User's Guide for Gappa

Guillaume Melquiond

Contents

Chapter 1

Introduction

Gappa (*Génération Automatique de Preuves de Propriétés Arithmétiques* — automatic proof generation of arithmetic properties) is a tool intended to help verifying and formally proving properties on numerical programs and circuits handling floating-point or fixed-point arithmetic.

This tool manipulates logical formulas stating the enclosures of expressions in some intervals. For example, a formal proof of the property $c \in [-0.3, -0.1] \ \wedge \ (2a \in [3,4] \Rightarrow b + c \in [1,2]) \ \wedge \ a - c \in [1.9, 2.05] \quad \Rightarrow \quad b + 1 \in [2, 3.5]$ can be generated thanks to the following Gappa script.

```
{
  c in [-0.3,-0.1] /\
  (2 * a in [3,4] -> b + c in [1,2]) /\
  a - c in [1.9,2.05]

  -> b + 1 in [2,3.5]
}

a -> a - c + c;
b -> b + c - c;
```

Through the use of rounding operators as part of the expressions, Gappa is specially designed to deal with formulas that could appear when certifying numerical codes. In particular, Gappa makes it simple to bound computational errors due to floating-point arithmetic.

Gappa is free software; you can redistribute it and/or modify it under the terms of the CeCILL Free Software License Agreement or under the terms of the GNU General Public License. (Refer to the files from the source archive for the precise wording of these two licences.) Gappa can be downloaded on its project page[1]. The tool and its documentation have been written by Guillaume Melquiond[2].

[1] http://gappa.gforge.inria.fr/
[2] http://www.lri.fr/~melquion/

Chapter 2

Invoking Gappa

2.1 Input and output

Gappa reads the script whose filename is passed as an argument to the tool, or on its standard input if none. Such a script is made of three parts: the definitions, the logical formula, and the hints. Warning messages, error messages, and results, are displayed on the standard error output. Gappa also sends to the standard output a formal proof of the logical formula; its format depends on the selected back-end. Finally, if the tool was unable to prove some goals, its return code is a nonzero value.

For example, the following command lines create a simple script in file `test.g`, pass it to Gappa, and store the generated Coq proof in file `test.v` file. They also test the return code of Gappa and send the generated proof to Coq so that it is automatically checked. Since the proof checker does not display anything, it means no errors were detected and the result computed by Gappa is correct.

```
$ echo "{ x in [-2,2] -> x * x in ? }" > test.g

$ gappa -Bcoq test.g > test.v

Results for x in [-2, 2]:
x * x in [0, 1b2 {4, 2^(2)}]

$ echo "Return code is $?"
Return code is 0

$ coqc -I path/to/gappalib-coq test.v

$
```

2.2 Command-line options

2.2.1 Selecting a proof back-end

These options are mutually exclusive and cannot be embedded into scripts.

2.2.1.1 Null back-end.

Option: `-Bnull`

Do not enable any back-end. This is the default behavior. This is also the only back-end compatible with the `-Munconstrained` option.

2.2.1.2 Coq back-end

Option: `-Bcoq`

When this back-end is selected, Gappa generates a script that proves the results it displays. This script can be automatically verified by the Coq proof-checker. It can also be reused in bigger formal developments made with Coq.

2.2.1.3 HOL Light back-end

Option: `-Bholl`

Similar to the previous option, but for the HOL Light proof-checker.

2.2.1.4 Coq lambda-term back-end

Option: `-Bcoq-lambda`

This back-end is used by the `gappa` tactic available for the Coq proof checker. It generates a lambda-term that can be directly loaded by the proof checker, but it only supports the features needed by the tactic: no bisection nor rewriting.

2.2.2 Setting internal parameters

2.2.2.1 Internal precision

Option: `-Eprecision=`*integer*

This option sets the internal MPFR precision that Gappa uses when computing bounds of intervals. The default value is 60 and should be sufficient for most uses.

2.2.2.2 Dichotomy depth

Option: `-Edichotomy=`*integer*

This option limits the depth of a dichotomy split. The default value is 100. It means that an interval of width 1 is potentially split into 2^{100} intervals of width 2^{-100} relatively to the original interval. This should be sufficient for most uses too.

