Certification of bounds on expressions involving rounded operators

Marc Daumas
ÉLIAUS (ÉA 3679 UPVD)
and
Guillaume Melquiond
LIP (UMR 5668 CNRS–ÉNS Lyon–INRIA)

Gappa is a tool designed to formally verify the correctness of numerical softwares and hardwares. It uses interval arithmetic and forward error analysis to bound mathematical expressions that involve rounded as well as exact operators. It then generates a theorem and its proof for each verified enclosure. This proof can be automatically checked with a proof assistant, such as Coq or HOL Light. It relies on the facts of a large companion library we have developed. This Coq library provides theorems dealing with addition, multiplication, division, and square root, for both fixed- and floating-point arithmetics. Gappa uses multiple-precision dyadic fractions for the endpoints of intervals and performs forward error analysis on rounded operators when necessary. When asked, Gappa reports the best bounds it is able to reach for a given expression in a given context. This feature can be used to identify where the set of facts and automatic techniques implemented in Gappa becomes insufficient. Gappa handles seamlessly additional properties expressed as interval properties or rewriting rules in order to establish more intricate bounds. Recent work showed that Gappa is suited to discharge proof obligations generated for small pieces of software. They may be produced by third-party tools and the first applications of Gappa use proof obligations written by designers or obtained from traces of execution.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Certification and Testing

General Terms: Interval Arithmetic, Floating-Point, Proof System

Additional Key Words and Phrases: Forward error analysis, Dyadic fraction, Coq, PVS, HOL Light, Proof obligation

1. INTRODUCTION

Gappa is a simple and efficient tool for formally certifying bounds in computer arithmetic [Revy 2006] and in the engineering of numerical software [de Dinechin et al. 2006; Melquiond and Pion 2007; Boldo et al. 2008] and hardware [Michard et al. 2006]. Gappa handles arithmetic expressions on real and rational numbers and their evaluations in computers on fixed- and floating-point data formats. Properties that are most often needed involve:

—ranges of rounded expressions to prevent exceptional behaviors (overflow, division by zero, and so on),

—ranges of absolute and/or relative errors to characterize the accuracy of results.

To the best of our knowledge, Gappa is a tool that was missing in computer arithmetic and related research areas. Gappa is not the first tool able to formally verify static ranges and error bounds. In particular, two other projects are currently mixing interval arithmetic and automatic proof checking [Gameiro and Manolios 2004; Daumas et al. 2005]. The first one uses ACL2 [Kaufmann et al. 2000] and

the second one uses PVS [Owre et al. 1992]. Gappa is, however, the first tool able to certify these bounds when the program relies on advanced numerical recipes like error compensation, iterative refinement, *etc.*

Gappa provides *invisible formal methods* [Tiwari et al. 2003] in the sense that it delivers formal certificates to users that are not expected to ever write any piece of proof in any formal proof system. It may become in the future a key asset for development teams that want to meet the highest Common Criteria Evaluated Assurance Level (EAL 7) [Schlumberger 2003; Rockwell Collins 2005] for numerical applications using fixed- and floating-point arithmetics.

Gappa considers and combines many techniques of fixed- and floating-point arithmetics modeled by dyadic fractions, multiple precision arithmetic, interval arithmetic, forward error analysis, expression rewriting, and automatic proof checking. Linking together these techniques and ensuring that the implemented pieces of software cooperate to cover a significant breadth of skills commonly used in computer arithmetic was certainly the challenge that prevented the development of competitors to Gappa. The functionalities of Gappa presented here show its potential in tackling generic problems that are unreachable with other available tools.

Countless efficient algorithms use symbolic computation or interval arithmetic to produce bounds on expressions but seldom provide gateways to automatic proof checkers. The continuing work on interval arithmetic [Neumaier 1990; Jaulin et al. 2001] has created a huge set of useful techniques to deliver accurate answers in a reasonable time. Each technique is adapted to a specific class of problems and most evaluations yield accurate bounds only if they are handled by the appropriate techniques in the appropriate order. Blending interval arithmetic and properties on dyadic fractions has also been heavily used in computer arithmetic [Rump et al. 2005].

Gappa as well as many other projects use computational power to combine educated searches and brute force explorations. It performs an exhaustive search on its built-in set of facts and techniques. It is also able to follow hints given by users to take into account new techniques.

Gappa is composed of two parts. First, a program written in C++, based on the Boost interval arithmetic library [Brönnimann et al. 2006] and MPFR [Fousse et al. 2005], verifies numeric properties given by the user. Along with these verifications, it generates formal certificates of their validity. Second, a companion library provides theorems with computable hypotheses. This set of theorems allows a proof assistant to interpret the formal certificates and hence to automatically check the validity of the numeric properties. The proof assistants we use are Coq [Huet et al. 2004] and HOL Light [Harrison 2000], but ongoing work shows that Gappa can generate formal certificates for other proof assistants.

Proofs generated by Gappa typically contain thousands to hundreds of thousands of lines. Some fully developed examples range from 4,800 lines [de Dinechin et al. 2006] to 755,009 lines [Boldo et al. 2008]. Gappa simplifies a valid proof once it has been produced in order to reduce the certification time, as in-depth proof checking is and will remain much slower than processor-native evaluation.

We first describe the input language of Gappa and we detail its built-in rewriting rules. We then present the set of theorems and interval operators Gappa relies on

to prove numeric properties and we describe how it interacts with proof checkers, extending [Daumas and Melquiond 2004]. We finish this report with perspectives, experiments, and concluding remarks.

2. DESCRIPTION OF THE INPUT LANGUAGE OF GAPPA

Consider for example that y is the result of a portion of code without loops and branches. The definition of y is an expression involving rounded operators and rounded constants. We may define Y (uppercase) as the exact answer without any rounding error. The expression Y is identical to y except that rounded operators are replaced by exact operators and rounded constants are replaced by exact constants. Constants may be defined either explicitly, e.g. 42, or by enclosing them in a range, e.g. Pi $\in [3.14, 3.15]$. In the second case, Gappa produces results valid for any real Pi between 3.14 and 3.15. In particular, the results are valid for the mathematical constant π.

If some numeric terms were considered negligible and were optimized out of the implementation y, these terms are introduced in Y. So the expression y gives the effectively computed value while the expression Y gives the ideal value y tries to approximate.

In order to certify the correctness of this code, we will possibly need

—an interval containing all the possible values of y to guarantee that y does not overflow and produces no invalid result (and similarly for all the sub-terms of y),
—an interval containing all the possible values of $y - Y$ or $(y - Y)/Y$ to guarantee that y is accurate and close to Y.

The grammar of the input language to Gappa is presented in Figure 1. It has been designed to efficiently express such needs. An input file is composed of three parts: a set of aliases (`PROG`, detailed in Section 2.1), the proposition to be proved (`PROP`, detailed in Section 2.2), and a set of hints (`HINTS`, detailed in Sections 2.3 and 2.4). When successful, Gappa produces a Coq or a HOL Light file with the proof of `PROP`. Its validity can be checked by Coq using the companion library and by HOL Light using a set of axioms until a companion library becomes available.

