1, -7) = 2 \\ (\max('\mathtt{b}, -7) + \mathcal{S}_\times(\mathcal{S}_+('\mathtt{a}, '\mathtt{b}, 1, -7), 1) = 10 \\ \max(\max('\mathtt{b}, -7) + \mathcal{S}_\times(\mathcal{S}_+('\mathtt{a}, '\mathtt{b}, 1, -7), 1), -7) \\ +\mathcal{S}_\times(\mathcal{S}_+('\mathtt{a}, \max('\mathtt{b}, -7) + \mathcal{S}_\times('\mathtt{a}, 1), 1, -7), 1) \\ -\max(\max('\mathtt{b}, -7) + \mathcal{S}_\times(\mathcal{S}_+('\mathtt{a}, '\mathtt{b}, 1, -7), 1) - '\mathtt{c}, -60) \\ -\iota(\max('\mathtt{b}, -7) + \mathcal{S}_\times(\mathcal{S}_+('\mathtt{a}, '\mathtt{b}, 1, -7), 1) - '\mathtt{c}, -60) \leq 21 \end{cases} \tag{20}$$

These equations are encoded in `Z3` by expanding the operators $\max$, $\mathcal{S}_+$, $\mathcal{S}_\times$, and ι following the definitions of Figure 4. For example, the `Z3` encoding of the first equality of Equation (20) is displayed in Figure 10. Globally, the encoding of the three equations of Equation (20) is a 1007 lines long `Z3` file. `Z3` solves these equations in 0.215 seconds (average measured time on 5 executions).

6 Experiments

In this section, we report some experiments showing how our type system behaves in practice. Section 6.1 presents `Numl` implementations of usual mathematical formulas while Section 6.2 introduce a larger example demonstrating the expressive power of our type system.

```
(assert (= 2
  (ite (and (= a 0) (= 1 0)) 0
   (ite (and (= a 1) (= 1 1)) 1
    (ite (and (= a -1) (= 1 -1)) -1
     (ite (and (= a 1) (= 1 0)) 1
      (ite (and (= a 0) (= 1 1)) 1
       (ite (and (= a -1) (= 1 0)) -1
        (ite (and (= a 0) (= 1 -1)) -1
         (ite (and (= a 1) (= 1 -1))
  (ite (> (+ (ite (>= b -7) b -7)
   (ite (and (= a 0) (= 1 0)) 0
    (ite (and (= a 1) (= 1 -1)) -1
     (ite (and (= a -1) (= 1 1)) -1
      (ite (and (= a -1) (= 1 -1)) 1
       (ite (and (= a 1) (= 1 1)) 1
        (ite (and (= a 1) (= 1 0)) 0
         (ite (and (= a 0) (= 1 1)) 0
          (ite (and (= a -1) (= 1 0)) 0
  (ite (and (= a 0) (= 1 -1)) 0
       2))))))))) -7) 1
   (ite (< (+ (ite (>= b -7) b -7)
    (ite (and (= a 0) (= 1 0)) 0
     (ite (and (= a 1) (= 1 -1)) -1
      (ite (and (= a -1) (= 1 1)) -1
       (ite (and (= a -1) (= 1 -1)) 1
        (ite (and (= a 1) (= 1 1)) 1
         (ite (and (= a 1) (= 1 0)) 0
          (ite (and (= a 0) (= 1 1)) 0
  (ite (and (= a -1) (= 1 0)) 0
    (ite (and (= a 0) (= 1 -1)) 0
         2))))))))) -7) -1 2))

(ite (and (= a -1) (= 1 1))
 (ite (< (+ (ite (>= b -7) b -7)
  (ite (and (= a 0) (= 1 0)) 0
   (ite (and (= a 1) (= 1 -1)) -1
    (ite (and (= a -1) (= 1 1)) -1
     (ite (and (= a -1) (= 1 -1)) 1
      (ite (and (= a 1) (= 1 1)) 1
       (ite (and (= a 1) (= 1 0)) 0
        (ite (and (= a 0) (= 1 1)) 0
         (ite (and (= a -1) (= 1 0)) 0
          (ite (and (= a 0) (= 1 -1)) 0
           2))))))))) -7) 1
 (ite (> (+ (ite (>= b -7) b -7)
  (ite (and (= a 0) (= 1 0)) 0
   (ite (and (= a 1) (= 1 -1)) -1
    (ite (and (= a -1) (= 1 1)) -1
     (ite (and (= a -1) (= 1 -1)) 1
      (ite (and (= a 1) (= 1 1)) 1
       (ite (and (= a 1) (= 1 0)) 0
        (ite (and (= a 0) (= 1 1)) 0
         (ite (and (= a -1) (= 1 0)) 0
          (ite (and (= a 0) (= 1 -1)) 0
           2))))))))) -7) -1 2))
  2))))))))))
```

Fig. 10. Z3 encoding of the first equality of Equation (20).