2.2.2.3 Fused multiply-add format

Option: `-E[no-]reverted-fma`

By default (`-Eno-reverted-fma`), the expression `fma(a,b,c)` is interpreted as `a * b + c`. As this may not be the preferred order for the operands, the option makes Gappa use `c + a * b` instead.

2.2.3 Setting modes

These options cannot be embedded into scripts.

2.2.3.1 Assuming vague hypotheses

Option: `-Munconstrained`

By default, Gappa checks that all its hypotheses hold before using a theorem. This mode weakens the engine by making it skip hypotheses that are not needed for computing intermediate results. For example, Gappa will no longer check that `x` is not zero before applying the lemma proving `x / x in [1,1]`.

This mode is especially useful when starting a proof on relative errors, as it makes it possible to get some early results about them without having to prove that they are well-defined.

At the end of its run, Gappa displays all the facts that are left unproven. In the following example, the property `NZR` indicates that Gappa possibly need to prove that a value (namely `x - 1`, which appears as a denominator and is equal to zero for `x` equal to 1) is nonzero.

```
{ x in [1,2] ->
  (x + 1) / (x + 1) in ? /\ (x - 1) / (x - 1) in ? }
```

```
Results for x in [1, 2]:
(x + 1) / (x + 1) in [1, 1]
(x - 1) / (x - 1) in [1, 1]
Unproven assumptions:
  NZR(x - 1)
```

While Gappa only displays the properties that are left unproven at the end of its run, it may contain false positive. This is especially true when one of the unproven properties actually relies on the proof of another one. Both will be displayed as unproven, although only the second one needs to be proved.

This mode cannot be used when a proof back-end is selected, as a generated proof would contain holes.

2.2.3.2 Improving generated proofs

Option: `-Mexpensive`

In this mode, Gappa tries to find shorter proofs. Although this mode may considerably slow down the computations, its effectiveness is unfortunately far from being guaranteed.

2.2.3.3 Gathering statistics

Option: `-Mstatistics`

At the end of its computations, Gappa displays some statistics. For example, the second script of Section 5.3 gives:

```
Statistics:
  2121 expressions were considered,
    but then 472 of these got discarded.
  7738 theorems were tried. Among these,
    4370 were successfully instantiated,
    yet 1752 of these were not good enough
    and 16 were only partially better.
```

The first two lines show how many intermediate expressions Gappa has prepared. The first number tells how many have been considered, and the second number tells how many of them were discarded because no theorem could handle them. Once this set of expressions is ready, Gappa tries to find properties by applying theorems. The next statistic how many theorems Gappa tried to apply. Among the theorems for which Gappa could satisfy hypotheses, some gave a usable result, and some others did not (because a better property was already proved at the time the theorem was considered).

2.2.3.4 Displaying propositions as Gappa scripts

Option: `-Msequent`

Gappa applies several transformations to the input proposition if it is too complicated. The left-hand sides of the resulting sub-propositions (conjunction of atoms implying conjunctions and disjunctions of atoms) are then displayed in the results. This mode causes the propositions to be displayed in a syntax close to Gappa's one and with their right-hand side, so that one can work directly on the simpler sub-propositions.

```
{ x in [0,1] \/ x in [2,3] -> not x + 1 in [0,1] }
```

```
Script:
  x in [1b1,3] /\
  x + 1 in [0,1] ->
  1 <= 0
Results:
A contradiction was built from the hypotheses.

Script:
  x in [0,1] /\
  x + 1 in [0,1] ->
  1 <= 0
Results:
Warning: no contradiction was found.
```

Note that `1 <= 0` is only there to complete the scripts, since they happen to not have any right-hand side, which means that Gappa tries to prove false from the hypotheses.

2.2.4 Enabling and disabling warning messages

By default, all the warning messages are enabled. See annex for details on warning messages during parsing and during computations.

2.3 Embedded options

Options setting internal parameters or enabling warning messages can be embedded in a Gappa proof script. It is especially important when the logical property cannot be proved with the default parameters. These options are passed through a special comment syntax: `#@ -Edichotomy=200`.