2.1 Definitions of aliases to describe programs (`PROG`)

Aliases of expressions are defined by constructions of the form `IDENT = REAL`. The identifiers then become available for later definitions, the proposition, and the hints. This construction is neither an equality nor an affectation but rather an alias. Gappa uses the identifier for its outputs and in the formal proof instead of machine generated names. An identifier cannot be aliased more than once, even if the right hand sides of both aliases are equivalent. Nor can it be aliased after being used as an unbounded variable. For example `b = a * 2; a = 1;` is not allowed. Using `flex` notation, `IDENT` tokens (a superset of `VARID` and `FUNID` tokens) are defined as

`{alpha}({alpha}|{digit}|_)*` with `alpha [a-zA-Z]` and `digit [0-9]`

Rounding operators are used in the arithmetic expressions describing numerical codes. They are real functions yielding rounded values according to the target data format (`precision` and `minimum_exponent`, or `lsb_weight`) and a predefined rounding mode amongst the ones presented in Table I. For modes that are not defined by

```
 0 $accept: BLOB $end
 1 BLOB: BLOB1 HINTS
 2 BLOB1: PROG '{' PROP '}'
 3 PROP: REAL '<=' SNUMBER
 4     | REAL 'in' '[' SNUMBER ',' SNUMBER ']'
 5     | REAL 'in' '?'
 6     | REAL '>=' SNUMBER
 7     | PROP '/\' PROP
 8     | PROP '\/' PROP
 9     | PROP '->' PROP
10     | 'not' PROP
11     | '(' PROP ')'
12 SNUMBER: NUMBER
13        | '+' NUMBER
14        | '-' NUMBER
15 INTEGER: NUMBER
16 SINTEGER: SNUMBER
17 FUNCTION_PARAM: SINTEGER
18               | IDENT
19 FUNCTION_PARAMS_AUX: FUNCTION_PARAM
20             | FUNCTION_PARAMS_AUX ',' FUNCTION_PARAM
21 FUNCTION_PARAMS: /* empty */
22                | '<' FUNCTION_PARAMS_AUX '>'
23 FUNCTION: FUNID FUNCTION_PARAMS
24 EQUAL: '='
25      | FUNCTION '='
26 PROG: /* empty */
27     | PROG IDENT EQUAL REAL ';'
28     | PROG '@' IDENT '=' FUNCTION ';'
-- /* failure rules skipped */
33 REAL: SNUMBER
34     | VARID
35     | IDENT
36     | FUNCTION '(' REALS ')'
37     | REAL '+' REAL
38     | REAL '-' REAL
39     | REAL '*' REAL
40     | REAL '/' REAL
41     | '|' REAL '|'
42     | 'sqrt' '(' REAL ')'
43     | 'fma' '(' REAL ',' REAL ',' REAL ')'
44     | '(' REAL ')'
45     | '+' REAL
46     | '-' REAL
47 REALS: REAL
48      | REALS ',' REAL
49 DPOINTS: SNUMBER
50        | DPOINTS ',' SNUMBER
51 DVAR: REAL
52     | REAL 'in' INTEGER
53     | REAL 'in' '(' DPOINTS ')'
54 DVARS: DVAR
55      | DVARS ',' DVAR
56 PRECOND: REAL '<>' SINTEGER
57        | REAL '<=' SINTEGER
58        | REAL '>=' SINTEGER
59        | REAL '<' SINTEGER
60        | REAL '>' SINTEGER
61 PRECONDS_AUX: PRECOND
62             | PRECONDS_AUX ',' PRECOND
63 PRECONDS: /* empty */
64         | '{' PRECONDS_AUX '}'
65 HINTS: /* empty */
66      | HINTS REAL '->' REAL PRECONDS ';'
67      | HINTS REALS '$' DVARS ';'
68      | HINTS '$' DVARS ';'
69      | HINTS REAL '~' REAL ';'
```

Fig. 1. Grammar of the input language to Gappa using `byson`

IEEE 754 standard [Stevenson et al. 1987] and its forthcoming revision, see [Even and Seidel 1999; Boldo and Melquiond 2008] and references herein. A floating-point format should also specify the maximum exponent and non-numeric values such as signed infinities and NaNs (Not-a-Number). Gappa's formalism as well as others introduced for automatic proof checking are based on real numbers and cope difficultly with propagating non-numeric values. This issue is consistently addressed by proving that no overflows nor division by 0 occur in numerical code.

Fixed- and floating-point rounding can be expressed with the following operators where rounding parameters `FUNCTION_PARAMS` are listed between angle brackets:

```
float< precision, minimum_exponent, rounding_direction >(...)
fixed< lsb_weight, rounding_direction >(...)
```

The syntax above can be abbreviated for the floating-point formats of Table II and for (fixed-point) integer arithmetic:

```
float< name, rounding_direction >(...)
int< rounding_direction >(...)
```

Aliases are permitted for rounding operators: Their definitions are prefixed by the `@` sign. Gappa does not allow quantification over a set of rounding modes but the alias mechanism allows to quickly switch from one rounding mode to another. Running Gappa 4 times with files differing only in 1 line is sufficient to prove a property for every rounding mode of the IEEE 754 standard.

Line 1 below defines the `rnd` function as rounding to nearest using IEEE 754 standard for 32 bit floating-point data. The example shows various ways of expressing rounded operations using the alternate constructions of `EQUAL`. When all the arithmetic operations on the right hand side of an alias are followed by the same rounding operator (as visible Line 2), this operator can be put once and for all on the left of the equal symbol (as presented Line 3).

```
1 @rnd = float< ieee_32, ne>;
2 y = rnd(x * rnd(1 - x));
3 z rnd= x * (1 - x);
```

Notice that Gappa will complain that y and z are two different names for the same expression. Indeed, Gappa tries to reuse as many user-defined names as possible in the generated proof. So the expression will be given the name y whenever it occurs in the proof. But this also means that there will not be any reference to z in the proof, which may be surprising to the user, hence the warning.

We will see in Section 2.3 that Gappa operates at the syntactical level. Therefore it cannot use even the simplest theorem of calculus to match mathematically equivalent expressions or identify common sub-expressions.

2.2 Formalizing the proposition (`PROP`) that Gappa proves

The proposition `PROP` that Gappa is expected to prove is written between brackets as presented below.

```
{ x - 2 in [-2,0] /\ (x + 1 in [0,2] -> y in [3,4])
  -> not x <= 1 \/ x + y in ? }
```

Table I. Rounding modes available in Gappa

Alias	Meaning
`zr`	toward zero
`aw`	away from zero
`dn`	toward minus infinity (down)
`up`	toward plus infinity
`od`	to odd mantissas
`ne`	to nearest, tie breaking to even mantissas
`no`	to nearest, tie breaking to odd mantissas
`nz`	to nearest, tie breaking toward zero
`na`	to nearest, tie breaking away from zero
`nd`	to nearest, tie breaking toward minus infinity
`nu`	to nearest, tie breaking toward plus infinity

Table II. Predefined floating-point formats available in Gappa

Alias	Meaning
`ieee_32`	IEEE-754 single precision
`ieee_64`	IEEE-754 double precision
`ieee_128`	IEEE-754 quadruple precision
`x86_80`	80 bit extended precision

It may contain any conjunction `/\`, disjunction `\/`, implication `->`, or negation of enclosures of expressions. Enclosures are either inequalities or bounded ranges on expressions `REAL`. Ranges may be left unspecified by using question marks instead of intervals. Endpoints of intervals and bounds of inequalities are numerical constants `SNUMBER`. These constants can either be written with the usual decimal notation (e.g. `11.2e-17`), with the C99 hexadecimal notation (e.g. `0xE1.3Ap-13`), or with a mixed notation: decimal mantissa / power-of-two exponent (e.g. `142b-3` $= 142 \cdot 2^{-3}$). This leads to the following `flex` notations for `SNUMBER`

```
(({integer}(\.{integer}?)?)|(\.{integer}))([eE][-+]?{integer})?
{integer}([bB][-+]?{integer})?
0x(({hexa}(\.{hexa}?)?)|(\.{hexa}))([pP][-+]?{integer})?
```

with `integer` matching `{digit}+` and `hexa` matching `[0-9a-fA-F]+`. In addition, `SNUMBER` can match several notations for zero.