6.1 Usual Mathematical Formulas

Our first examples concern usual mathematical formulas, to compute the volume of geometrical objects or formulas related to polynomials. These examples aim at showing that usual mathematical formulas are typable in our system. We start with the volume of the sphere and of the cone.

```
> let sphere r = (4.0 / 3.0) * 3.1415926{+,1,20} * r * r * r ;;
val sphere : real{'a,'b,'c} -> real{<expr>,<expr>,<expr>} = <fun>

> sphere 1.0 ;;
- : real{+,7,20} = 4.188

> let cone r h = (3.1415926{+,1,20} * r * r * h) / 3.0 ;;
val cone : real{'a,'b,'c} -> real{'a,'b,'c}
           -> real{<expr>,<expr>,<expr>} = <fun>

> cone 1.0 1.0 ;;
- : real{+,4,20} = 1.0472
```

We repeatedly define the function `sphere` with more precision in order to show the impact on the accuracy of the results. Note that the results now have 15 digits instead of the former 5 digits.

```
> let sphere r = (4.0 / 3.0) * 3.1415926535897932{+,1,53} * r * r * r ;;
val sphere : real{'a,'b,'c} -> real{<expr>,<expr>,<expr>} = <fun>

> sphere 1.0 ;;
- : real{+,7,52} = 4.1887902047863
```

The next examples concern polynomials. We start with the computation of the discriminant of a second degree polynomial.

```
> let discriminant a b c = b * b - 4.0 * a * c ;;
val discriminant : real{'a,'b,'c} -> real{'d,'e,'f} -> real{'g,'h,'i}
                   -> real{<expr>,<expr>,<expr>} = <fun>

> discriminant 2.0 -11.0 15.0 ;;
- : real{+,8,52} = 1.000000000000
```

Our last example concerning usual formulas is the Taylor series development of the sine function. In the code below, observe that the accuracy of the result is correlated to the accuracy of the argument. As mentioned in Section 2, error methods are neglected, only the errors due to the finite precision are calculated (indeed, $\sin \frac{\pi}{8} = 0.382683432\ldots$).

```
let sin x = x - ((x * x * x) / 3.0) + ((x * x * x * x * x) / 120.0) ;;
val sin : real{'a,'b,'c} -> real{<expr>,<expr>,<expr>} = <fun>

> sin (3.14{1,6} / 8.0) ;;
- : real{*,0,6} = 0.3

> sin (3.14159{1,18} / 8.0) ;;
- : real{*,0,18} = 0.37259
```


6.2 Newton-Raphson Method

In this section, we introduce a larger example to compute the zero of a function using the Newton-Raphson method. This example, which involves several higher order functions, shows the expressiveness of our type system. In the programming session below, we first define a higher order function `deriv` which takes as argument a function and computes its numerical derivative at a given point. Then we define a function `g` and compute the value of its derivative at point 2.0. Next, by partial application, we build a function computing the derivative of `g` at any point. Finally, we define a function `newton` which searches the zero of a function. The `newton` function is also an higher order function taking as argument the function for which a zero has to be found and its derivative.

```
> let deriv f x h = ((f (x + h)) - (f x)) / h ;;
val deriv : (real{<expr>,<expr>,<expr>} -> real{'a,'b,'c})
             -> real{<expr>,<expr>,<expr>} -> real{'d,'e,'f}
             -> real{<expr>,<expr>,<expr>} = <fun>

> let g x = (x*x) - (5.0*x) + 6.0 ;;
val g : real{'a,'b,'c} -> real{<expr>,<expr>,<expr>} = <fun>

> deriv g 2.0 0.01 ;;
```

```
- : real{*,5,51} = -0.9900000000000

> let gprime x = deriv g x 0.01 ;;
val gprime : real{<expr>,<expr>,<expr>} -> real{<expr>,<expr>,<expr>} = <fun>

> let rec newton x xold f fprime = if ((abs (x-xold))<0.01{*,10,20}) then x
                                   else newton (x-((f x)/(fprime x))) x f fprime ;;
val newton : real{*,10,21} -> real{0,10,20} -> (real{*,10,21} -> real{'a,'b,'c})
             -> (real{*,10,21} -> real{'d,'e,'f}) -> real{*,10,21} = <fun>

> newton 9.0 0.0 g gprime ;;
- : real{*,10,21} = 3.0001
```