Chapter 3

Formalizing a problem

3.1 Sections of a Gappa script

A Gappa script contains three parts. The first one is used to give names to some expressions in order to simplify the writing of the next parts. The second one is the logical formula one wants to prove with the tool. The third one gives some hints to Gappa on how to prove it.

The following script shows these three parts for the example of Section 5.1.

```
# some notations for making the script readable
@rnd = float<ieee_32, ne>;
x = rnd(xx);
y rnd= x * (1 - x);
z = x * (1 - x);

# the logical formula that Gappa will try (and succeed) to prove
{ x in [0,1] -> y in [0,0.25] /\ y - z in [-3b-27,3b-27] }

# three hints to help Gappa finding the proof
z -> 0.25 - (x - 0.5) * (x - 0.5);
y $ x;
y - z $ x;
```

3.1.1 Logical formula

This is the fundamental part of the script, it contains the logical formula Gappa is expected to prove. This formula is written between brackets and can contain any implication, disjunction, conjunction of enclosures of mathematical expressions. Enclosures are either bounded ranges or inequalities. Any identifier without definition is assumed to be universally quantified over the set of real numbers.

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

The logical formula is first modified and loosely broken according to the rules of sequent calculus. Each of the new formulas will then be verified by Gappa. Ranges on the right of these sub-formulas can be left unspecified. Gappa will then suggest a range such that the logical formula is verified.

```
(1)   x <= 1 /\ x - 2 in [-2,0] -> x + 1 in [0,2] \/ x + y in ?
(2)   x <= 1 /\ x - 2 in [-2,0] /\ y in [3,4] -> x + y in ?
```

3.1.1.1 Inequalities

Inequalities can be used instead of enclosures. The tool will display them as intervals with an infinite bound.

They can be present on both sides of a sub-formula. On the left, it will be used only if Gappa is already able to compute an enclosure of the expression by another way.

On the right, Gappa will put a reverted copy on the left in order to increase the number of available hypotheses, if the equality was part of a disjunction at top level.

For instance, given the first formula below, Gappa actually tries to prove the second one.

```
(before)  a <= 1 /\ b in [2,3] -> ((c >= 4 \/ d in [5,6]) /\ e in [7,8]) \/ f <= 9
(after)   a <= 1 /\ b in [2,3] /\ f >= 9 -> ((c >= 4 \/ d in [5,6]) /\ e in [7,8]) \/ f <= ↩
    9
```

In particular, when displaying the results, the tool will display the new hypothesis on `f`:

```
Results for a in [-inf, 1] and b in [2, 3] and f in [9, inf]:
```

Inequalities that give an upper bound on an absolute value are treated differently. They are assumed to be full enclosures with a lower bound equal to zero. In other words, `|x| <= 1` is syntactic sugar for `|x| in [0,1]`.

Gappa does not know much about inequalities, as they never appear as hypotheses of its theorems. They can only be used to refine a previously computed enclosure. For instance, Gappa cannot deduce a contradiction from `x * x <= -1` alone. It needs to know either an enclosure of `x` or a positive enclosure of `x * x`.

3.1.1.2 Relative errors

The relative error between an approximate value `x` and a more precise value `y` is expressed by `(x - y) / y`. For instance, when one wants to prove a bound on it, one can write:

```
{ ... -> |(x - y) / y| <= 0x1p-53 }
```

However, this bound can only be obtained if `y` is (proved to be) nonzero. Similarly, if a hypothesis was a bound on a relative error, this hypothesis would be usable by Gappa only if it can prove that its denominator is nonzero.

Gappa provides another way of expressing relative errors which does not depend on the denominator being nonzero. It cannot be used as a subexpression; it can only be used as the left-hand side of an enclosure:

```
{ ... -> x -/ y in [-0x1p-53,0x1p-53] }
```

This enclosure on the relative error represents the following proposition: $\exists \varepsilon \in [-2^{-53}, 2^{-53}],\ x = y \times (1+\varepsilon)$.

When the bounds on a relative error are symmetric, the enclosure can be written as `|x -/ y| <= ...` instead.