Expressions `REAL` may contain numeric constants `SNUMBER`, identifiers `IDENT`/`VARID`, user-defined as well as built-in rounding operators `FUNCTION`, and arithmetic operators: addition, subtraction, multiplication, division, absolute value, square root, negation, and fused multiply-and-add. Identifiers that are not aliased to any expression (see Section 2.1) are assumed to be universally quantified. For instance, in the example above, the proposition has an implicit prefix

$$\forall\, x \in \mathbb{R} \qquad \forall\, y \in \mathbb{R}.$$

The goal of Gappa is to prove the whole logical proposition. If question marks are used in some expression enclosures, Gappa suggests intervals for these enclosures such that the proposition can be proved. In the example above, Gappa suggests $x + y \in [3, 5]$, which happens to be the tightest interval such that the proposition

holds. Question marks are mostly useful for the debugging and development of proofs, as Gappa then displays the bounds it has obtained for the given expressions. Unexpectedly large bounds may hint that Gappa needs some indications from the user (see Sections 2.3 and 2.4).

Since Gappa stores interval endpoints as dyadic fractions, it produces an error message when a goal contains an interval so tight that it has to be replaced with an empty interval. For example, Gappa is unable to prove the goal `13/10 in [1.3,1.3]`, as the empty set is the biggest representable subset of the set $\{1.3\}$.

The fact that bounds are numerical constants is not a strong limitation to the use of Gappa. For example, linear dependencies on intervals can be introduced by manipulating expressions: The enclosure $y - Y \in [-i \times 10^{-6}, i \times 10^{-6}]$ is not allowed, but the enclosure $(y-Y)/i \in [-10^{-6}, 10^{-6}]$ is allowed [Boldo et al. 2008].

2.3 Rewriting expressions to suppress some dependency effects (first use of `HINT`)

Let Y be an expression and y an approximation of Y. The absolute error is $y - Y$ and the relative error is $(y - Y)/Y$. As soon as Gappa has computed some ranges for y and Y, it naively computes an enclosing interval of $y-Y$ and $(y-Y)/Y$ using theorems on subtraction and division of intervals.

Unfortunately, expressions y and Y are strongly correlated and error ranges computed that way are useless. To suppress some dependency effects and reproduce many of the techniques used in numerical analysis and in computer arithmetic [Kahan 1965; Higham 2002; Boldo and Daumas 2004; de Dinechin et al. 2004], Gappa manipulates error expressions through a set of built-in pattern-matching as well as user-defined rewriting rules.

We assume that $y = \texttt{rnd}(a+b)$ and $Y = A + B$. Gappa uses the `sub_xals` rule to transform the absolute error $\texttt{rnd}(a+b) - (A+B)$ into $(\texttt{rnd}(a+b) - (a+b)) + ((a+b)-(A+B))$. It then finds an enclosure of the first term using a theorem on the `rnd` rounding operator. For the second term, Gappa performs a second rewrite with the `add_mibs` rule: $(a+b) - (A+B)$ is equal to $(a-A) + (b-B)$. This transformation gives sensible results, as long as a and b are close to A and B respectively.

Table III contains some of the rules Gappa tries to apply automatically. There are two kinds of rewriting rules. Rules of the first kind are presented in the upper part of the table, for example `add_firs`. They are meant to produce simpler expressions. Rules of the second kind are presented in the lower part of the table, for example `sub_xals`. They are used to reproduce common practices of computer arithmetic by introducing intermediate terms in expressions. In order for an expression to match an uppercase letter in such a rule, the expression that matches the same letter in lowercase has to be tagged as an approximation of the former.

In the example above, the rule `sub_xals` was applied because Gappa automatically tags $\texttt{rnd}(e)$ as an approximation of e, for any expression e, since `rnd` is a rounding operator. Gappa also creates such pairs for expressions that define absolute and relative errors in some hypotheses of a sub-formula of the proposition `PROP`.

Table III. Built-in rewriting rules available in Gappa

(1) Rules meant to produce simpler expressions

Rule	Before	After	Condition[1]
opp_mibs	$-a--b$	$-(a-b)$	$a \not\equiv b$
opp_mibq	$(-a--b)/-b$	$(a-b)/b$	$b \neq 0 \wedge a \not\equiv b$
add_mibs	$(a+b)-(c+d)$	$(a-c)+(b-d)$	$a \not\equiv c \wedge b \not\equiv d$
add_fils	$(a+b)-(a+c)$	$b-c$	$b \not\equiv c$
add_firs	$(a+b)-(c+b)$	$a-c$	$a \not\equiv c$
sub_mibs	$(a-b)-(c-d)$	$(a-c)+-(b-d)$	$a \not\equiv c \wedge b \not\equiv d$
sub_fils	$(a-b)-(a-c)$	$-(b-c)$	$b \not\equiv c$
sub_firs	$(a-b)-(c-b)$	$a-c$	$a \not\equiv c$
mul_fils	$ab-ac$	$a(b-c)$	$b \not\equiv c$
mul_firs	$ac-bc$	$(a-b)c$	$a \not\equiv b$
mul_mars	$ab-cd$	$a(b-d)+(a-c)d$	$a \not\equiv c \wedge b \not\equiv d$
mul_mals	$ab-cd$	$(a-c)b+c(b-d)$	$a \not\equiv c \wedge b \not\equiv d$
mul_mabs	$ab-cd$	$a(b-d)+(a-c)b+-((a-c)(b-d))$	$a \not\equiv c \wedge b \not\equiv d$
mul_mibs	$ab-cd$	$c(b-d)+(a-c)d+(a-c)(b-d)$	$a \not\equiv c \wedge b \not\equiv d$
mul_filq	$(ab-ac)/(ac)$	$(b-c)/c$	$ac \neq 0 \wedge b \not\equiv c$
mul_firq	$(ab-cb)/(cb)$	$(a-c)/c$	$bc \neq 0 \wedge a \not\equiv c$
div_mibq	$(a/b-c/d)/(c/d)$	$((a-c)/c-(b-d)/d)/(1+(b-d)/d)$	$bcd \neq 0 \wedge b \not\equiv d$
div_firq	$(a/b-c/b)/(c/b)$	$(a-c)/c$	$bc \neq 0 \wedge a \not\equiv c$
square_sqrt	$\sqrt{a} \times \sqrt{a}$	a	$a \geq 0$
sqrt_mibs	$\sqrt{a}-\sqrt{b}$	$(a-b)/(\sqrt{a}+\sqrt{b})$	$a \geq 0 \wedge b \geq 0 \wedge a \not\equiv b$
sqrt_mibq	$(\sqrt{a}-\sqrt{b})/\sqrt{b}$	$\sqrt{1+(a-b)/b}-1$	$a \geq 0 \wedge b > 0 \wedge a \not\equiv b$
err_fabq	$1+(a-b)/b$	a/b	$b \neq 0 \wedge a \not\equiv b$
addf_1	$a/(a+b)$	$1/(1+b/a)$	$a(a+b) \neq 0 \wedge a \neq 1$
addf_2	$a/(a+b)$	$1-1/(1+a/b)$	$b(a+b) \neq 0 \wedge a \neq 1$
addf_3	$a/(a-b)$	$1/(1-b/a)$	$a(a-b) \neq 0 \wedge a \neq 1$
addf_4	$a/(a-b)$	$1+1/(a/b-1)$	$b(a-b) \neq 0 \wedge a \neq 1$