We call the `newton` function with our function `g` and its derivative computed by partial application of the `deriv` function. We obtain a root of our polynomial `g` with a guaranteed accuracy. Note that while Newton-Raphson method converges quadratically in the reals, numerical errors may perturb the process [6].

7 Case of the IEEE754 Floating-Point Arithmetic

In this section, we introduce modified versions of the types of primitives introduced in Section 3.2. These modified versions are specific to the IEEE754 floating-point arithmetic [1]. The types introduced in Figure 4 for the primitives corresponding to the elementary operations $+$, $-$, $\times$ and $\div$ are not tailored for a specific arithmetic. They only assume that the system has enough bits to perform the operations in the format given by the types so that the results of the operations are not rounded. `Numl` interpreter fulfills this requirement by performing all the numerical computations in multiple precision, using the GNU Multiple Precision Arithmetic library `GMP`. Indeed, the type inference enables to determine *a priori* the precision needed by `GMP` for the values and arithmetic operations. An optimization would consists of also detecting when the computations fit into hardware formats (generally the formats of the IEEE754 arithmetic introduced in Figure 1) in order to avoid the calls to `GMP` when possible. The type information also permits to generate code for the fixed-point arithmetic [12]. In this case, if the precision of the formats corresponds to the types, no additional roundoff errors have to be added and the general equations of Figure 4 hold again. In future work, we plan to develop a compiler for our language (in addition to the current interpreter) which, based on the formats given by the types, generates code using either the IEEE754 or the multiple precision arithmetic (only when necessary). This compiler would also generate code for the fixed-point arithmetic.

In practice, in many cases, one wants to use the IEEE754 floating-point arithmetic and not multiple precision libraries, for efficiency reasons or because these library are not available in certain contexts. In this case, the values and the results of the operations do not necessarily fit inside the IEEE754 formats of Figure 1, they must be rounded. The IEEE754 Standard defines five rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero, to the nearest ties to even and to the nearest ties to away and we write them $\circ_{-\infty}$, $\circ_{+\infty}$, $\circ_0$, $\circ_{\sim_e}$ and $\circ_{\sim_a}$, respectively. The semantics of the elementary operations $* \in \{+,\ -,\ \times,\ \div\}$ is then defined by

$$f_1 *_\circ f_2 = \circ(f_1 * f_2) \tag{21}$$

$$\varrho \in \{11, 24, 53, 113\}$$

$$\mathcal{P}^{\varrho}_{+}(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) =$$

$$(\mathtt{u}_1, \mathtt{u}_2) + \sigma_+(\mathtt{s}_1, \mathtt{s}_2) - \max \begin{pmatrix} \mathtt{u}_1 - \mathtt{p}_1, \\ \mathtt{u}_2 - \mathtt{p}_2, \\ \mathcal{U}_+(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2) - \varrho \end{pmatrix} - \iota_3 \begin{pmatrix} \mathtt{u}_1 - \mathtt{p}_1, \\ \mathtt{u}_2 - \mathtt{p}_2, \\ \mathcal{U}_+(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2) - \varrho \end{pmatrix}$$

$$\mathcal{P}^{\varrho}_{-}(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) =$$

$$(\mathtt{u}_1, \mathtt{u}_2) + \sigma_-(\mathtt{s}_1, \mathtt{s}_2) - \max \begin{pmatrix} \mathtt{u}_1 - \mathtt{p}_1, \\ \mathtt{u}_2 - \mathtt{p}_2, \\ \mathcal{U}_-(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2) - \varrho \end{pmatrix} - \iota_3 \begin{pmatrix} \mathtt{u}_1 - \mathtt{p}_1, \\ \mathtt{u}_2 - \mathtt{p}_2, \\ \mathcal{U}_-(\mathtt{s}_1, \mathtt{u}_1, \mathtt{s}_2, \mathtt{u}_2) - \varrho \end{pmatrix}$$