3.1.1.3 Equalities

While equalities could be encoded with enclosures,

```
{ ... -> x - y in [0,0] -> ...
```

Gappa also understands the following notation:

```
{ ... -> x = y -> ... }
```

Note that equalities are implicitly directed from left to right. For instance, when looking for a property on `a + x`, Gappa will consider `a + y` (assuming that property `x = y` holds), but not the other way around. This is similar to the handling of rewriting rules.

Whenever possible, equalities should be expressed as definitions rather than inside the logical formulas, as it offers more chances for Gappa to apply its theorems.

3.1.1.4 Expression precision

In addition to equalities and inequalities, Gappa also supports properties about the precision needed to represent expressions. These are expressed by using the predicates `@FIX(x,k)` and `@FLT(x,k)` where `x` is an expression and `k` an integer. Both properties state that `x` can be decomposed as a generic binary floating-point number: $\exists m, e \in Z,\ x = m \cdot 2^e$. For `@FIX` this decomposition satisfies $e \geq k$, while for `@FLT` it satisfies $|m| < 2^k$. For instance, the following Gappa formula holds

```
x = float<ieee_32,ne>(x_);
{ @FIX(x,-149) /\ @FLT(x,24) }
```

3.1.2 Definitions

Typing the whole expressions in the logical formula can soon become annoying and error-prone, especially if they contain rounding operators. To ease this work, the user can define in another section beforehand the expressions that will be used more than once. A special syntax also allows to avoid embedding rounding operators everywhere.

First, rounding operators can be given a shorter name by using the `@name = definition<parameters>` syntax. `name` can then be used wherever the whole definition would have been used.

Next, common sub-terms of a mathematical expression can be shared by giving them a name with the `name = term` syntax. This name can then be used in later definitions or in the logical formula or in the hints. The equality itself does not hold any semantic meaning. Gappa will only use the name as a shorter expression when displaying the sub-term, and in the generated proof instead of a randomly generated name.

Finally, when all the arithmetic operations on the right side of a definition are followed by the same rounding operator, this operator can be put once and for all on the left of the equal symbol. For example, with the following script, Gappa will complain that `y` and `z` are two different names for the same expression.

```
@rnd = float<ieee_32, ne>;
y = rnd(x * rnd(1 - x));
z rnd= x * (1 - x);
{ y - z >= 0 }
```

3.1.3 Hints

Hints for Gappa's engine can be added in this last section, if the tool is not able to prove the logical formula. These hints are either rewriting rules or bisection directives. Approximate and accurate expressions can also be provided in this section in order to generate implicit rewriting rules.

3.2 Preferred expressions and other peculiarities

Gappa rewrites expressions by matching them with well-known patterns. The enclosure of an unmatchable expression will necessarily have to be computed through interval arithmetic. As a consequence, to ensure that the expressions will benefit from as much rewriting as possible, special care needs to be taken when expressing the computational errors.

3.2.1 Absolute and relative errors

Let `exact` be an arithmetic expression and `approx` an approximation of `exact`. The approximation usually contains rounding operators while the exact expression does not. The absolute error between these two expressions is their difference: `approx - exact`.

Writing the errors differently will prevent Gappa from applying theorems on rounding errors: `-(x - rnd(x))` may lead to a worse enclosure than `rnd(x) - x`.

The situation is similar for the relative error. It should be expressed as the quotient between the absolute error and the exact value: `(approx - exact) / exact`. For enclosures of relative errors, this is the same: `approx -/ exact`.

3.2.2 Global errors

The `approx` and `exact` expressions need to have a similar structure in order for the rewriting rules to kick in. E.g., if `exact` is a sum of two terms `a` and `b`, then `approx` has to be the sum of two terms `c` and `d` such that `c` and `d` are approximations of `a` and `b` respectively.

Indeed the rewriting rule for the absolute error of the addition is: `(a + b) - (c + d) -> (a - c) + (b - d)`. Similarly, the rewriting rules for the multiplication keep the same ordering of sub-terms. For example, one of the rules is: `a * b - c * d -> (a - c) * b + c * (b - d)`. Therefore, one should try not to mess with the structure of expressions.