(2) Rules meant to reproduce common practices of computer arithmetic

Rule	Before	After	Condition
add_xals	$a+b$	$(a-A)+(A+b)$	
add_xars	$c+a$	$(c+A)+(a-A))$	
sub_xals[2]	$a-b$	$(a-A)+(A-b)$	$a \not\equiv b \wedge A \not\equiv b$
sub_xals[2]	$b-A$	$(b-a)+(a-A)$	$A \not\equiv b \wedge a \not\equiv b$
sub_xars	$b-a$	$(b-A)+-(a-A)$	$b \not\equiv a$
mul_xals	ab	$(a-A)b+Ab$	
mul_xars	ba	$b(a-A)+bA$	
val_xabs	a	$A+(a-A)$	
val_xebs	A	$a+-(a-A)$	
val_xabq	a	$A(1+(a-A)/A)$	$A \neq 0$
val_xebq	A	$a/(1+(a-A)/A)$	$aA \neq 0$

[1] Condition $a \not\equiv b$ means that a and b are syntactically-different expressions. Condition $a(a+b) \neq 0$ means that Gappa has to compute enclosures of both a and $a+b$ and check neither of them contain zero before rewriting the expression.

[2] Both rules have the same name, since they instantiate the same theorem in the generated proof:

$$\forall x, y, z \in \mathbb{R}\ \forall I \in \mathbb{IF} \quad (x-y)+(y-z) \in I \Rightarrow x-z \in I$$

For example, on the following input, Gappa computes the bounds $[-1.1, 0.1]$ by evaluating the expression $(\lfloor x \rfloor - x) + (y - x)$, since it considers x to be an approximation of y, and $\lfloor x \rfloor$ an approximation of x.

```
@floor = int<dn>;
{ x - y in [-0.1,0.1] -> floor(x) - y in ? }
```

Thanks to its built-in rules to rewrite expressions and detect approximations, Gappa is able to automatically verify most properties on numerical applications that use common practices. Gappa, however, is not a complete decision procedure[3] and it may fail to prove some propositions. When that happens, users can give some hints to the tool.

The first kind of hint allows the user to tag some expressions as approximations of other expressions. In the previous example, had Gappa not known that $\lfloor x \rfloor$ is an approximation of x, the user could have written `floor(x) ~ x` in order to register this pair. This kind of hint is useful in order to lead Gappa toward the expressions that appear in user-defined rewriting rules.

These user-defined rules are the second kind of hint: `primary -> secondary`. This rule states that Gappa can use an enclosure of the `secondary` expression whenever it needs an enclosure of the `primary` expression. The following example describes Newton's relation between the reciprocal $\frac{1}{y}$ and its approximation $x \cdot (2 - x \cdot y)$.

```
x * (2 - x * y) - 1/y ->
  (x - 1/y) * (x - 1/y) * -y    { y <> 0 };
```

The bracketed expression `{ y <> 0 }` tells Gappa that the hint can be used only after having proved that y is not zero.

Such rules usually make explicit some techniques applied by designers that are not necessarily visible in the source code. We cannot expect an automatic tool to re-discover innovative techniques. Yet, we will incorporate in Gappa any technique that we find to be commonly used.

In order for the `primary -> secondary` rule to be valid, any value of `primary` must be contained in the computed enclosure of `secondary`. This property generally holds if both expressions are equal. As a consequence, Gappa tries to check if they are equal and warns if they are not, in order to detect mistypings early. Note that Gappa does not prove that divisors are always different from zero before applying user-defined rewriting rules, unless requested in the brackets on the right of the rules. Any user-defined rewriting rule produces an hypothesis in the generated proof. For the example above, the generated certificate hence depends on the theorem

$$\forall I \in \mathbb{IF} \quad y \neq 0 \Rightarrow (x - 1/y)^2 \cdot (-y) \in I \Rightarrow x \cdot (2 - x \cdot y) - 1/y \in I.$$

Numerical issues are only a part of the complete verification of a program. In particular, the certificate generated by Gappa may be inserted in a bigger formal

[3]While seemingly simple, the formalism of Gappa is rich enough so that any quantifier-free first-order formula for Peano arithmetic can be expressed. As a consequence, it is impossible to design an algorithm that is able to automatically decide for all the propositions whether they are provable or not.

development. This will require the user to prove that the theorem above holds. Fortunately, this task is automated in many proof systems, as both expressions contained in I are provably equal when y is not zero.

2.4 Tiling the range of some quantities (second use of HINT)

Rewriting expressions is usually very efficient but it fails if different proof structures are needed on various parts of the range of some quantity, as in the example below. The objective is to prove that a bound is valid for any floating-point input $x \in [0, 3]$. We first state in Gappa that x is not any real number but a floating-point number by pretending that x is the rounded value of a dummy variable `x_` (which can be instantiated by x itself for example). For the following proposition, a straight error analysis only works for $x \in [0, 0.5]$. A specific proof is needed for $x \in [0.5, 3]$. It relies on the fact that $\mathtt{rnd}(y) - y$ is always zero there, since x is a floating-point number. Neither of these proof structures work on the other part of the domain, so Gappa cannot find a single proof structure that is valid for the whole domain $[0, 3]$. So this example introduces the last kind of hint that can be used when Gappa is unable to automatically prove a formula.

```
@rnd = float< ieee_32, ne >;
x = rnd(x_);
y = x - 1;
z = x * (rnd(y) - y);
{ x in [0,3] -> |z| <= 1b-26 }
|z| $ x;      # tiling hint
```

This kind of hint instructs Gappa to tile the range of some quantities and to prove independently the proposition `PROP` on each tile. In the example Gappa finishes the proof as soon as it is told (last line) that splitting the range of x allows to compute better bounds on $|z|$. There are three constructions for tiling, each involving a `$` sign in the hints section:

—Evenly split the range into as many sub-intervals as asked. E.g. `$ x in 6` splits the range of x in six sub-intervals. If the number of intervals is omitted (e.g. `$ x`) and no expression is present on the left of `$`, the default is 4.

—Split an interval on user-provided points. E.g. `$ x in (0.5,2)` splits the range $[0, 3]$ of x above in three sub-intervals, the middle one being $[0.5, 2]$.

—Look by dichotomy for a set of tiles such that a given enclosure of the proposition holds. (This is the kind of tiling used in the example above.) The enclosure has to be present in the proposition, since Gappa cannot guess its range. The target enclosure is specified by writing its expression on the left of the `$` symbol. If several expressions are written, the tiling will try to satisfy all of their enclosures.

Several tiling hints can be used in the same script. For instance, the two hints below will be used sequentially one after the other. The first one splits the range of u until all the enclosures on a, b, and c are verified. In practice, the order of the tiling operations may make the difference between success and failure, as tilings use a lot of memory to replicate the proof trees.

```
a, b, c $ u;
d, e $ v;
```

Users can build higher dimension tilings by using more than one term on the right of the `$` symbol. (This can quickly lead to combinatorial explosions though.) For example, the following statement asks Gappa to find tilings of u and w such that the enclosures on a and b are satisfied when the range of v is evenly split into three sub-intervals.

```
a, b $ u, v in 3, w
```

3. GAPPA'S ENGINE

3.1 Work on the logical proposition

The logical proposition the user wants to prove is first modified and loosely broken into sub-sequents according to the rules of sequent calculus. The objective is to get an equivalent set of logical propositions that are in a more suitable form. These new propositions are such that their left hand side is a conjunction of enclosures and such that their right hand side is a disjunction of trees of conjunctions and disjunctions of enclosures. In particular, all the negation symbols and the inner implications are removed. Hence implications of the form $e_1 \in I_1 \wedge \cdots \wedge e_m \in I_m \Rightarrow f_1 \in J_1 \vee \cdots \vee f_m \in J_m$ are suitable for an immediate use by Gappa.