$$\mathcal{P}^{\varrho}_{\times}(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) =$$

$$\mathtt{u}_1 + \mathtt{u}_2 + 1 - \max \begin{pmatrix} \mathtt{u}_1 + \mathtt{u}_2 + 1 - \mathtt{p}_1, \\ \mathtt{u}_1 - \mathtt{u}_2 + 1 - \mathtt{p}_2, \\ \mathtt{u}_1 - \mathtt{u}_2 + 1 - \varrho \end{pmatrix} - \iota_3 \begin{pmatrix} \mathtt{u}_1 + \mathtt{u}_2 + 1 - \mathtt{p}_1, \\ \mathtt{u}_1 - \mathtt{u}_2 + 1 - \mathtt{p}_2, \\ \mathtt{u}_1 - \mathtt{u}_2 + 1 - \varrho \end{pmatrix}$$

$$\mathcal{P}^{\varrho}_{\div}(\mathtt{s}_1, \mathtt{u}_1, \mathtt{p}_1, \mathtt{s}_2, \mathtt{u}_2, \mathtt{p}_2) = \mathcal{P}^{\varrho}_{\times}(\mathtt{u}_1, \mathtt{p}_1, \mathtt{u}_2, \mathtt{p}_2)$$

$$\iota_3(x, y, z) = \begin{cases} 1 \text{ if } x = y \ \vee \ x = z \ \vee \ y = z, \\ 0 \text{ otherwise.} \end{cases}$$

Fig. 11. Types of the IEEE754 floating-point arithmetic operators in precision ϱ.

where $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_{\sim_e}, \circ_{\sim_a}\}$ denotes the rounding mode. Equation (21) states that the result of a floating-point operation $*_\circ$ done with the rounding mode $\circ$ returns what we would obtain by performing the exact operation $*$ and next rounding the result using $\circ$. The IEEE754 Standard also specifies how the square root function must be rounded in a similar way to Equation (21) but does not specify the roundoff of other functions like sin, log, etc.

In the IEEE754 arithmetic, additional errors arise compared to the general context of Section 3.2 and the types of the primitives of Figure 4 must be modified to correctly model the errors of this specific arithmetic. The types of the IEEE754 primitives in precision $\varrho \in \{11, 24, 53, 113\}$, i.e. in half, single, double or quadruple precision, is given in Figure 11. We assume that the rounding mode is $\sim \in \{\sim_a, \sim_e\}$ (to the nearest.) These equations model the fact that the accuracy of the result is dominated by either the error on first operand or on the second operand or on the rounding of the result in precision ϱ. For example, the error on $x +_\sim y$ is $e_+ = \varepsilon(x) + \varepsilon(y) + \circ(x + y)$ with, by Equation (21),

$$\circ(x+y) < \frac{1}{2}\mathsf{ulp}(x+y) = \frac{1}{2}\mathsf{ufp}(x+y) - \varrho \ . \tag{22}$$

The types of the other operators are obtained in a similar way to the addition. Let us also note that in the IEEE754 floating-point arithmetic the constants may no longer be in any precision. They must fit one of the formats given the standard.

8 Related Work

Several approaches have been proposed to determine the best floating-point formats as a function of the expected accuracy on the results. Darulova and Kuncak use a forward static analysis to compute the propagation of errors [8]. If the computed bound on the accuracy satisfies the post-conditions then the analysis is run again with a smaller format until the best format is found. Note that in this approach, all the values have the same format (contrarily to our framework where each control-point has its own format). While Darulova and Kuncak develop their own static analysis, other static techniques [11, 24] could be used to infer from the forward error propagation the suitable formats. Chiang *et al.* [5] have proposed a method to allocate a precision to the terms of an arithmetic expression (only). They use a formal analysis via Symbolic Taylor Expansions and error analysis based on interval functions. In spite of our linear constraints, they solve a quadratically constrained quadratic program to obtain annotations.

Other approaches rely on dynamic analysis. For instance, the Precimonious tool tries to decrease the precision of variables and checks whether the accuracy requirements are still fulfilled [19, 23]. Lam *et al* instrument binary codes in order to modify their precision without modifying the source codes [14]. They also propose a dynamic search method to identify the pieces of code where the precision should be modified.

Finally other work focus on formal methods and numerical analysis. A first related research direction concerns formal proofs and the use of proof assistants to guaranty the accuracy of finite-precision computations [3, 13, 15]. Another related research direction concerns the compile-time optimization of programs in order to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [7, 20].