For example, the proposition seen in Section 2.2

```
{ x - 2 in [-2,0] /\ (x + 1 in [0,2] -> y in [3,4])
  -> not x <= 1 \/ x + y in ? }
```

is transformed into these two propositions:

$$x \leq 1 \wedge x - 2 \in [-2, 0] \implies x + 1 \in [0, 2] \vee x + y \in \texttt{?}$$
$$x \leq 1 \wedge x - 2 \in [-2, 0] \wedge y \in [3, 4] \implies x + y \in \texttt{?}$$

Gappa then verifies that both propositions hold, in order to prove that the original proposition does.

Unspecified ranges (question marks) are allowed as long as they appear only on the right hand sides of these decomposed formulas. Indeed, Gappa is not able to guess hypotheses from conclusions, so it would only propose the whole set of real numbers for these ranges, which is correct yet useless.

Inequalities may appear on both sides of the new propositions. Any inequality on the left hand side will be used only if Gappa can compute an enclosure of the expression by some other means. If an inequality appears at the top level of the disjunction on the right hand side, it is copied to the hypotheses as permitted by classical logic, provided that it is reverted first. For example, proposition

$$x \in [2, 3] \Rightarrow (y \in [4, 5] \vee z \geq 6)$$

is equivalent to proposition

$$(x \in [2, 3] \wedge z \leq 6) \Rightarrow (y \in [4, 5] \vee z \geq 6),$$

but the second one provides a bigger set of usable enclosures on its left hand side.

Note that support for disjunctions is currently limited: When the right hand side of the formula is a disjunction of several enclosures, Gappa searches for an enclosure that holds under the hypotheses of the proposition. For instance, Gappa is unable

Table IV. Interval operators used in Table V

(a) Arithmetic operators

Operator	Constraint	Definition
$-I$		$[-\overline{I}, -\underline{I}]$
I^{-1}	$0 \notin I$	$[1/\overline{I}, 1/\underline{I}]$
$I + J$		$[\underline{I} + \underline{J}, \overline{I} + \overline{J}]$
$I - J$		$I + (-J)$
$I \times J$		$[\min(\underline{I}\underline{J}, \underline{I}\overline{J}, \overline{I}\underline{J}, \overline{I}\overline{J}), \max(\underline{I}\underline{J}, \underline{I}\overline{J}, \overline{I}\underline{J}, \overline{I}\overline{J})]$
I/J	$0 \notin J$	$I \times J^{-1}$
$\sqrt{I}$	$I \geq 0$	$\left[\sqrt{\overline{I}}, \sqrt{\underline{I}}\right]$
$\lvert I \rvert$		I if $I \geq 0$, $\quad -I$ if $I \leq 0$, $\quad [0, \max(-\underline{I}, \overline{I})]$ otherwise

(b) Operators that may (or may not) be defined for each rounding mode `rnd` of Table I

Operator	Constraint	Definition
$\texttt{rnd}(I)$		Bound the rounded value based on a bound on the exact value
$\texttt{err}_{\texttt{rnd},0}(I)$		Uniform bound on the absolute rounding error for fixed-point arithmetic
		Bound the absolute rounding error based on a bound on the
$\texttt{err}_{\texttt{rnd},1}(I)$		exact value
$\texttt{err}_{\texttt{rnd},2}(I)$		rounded value
$\texttt{err}_{\texttt{rnd},3}(I)$		magnitude of the exact value
$\texttt{err}_{\texttt{rnd},4}(I)$		magnitude of the rounded value
		Bound the relative rounding error based on a bound on the
$\texttt{err}_{\texttt{rnd},5}(I)$	$0 \notin I$	exact value
$\texttt{err}_{\texttt{rnd},6}(I)$	$0 \notin I$	rounded value
$\texttt{err}_{\texttt{rnd},7}(I)$	$0 \notin I$	magnitude of the exact value
$\texttt{err}_{\texttt{rnd},8}(I)$	$0 \notin I$	magnitude of the rounded value

to prove the valid proposition $x \in [0, 2] \Rightarrow x \in [0, 1] \vee x \in [1, 2]$, as neither $x \in [0, 1]$ nor $x \in [1, 2]$ hold under hypothesis $x \in [0, 2]$.

3.2 Properties on expressions

Enclosure (`BND`) is the only predicate available to users but Gappa internally relies on more predicates to describe properties on an expression x. Such predicates appear in intermediate lemmas of generated proofs.

$$\begin{aligned}
\texttt{BND}(x, I) &\equiv x \in I \\
\texttt{ABS}(x, I) &\equiv |x| \in I \wedge I \geq 0 \\
\texttt{FIX}(x, e) &\equiv \exists m \in \mathbb{Z}, x = m \cdot 2^e \\
\texttt{FLT}(x, p) &\equiv \exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge |m| < 2^p
\end{aligned}$$

The `FIX` and `FLT` predicates express that the set of computer numbers is generally a discrete subset of the real numbers, while intervals only consider connected subsets. They are especially useful for automatically detecting rounded operations that actually are exact operations, and hence do not contribute any rounding error.

Table V lists most of the theorems used by Gappa. These theorems rely on some

Table V. Theorems on interval arithmetic available from the Coq companion library to Gappa

Target	Hypotheses	Constraint
$\texttt{BND}(\texttt{rnd}(a)-a, I)$		$I \supset \texttt{err}_{\texttt{rnd},0}$
$\texttt{BND}(\texttt{rnd}(a)-a, I)$	$\texttt{BND}(a, J)$	$I \supset \texttt{err}_{\texttt{rnd},1}(J)$
$\texttt{BND}(\texttt{rnd}(a)-a, I)$	$\texttt{BND}(\texttt{rnd}(a), J)$	$I \supset \texttt{err}_{\texttt{rnd},2}(J)$
$\texttt{BND}(\texttt{rnd}(a)-a, I)$	$\texttt{ABS}(a, J)$	$I \supset \texttt{err}_{\texttt{rnd},3}(J)$
$\texttt{BND}(\texttt{rnd}(a)-a, I)$	$\texttt{ABS}(\texttt{rnd}(a), J)$	$I \supset \texttt{err}_{\texttt{rnd},4}(J)$
$\texttt{BND}((\texttt{rnd}(a)-a)/a, I)$	$\texttt{BND}(a, J)$	$I \supset \texttt{err}_{\texttt{rnd},5}(J)$
$\texttt{BND}((\texttt{rnd}(a)-a)/a, I)$	$\texttt{BND}(\texttt{rnd}(a), J)$	$I \supset \texttt{err}_{\texttt{rnd},6}(J)$
$\texttt{BND}((\texttt{rnd}(a)-a)/a, I)$	$\texttt{ABS}(a, J)$	$I \supset \texttt{err}_{\texttt{rnd},7}(J)$
$\texttt{BND}((\texttt{rnd}(a)-a)/a, I)$	$\texttt{ABS}(\texttt{rnd}(a), J)$	$I \supset \texttt{err}_{\texttt{rnd},8}(J)$
$\texttt{BND}(\texttt{rnd}(a), I)$	$\texttt{BND}(a, J)$	$I \supset \texttt{rnd}(J)$
$\texttt{BND}(\texttt{rnd}(a), I)$	$\texttt{BND}(\texttt{rnd}(a), J)$	$I \supset J \cap \mathbb{F}_{\texttt{rnd}}$
$\texttt{BND}(-a, I)$	$\texttt{BND}(a, J)$	$I \supset -J$
$\texttt{BND}(\lvert a\rvert, I)$	$\texttt{BND}(a, J)$	$I \supset \lvert J\rvert$
$\texttt{BND}(\sqrt{a}, I)$	$\texttt{BND}(a, J)$	$J \geq 0 \wedge I \supset \sqrt{J}$
$\texttt{BND}(a-a, I)$		$0 \in I$
$\texttt{BND}(a/a, I)$	$\texttt{ABS}(a, J)$	$1 \in I \wedge J > 0$
$\texttt{BND}(a \times a, I)$	$\texttt{BND}(a, J)$	$I \supset \lvert J\rvert \times \lvert J\rvert$
$\texttt{BND}(a+b, I)$	$\texttt{BND}(a, J), \texttt{BND}(b, K)$	$I \supset J + K$
$\texttt{BND}(a-b, I)$	$\texttt{BND}(a, J), \texttt{BND}(b, K)$	$I \supset J - K$
$\texttt{BND}(a \times b, I)$	$\texttt{BND}(a, J), \texttt{BND}(b, K)$	$I \supset JK$
$\texttt{BND}(a/b, I)$	$\texttt{BND}(a, J), \texttt{BND}(b, K)$	$0 \notin K \wedge I \supset J/K$
$\texttt{ABS}(-a, I)$	$\texttt{ABS}(a, J)$	$I \supset J$
$\texttt{ABS}(\lvert a\rvert, I)$	$\texttt{ABS}(a, J)$	$I \supset J$
$\texttt{ABS}(\sqrt{a}, I)$	$\texttt{ABS}(a, J)$	$I \supset \sqrt{J}$
$\texttt{ABS}(a \pm b, I)$	$\texttt{ABS}(a, J), \texttt{ABS}(b, K)$	$I \supset \lvert J-K\rvert \cup (J+K)$
$\texttt{ABS}(a \times b, I)$	$\texttt{ABS}(a, J), \texttt{ABS}(b, K)$	$I \supset J \times K$
$\texttt{ABS}(a/b, I)$	$\texttt{ABS}(a, J), \texttt{ABS}(b, K)$	$K > 0 \wedge I \supset J/K$
$\texttt{BND}(a, I)$	$\texttt{ABS}(a, J)$	$I \supset J \cup -J$
$\texttt{BND}(a, I)$	$\texttt{BND}(a, J), \texttt{ABS}(a, K)$	$I \supset (J \cap K) \cup (J \cap -K)$
$\texttt{BND}(\lvert a\rvert, I)$	$\texttt{ABS}(a, J)$	$I \supset J$
$\texttt{ABS}(a, I)$	$\texttt{BND}(\lvert a\rvert, J)$	$I \supset J$
$\texttt{BND}(\xi, I)$		$I \supset \{\xi\}$ (ξ is a constant)
$\texttt{FIX}(a \pm b, e)$	$\texttt{FIX}(a, f), \texttt{FIX}(b, g)$	$e \leq \min(f, g)$
$\texttt{FIX}(a \times b, e)$	$\texttt{FIX}(a, f), \texttt{FIX}(b, g)$	$e \leq f + g$
$\texttt{FLT}(a \times b, p)$	$\texttt{FLT}(a, q), \texttt{FLT}(b, r)$	$p \geq q + r$
$\texttt{FIX}(a, e)$	$\texttt{FLT}(a, q), \texttt{ABS}(a, J)$	$J > 0 \wedge e \leq 1 + \log_2(\underline{J}) - q$
$\texttt{FLT}(a, p)$	$\texttt{FIX}(a, e), \texttt{ABS}(a, J)$	$p \geq 1 + \log_2(\overline{J}) - e$
$\texttt{FIX}(a, e)$	$\texttt{BND}(a, [x, x])$	$\exists m \in \mathbb{Z}, x = m \cdot 2^e$
$\texttt{FLT}(a, p)$	$\texttt{BND}(a, [x, x])$	$\exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge \lvert m\rvert < 2^p$
$\texttt{FIX}(\texttt{rnd}(a), e)$		$e \leq e_{\texttt{rnd}}$
$\texttt{FLT}(\texttt{rnd}(a), p)$		$p \geq p_{\texttt{rnd}}$
$\texttt{BND}(\texttt{rnd}(a)-a, I)$	$\texttt{FIX}(a, e), \texttt{FLT}(a, p)$	$0 \in I \wedge e \geq e_{\texttt{rnd}} \wedge p \leq p_{\texttt{rnd}}$

interval operators defined in Table IV. Several interval operators $\texttt{err}(\cdot)$ relate to the rounding modes. For a given rounding $\texttt{rnd}$, $\texttt{err}_{\texttt{rnd},1}(I)$ computes a bound on the absolute error $\texttt{rnd}(x) - x$ reached when rounding a value $x \in I$, while another operator $\texttt{err}_{\texttt{rnd},6}(I)$ computes a bound on the relative error $\frac{\texttt{rnd}(x)-x}{x}$ for $\texttt{rnd}(x) \in I$, and so on. Not all of the nine operators have to be defined for a given rounding: Gappa will use whatever operators are available, as they are partly redundant.

Gappa proceeds by proving properties on real-valued expressions (floating-point variables, non-rounded values, absolute errors, relative errors, and so on). For instance, given an expression, it tries to evaluate its attached range. This range is the intersection of all the intervals obtained by applying theorems or rewriting rules concerning the expression. When a new or a tightened interval is produced for an expression, Gappa reapplies all the theorems whose hypotheses depend on an enclosure of this expression. In turn, this may produce or tighten enclosures of other expressions. This updating process continues until Gappa has obtained a proof of the logical proposition or until all the ranges have stopped evolving. In the later case, either the logical proposition is incorrect, or the tool needs some additional help in order to prove it.

As Gappa keeps track of its computations in order to later generate a formal proof, the exploration may exhaust memory if it does not reach a stable state or a proof of the proposition. This is a common failure of Gappa on huge problems.

3.3 Handling automatic proof checker

From that point, generating a formal proof is simple: Gappa dumps the definitions, properties, and lemmas, as they are needed. This is performed in such a way that, as each lemma appears, everything it depends on has already been dumped into the script. So a lemma is nothing more than an application of the appropriate theorem in the companion library.

Let us assume that Gappa has used an interval addition in order to prove the proposition "if $x \in [1, 2]$ (property `p1`) and $y \in [3, 4]$ (property `p2`), then $x+y \in [0, 6]$ (property `p3`)". The proof script generated for Coq then contains the following lemma, which corresponds to an instance of the line `BND`$(a + b, I)$ of Table V.

```
1 Lemma l1 : p1 -> p2 -> p3.
2  intros h0 h1.
3  apply add with (1 := h0) (2 := h1) ; finalize.
4 Qed.
```

The first line defines the lemma: If the hypotheses `p1` and `p2` hold, the property `p3` holds too. The second line starts the proof in a suitable state by using the `intros` tactic of Coq. The third line applies the `add` theorem of Gappa support library with the `apply` tactic.