9 Conclusion

In this article, we have introduced a dependent type system able to infer the accuracy of numerical computations. Our type system allows one to type non-trivial programs corresponding to implementations of classical numerical analysis methods. Unstable computations are rejected by the type system. The consistency of typed programs is ensured by a subject reduction theorem. To our knowledge, this is the first type system dedicated to numerical accuracy. We believe that this approach has many advantages going from early debugging to compiler optimizations. Indeed, we believe that the usual type `float` proposed by usual `ML` implementations, and which is a simple clone of the type `int`, is too poor for numerical computations. We also believe that this approach is a credible alternative to static analysis techniques for numerical precision [8, 11, 24].

For the developer, our type system introduces few changes in the programming style, limited to giving the accuracy of the inputs of the accuracy of comparisons to allow the typing of certain recursive functions.

A first perspective to the present work is the implementation of a compiler for `Numl`. We aim at using the type information to select the most appropriate formats (the IEEE754 formats of Figure 1, multiple precisions numbers of the GMP library when needed or requested by the user or fixed-point numbers.) In the longer term, we also aim at introducing safe compile-time optimizations based on type preservation: an expression may be safely (from the accuracy point of view) substituted to another expression as long as both expressions are mathematically equivalent and that the new expression has a greater type than the older one in the sense of Equation (6). Finally, a second perspective is to integrate our type system into other applicative languages. In particular, it would be of great interest to have such a type system inside a language used to build critical embedded systems such as the synchronous language `Lustre` [4]. In this context numerical accuracy requirements are strong and difficult to obtain. Our type system could be integrated naturally inside `Lustre` or similar languages.

References

1. ANSI/IEEE: IEEE Standard for Binary Floating-point Arithmetic (2008)
2. Atkinson, K.: An Introduction to Numerical Analysis, 2nd Edition. Wiley (1989)
3. Boldo, S., Jourdan, J., Leroy, X., Melquiond, G.: Verified compilation of floating-point computations. J. Autom. Reasoning 54(2), 135–163 (2015)
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL. pp. 178–188. ACM Press (1987)
5. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamaric, Z.: Rigorous floating-point mixed-precision tuning. In: POPL. pp. 300–315. ACM (2017)
6. Damouche, N., Martel, M., Chapoutot, A.: Impact of accuracy optimization on the convergence of numerical iterative methods. In: LOPSTR'15. LNCS, vol. LNCS 9527, pp. 1–18. Springer (2015)
7. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. STTT 19(4), 427–448 (2017)
8. Darulova, E., Kuncak, V.: Sound compilation of reals. In: POPL'14. pp. 235–248. ACM (2014)
9. Denis, C., de Oliveira Castro, P., Petit, E.: Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In: ARITH'16. pp. 55–62. IEEE (2016)
10. Franco, A.D., Guo, H., Rubio-González, C.: A comprehensive study of real-world numerical bug characteristics. In: ASE. pp. 509–519. IEEE (2017)
11. Goubault, E.: Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In: SAS. LNCS, vol. 7935, pp. 1–3. Springer (2013)
12. Graphics, M.: Algorithmic C Datatypes, software version 2.6 edn. (2011), http://www.mentor.com/esl/catapult/algorithmic
13. Harrison, J.: Floating-point verification. J. UCS 13(5), 629–638 (2007)
14. 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. pp. 369–378. ACM (2013)
15. Lee, W., Sharma, R., Aiken, A.: On automatically proving the correctness of math.h implementations. PACMPL 2(POPL), 47:1–47:32 (2018)

16. Martel, M.: Floating-point format inference in mixed-precision. In: NFM. LNCS, vol. 10227, pp. 230–246 (2017)
17. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. MIT Press (1997)
18. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
19. Nguyen, C., Rubio-Gonzalez, C., Mehne, B., Sen, K., Demmel, J., Kahan, W., Iancu, C., Lavrijsen, W., Bailey, D.H., Hough, D.: Floating-point precision tuning using blame analysis. In: ICSE. ACM (2016)
20. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI. pp. 1–11. ACM (2015)
21. Pierce, B.C.: Types and programming languages. MIT Press (2002)
22. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press (2004)
23. 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: HPCNSA. pp. 27:1–27:12. ACM (2013)
24. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: FM. LNCS, vol. 9109, pp. 532–550. Springer (2015)