The `add` theorem is stated as follows. The `lower` and `upper` functions return the lower and the upper bound of an interval. Intervals are pairs of dyadic fractions (`FF` or $\mathbb{IF}$). `Fplus2` is the addition of dyadic fractions. `Fle2` compares two dyadic fractions (less or equal) and returns a boolean value. The `BND` predicate holds, when its first argument, an expression on real numbers, is an element of its second argument, an interval defined by dyadic fraction bounds.

```
Definition add_helper (xi yi zi : FF) :=
 Fle2 (lower zi) (Fplus2 (lower xi) (lower yi)) &&
 Fle2 (Fplus2 (upper xi) (upper yi)) (upper zi).

Theorem add :
 forall x y : R, forall xi yi zi : FF,
 BND x xi -> BND y yi ->
```

```
add_helper xi yi zi = true ->
BND (x + y) zi.
```

The mathematical expression of the theorem is as follows:

$$\begin{aligned}\texttt{add}: \; & \forall x, y \in \mathbb{R}, \quad \forall I_x, I_y, I_z \in \mathbb{IF}, \\ & x \in I_x \Rightarrow y \in I_y \Rightarrow \\ & f_{\texttt{add}}(I_x, I_y, I_z) = \mathit{true} \Rightarrow \\ & x + y \in I_z.\end{aligned}$$

Hypothesis $f_{\texttt{add}}(I_x, I_y, I_z) = \mathit{true}$ implies $I_x + I_y \subseteq I_z$. But since its left hand side is a boolean expression, it can be evaluated by the proof checker in order to automatically verify that I_z indeed contains the sum $x + y$. In lemma `l1`, this evaluation is triggered by the `finalize` tactic which checks that the current goal can be reduced to $\mathit{true} = \mathit{true}$. This concludes the formal proof.

All the theorems of Gappa's companion library are built the same way: Instead of having standard hypotheses that Coq would be unable to automatically decide, every theorem relies on a computable boolean expression. The companion library formally proves that, when this expression evaluates to *true*, the standard hypotheses hold, and hence the goal of the theorem applies. This approach is a simpler form of reflection techniques [Boutin 1997]. Although the use of booleans seems to restrict the use of Gappa to the Coq proof checker, the interval arithmetic library [Muñoz and Lester 2005; Daumas et al. 2005] developed for PVS shows that proofs through interval computations are also attainable to other proof assistants.

3.4 Dyadic fractions as interval bounds

In order to get computable boolean expressions, a decidable subset of the real numbers was chosen to represent interval bounds in the generated certificates. One possibility would have been the rational numbers. But we decided for a simpler subset: the dyadic fractions ($m \cdot 2^e$ with m and e relative integers). As with the rational numbers, they can efficiently be added, multiplied, and compared. But the certification process may also have to emulate some of the rounding operations occurring in numerical codes, and dyadic fractions are the simplest representation for this purpose. Moreover, they have a smaller footprint when manipulating bounds on rounding errors. For instance, 2^{-53} would be written 1/9007199254740992 as a rational number.

In order to check the certificate, the proof assistant does not have to perform the exact same computations than Gappa and to check that their results match Gappa's ones. It is sufficient to check that Gappa's results are compatible with the computations. For instance, if the certificate states that the expression $\sqrt{3}$ is enclosed in $[1, 2]$, the proof assistant does not have to compute the real number $\sqrt{3}$ in order to verify it. It just has to check that the inequalities $0 \leq 2$ and $1^2 \leq 3 \leq 2^2$ hold. Therefore, the proof assistant does not have to perform any division or square root on the interval bounds. Addition and multiplication are sufficient when checking a Gappa certificate.

While Gappa performs its own computations with 60-bit precision (and the user can request a higher precision), the bounds stored in the generated certificates do not have to be that precise. Any wider interval with less precise bounds can be

used, as long as no step of the certificate is invalidated. For instance, if Gappa has to prove that $\sqrt{3}$ is not zero because it appears as a denominator, it computes an interval enclosure of $\sqrt{3}$ with 60 bits and checks that the lower bound is positive. Later, when generating the certificate, it will look for a simpler enclosure. It will notice that $\sqrt{3} \in [1, 2]$ is sufficient to prove that $\sqrt{3}$ is not zero. Expanding the interval enclosure of $\sqrt{3}$ may, however, invalidate other parts of the certificate that were relying on a tighter enclosure. In order to avoid this issue, Gappa first finds a correct proof path and then it greedily operates backwards from the last proved results to the first proved results, widening the intervals along the way.

Such simplifications are important, since a proof checker like Coq is considerably slower than a specialized mathematical library. As a consequence, these simplified numbers can considerably speed up the verification process of propositions, especially when they involve error bounds. For the example of Listing 1 with an error bound $|e - E_0| \leq 137 \cdot 2^{-31}$, the simplified numbers incur a 40% speed-up of the Coq verification.

These considerations are also true for case studies: Searching for a better set of (possibly overlapping) tiles and certifying it, is always faster than directly certifying the first tiling that has been found by Gappa. The time spent by Gappa in doing all the computations over and over in order to find a better set of tiles is negligible in comparison to the time necessary to certify the property on one single tile with a proof checker.

4. PERSPECTIVES AND CONCLUDING REMARKS

In our approach to program certification, generation of proof obligations, proof generation, and proof verification are distinct steps. The intermediate step is performed by Gappa with its own computational methods, and the last one is done by a proof checker with the help of our support library.

The developments presented so far already allowed us to guarantee the correct behavior of many useful functions. As we continue using Gappa, we may discover practices that cannot be handled appropriately. We will extend Gappa, should this become necessary. Our software, a user's guide and details of some projects using Gappa are available on the Internet at the address below.

`http://lipforge.ens-lyon.fr/www/gappa/`

Gappa is used to certify CRlibm, a library of elementary functions with correct rounding in the four IEEE-754 rounding modes and performances comparable to standard mathematical libraries [de Dinechin et al. 2006; de Dinechin et al. 2004]. Listing 1 presents the input file needed to reproduce some parts of an earlier validation in HOL Light [Harrison 1997]. These expressions define an almost correctly-rounded exponential function in single precision [Tang 1989]. Gappa produces in less than 2 seconds a proof for Coq outlined in Listings 2. The output language of Gappa has not been presented since it is a subset of the Coq or HOL Light input language.

Gappa has also been used to develop robust semi-static filters for the CGAL project [Melquiond and Pion 2007] and in the validation of delayed linear algebra over finite fields [Boldo et al. 2008]. For all these applications, either Gappa

Listing 1. Gappa script proving that e accurately approximates $E_0 = \exp(R_0)$ when the input R is close to R_0

```
# 1.  PROG: Definitions of aliases
# work in single precision
@rnd = float< ieee_32, ne >;

# a few floating-point constants
a1 = 8388676b-24;
a2 = 11184876b-26;
l2 = 12566158b-48;
s1 = 8572288b-23;
s2 = 13833605b-44;

# the algorithm for computing the exponential
r2 rnd= -n * l2;
r rnd= r1 + r2;
q rnd= r * r * (a1 + r * a2);
p rnd= r1 + (r2 + q);
s rnd= s1 + s2;
e rnd= s1 + (s2 + s * p);

# a few mathematical expressions to simplify later sections
R = r1 + r2;
S = s1 + s2;

E = s1 + (s2 + S * (r1 + (r2 + R * R * (a1 + R * a2))));
Er = S * (1 + R + a1 * R * R + a2 * R * R * R + 0);
E0 = S0 * (1 + R0 + a1 * R0 * R0 + a2 * R0 * R0 * R0 + Z);

# 2.  PROP: Logical proposition Gappa has to verify
{ # provide the domains and accuracies of some variables
  Z in [-55b-39,55b-39] /\ S - S0 in [-1b-41,1b-41] /\
  R - R0 in [-1b-34,1b-34] /\ R in [0,0.0217] /\
  n in [-10176,10176] ->
  # ask for the range of e and its absolute error
  e in ? /\ e - E0 in ? }

# 3.  HINTS: Hints provided by the user
e - E0 -> (e - E) + (Er - E0);   # valid since E = Er
r1 -> R - r2;                    # valid since R = r1 + r2
```

carefully bounds the accumulation of individual errors, or it proves that the result is exact since no error ever occurred.

When doing a formal development with the help of Gappa, the whole work of performing the proof is pushed toward it: All the intervals are precomputed and none of the complex tactics of Coq are used. The proof checker only has to be able to add, multiply, and compare integers; it does not have to be able to manipulate

Listing 2. Excerpts of the 2038-line Coq proof produced by Gappa from Listing 1

```
Require Import Gappa_library.
Section Generated_by_Gappa.
Variable _Z : R.
Variable _R0 : R.
...
Notation r4 := ((r6 + _Z)%R).
Variable _S0 : R.
Notation _E0 := ((_S0 * r4)%R).
...
Notation _e := (float2R ((rounding_float roundNE (24) (149)) (r51))).
...
Notation r70 := ((_e - _E0)%R).
...
Definition f49 := Float2 (154166255364809243) (-81).
Definition f50 := Float2 (-75807082762648785) (-80).
Definition i35 := makepairF f50 f49.
Notation p37 := (BND r70 i35).
  (* BND(e - E0, [-6.27061e-08, 6.37617e-08]) *)
...
Lemma l33 : p1 -> p2 -> p34 -> p35 -> p36 -> p37.
  (* BND(e - E0, [-6.27061e-08, 6.37617e-08]) *)
 intros h0 h1 h2 h3 h4.
 apply l34. exact h0. exact h1. exact h2. exact h3. exact h4.
Qed.
End Generated_by_Gappa.
```

rational or real numbers. Consequently, one of our goals is to generate proofs not only for Coq or HOL Light, but for other proof checkers too.

Branches and loops handling are outside the scope of this work. Both problems are not new to program verification and nice results have been published in both areas. We do not want to propose our solution for these problems. Our decision is to interact with the two following tools.

—Why [Filliâtre 2003] is a tool to certify programs written in a generic language (C and Java can be converted to this language). It certifies appropriate memory allocation and usage. It is able to handle hierarchically structured code with functions and assertions. Why also takes care of conditional branches. It duplicates the appropriate proof obligations and guarantees that both pieces of code meet their shared post-conditions. A floating-point formalism designed with Gappa in mind has recently been added to Why [Boldo and Filliâtre 2007]: Whenever Why encounters a proof obligation about floating-point numbers, it submits it to Gappa in order to solve it without any user intervention.

—Fluctuat [Putot et al. 2004] handles loops by effectively computing loop invariants. Once these invariants are provided, Gappa can certify the correct behavior of any numerical code. Results of Fluctuat will be used as oracles and certified by Gappa. Should there be a significant bug in Fluctuat, Gappa will stop without being able to meet its goals as it cannot certify erroneous results.

REFERENCES

Boldo, S. and Daumas, M. 2004. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms 37,* 1-4, 45–60.

Boldo, S., Daumas, M., and Giorgi, P. 2008. Formal proof for delayed finite field arithmetic using floating point operators. In *Real Numbers and Computers*. Santiago de Compostela, Spain.

Boldo, S. and Filliâtre, J.-C. 2007. Formal verification of floating point programs. In *Proceedings of the 18th Symposium on Computer Arithmetic*. Montpellier, France.

Boldo, S. and Melquiond, G. 2008. Emulation of a FMA and correctly rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers 57,* 4, 462–471.

Boutin, S. 1997. Using reflection to build efficient and certified decision procedures. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*. London, United Kingdom, 515–529.

Brönnimann, H., Melquiond, G., and Pion, S. 2006. The design of the Boost interval arithmetic library. *Theoretical Computer Science 351*, 111–118.

Daumas, M. and Melquiond, G. 2004. Generating formally certified bounds on values and round-off errors. In *Real Numbers and Computers*. Dagstuhl, Germany, 55–70.

Daumas, M., Melquiond, G., and Muñoz, C. 2005. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th Symposium on Computer Arithmetic*, P. Montuschi and E. Schwarz, Eds. Cape Cod, Massachusetts, 188–195.

de Dinechin, F., Defour, D., and Lauter, C. 2004. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research report 5137, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France.

de Dinechin, F., Lauter, C. Q., and Melquiond, G. 2006. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon, France, 1318–1322.

Even, G. and Seidel, P.-M. 1999. A comparison of three rounding algorithms for IEEE floating-point multiplication. In *Proceedings of the 14th Symposium on Computer Arithmetic*, I. Koren and P. Kornerup, Eds. Adelaide, Australia, 225–232.

Filliâtre, J.-C. 2003. Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud.

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. 2005. MPFR: A multiple-precision binary floating-point library with correct rounding. Tech. Rep. RR-5753, INRIA.

Gameiro, M. and Manolios, P. 2004. Formally verifying an algorithm based on interval arithmetic for checking transversality. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*. Austin, Texas, 17.

Harrison, J. 1997. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory.

Harrison, J. 2000. *The HOL Light manual*. Version 1.1.

Higham, N. J. 2002. *Accuracy and stability of numerical algorithms*. SIAM. Second edition.

Huet, G., Kahn, G., and Paulin-Mohring, C. 2004. *The Coq proof assistant: a tutorial: version 8.0*.

Jaulin, L., Kieffer, M., Didrit, O., and Walter, E. 2001. *Applied interval analysis*. Springer.

Kahan, W. 1965. Further remarks on reducing truncation errors. *Communications of the ACM 8,* 1, 40.

Kaufmann, M., Manolios, P., and Moore, J. S. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Melquiond, G. and Pion, S. 2007. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications 41,* 1, 57–70.

Michard, R., Tisserand, A., and Veyrat-Charvillon, N. 2006. Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In *Symposium en Architecture de Machines*. Perpignan, France, 1318–1322.

Muñoz, C. and Lester, D. 2005. Real number calculations and theorem proving. In *18th International Conference on Theorem Proving in Higher Order Logics*. Oxford, England, 239–254.

Neumaier, A. 1990. *Interval methods for systems of equations*. Cambridge University Press.

Owre, S., Rushby, J. M., and Shankar, N. 1992. PVS: a prototype verification system. In *11th International Conference on Automated Deduction*, D. Kapur, Ed. Springer-Verlag, Saratoga, New-York, 748–752.

Putot, S., Goubault, E., and Martel, M. 2004. Static analysis based validation of floating point computations. In *Novel Approaches to Verification*. Lecture Notes in Computer Science, vol. 2991. Dagstuhl, Germany, 306–313.

Revy, G. 2006. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Tech. Rep. ensl-00119498, École Normale Supérieure de Lyon.

Rockwell Collins. 2005. Rockwell Collins receives MILS certification from NSA on microprocessor. Press Releases.

Rump, S. M., Ogita, T., and Oishi, S. 2005. Accurate floating-point summation. Tech. Rep. 05.12, Hamburg University of Technology, Hamburg, Germany.

Schlumberger. 2003. Schlumberger leads the way in smart card security with common criteria EAL7 security methodology. Press Releases.

Stevenson, D. et al. 1987. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices 22,* 2, 9–25.

Tang, P. T. P. 1989. Table driven implementation of the exponential function in IEEE floating point arithmetic. *ACM Transactions on Mathematical Software 15,* 2, 144–157.

Tiwari, A., Shankar, N., and Rushby, J. 2003. Invisible formal methods for embedded control systems. *Proceedings of the IEEE 91,* 1, 29–39.