If the two sides of an error expression happen to not have a similar structure, then a user rewriting rule has to be added in order to put them in a suitable state. For instance, let us assume we are interested in the absolute error between `a` and `d`, but they have dissimilar structures. Yet there are two expressions `b` and `c` that are equal and with structures similar to `a` for `b` and `d` for `c`. Then we just have to add the following hints to help Gappa:

```
b - c -> 0; # or "b-c in [0,0]" as a hypothesis
a ~ b; # a is an approximation of b
c ~ d; # c is an approximation of d
```

3.2.3 Discrete values

When an expression is known to take a few separate values that are not equally distributed, a disjunction is the simplest solution but it can be avoided if needed. For instance, if `x` can only be 0, 2, or 17, one can write:

```
x = i * (19b-1 * i - 15b-1);
{ @FIX(i, 0) /\ i in [-1,1] -> ... }
$ i in 3; # this dichotomy hint helps Gappa to understand that i is either -1, 0, or 1
```

3.2.4 Disjunction

When the goal of a formula contains a disjunction, one of the sides of this disjunction has to always hold with respect to the set of hypotheses. This is especially important when performing dichotomies. Even if Gappa is able to prove the formula in each particular branch of a dichotomy, if a different property of the disjunction is used each time, the tool will fail to prove the formula in the general case. Note that, whenever a contradiction is found for a specific set of hypotheses, whichever side of the disjunction holds no longer matter, since Gappa can infer any of them.

3.3 Providing hints

3.3.1 Rewriting rules

Internally, Gappa tries to compute the range of mathematical terms. For example, if the tool has to bound the product of two factors, it will check if it knows the ranges of both factors. If it does, it will apply the theorem about real multiplication in order to compute the range of the product.

Unfortunately, there may be some expressions that Gappa cannot bound tightly. This usually happens because it has no result on a sub-term or because the expression is badly correlated. In this case, the user can provide an intermediate expression with the following hint.

```
primary -> secondary;
```

If Gappa finds a property about the secondary expression, it will use it as if it was about the primary expression. This transformation is valid as long as both expressions are equal. So, when generating a theorem proof, Gappa adds this equality as a hypothesis. It is then up to the user to prove it in order to be able to apply the theorem.

To detect mistyping early, Gappa checks if both expressions can be normalized to the same one according to field rules (and are therefore equal) and warn if they are not. (It will not generate a proof of their equality though.) Note that Gappa does not check if the divisors that appear in the expressions are always nonzero; it just warns about them.

If an equality requires some conditions to be verified, they can be added to the rule:

```
sqrt(x * x) -> -x { x <= 0 };
```

3.3.2 Approximated expressions

As mentioned before, Gappa has an internal database of rewriting rules. Some of these rules need to know about accurate and approximate terms. Without this information, they do not match any expression. For example, Gappa will replace the term `B` by `b + -(b - B)` only it knows a term `b` that is an approximation of the term `B`.

Gappa has two heuristics to detect terms that are approximations of other terms. First, rounded values are approximations of their non-rounded counterparts. Second, any absolute or relative error that appears as a hypothesis of a logical sub-formula will define a pair of accurate and approximate terms.

Since these pairs create lots of already proved rewriting rules, it is helpful for the user to define its own pairs. This can be done with the following syntax.

```
approximate ~ accurate;
```

In the following example, the comments show two hints that could be added if they had not already been guessed by Gappa. In particular, the second one enables a rewriting rule that completes the proof.

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

3.3.3 Dichotomy search

The last kind of hint can be used when Gappa is unable to prove a formula but would be able to prove it if the hypothesis ranges were not so wide. Such a failure is usually caused by a bad correlation between the sub-terms of the expressions. This can be solved by rewriting the expressions. But the failure can also happen when the proof of the formula is not the same everywhere on the domain, as in the following example. In both cases, the user can ask Gappa to split the ranges into tighter ranges and see if it helps proving the formula.

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

With this script, Gappa will answer that $|z| \leq 3 \cdot 2^{-24}$. This is not the best answer because Gappa does not notice that `rnd(y) - y` is always zero when $\frac{1}{2} \leq x \leq 3$. The user needs to ask for a bisection with respect to the expression `rnd(x_)`.

There are three types of bisection. The first one splits an interval in as many equally wide sub-intervals as asked by the user. The second one splits an interval along the points provided by the user.

```
$ x in 6;                # split the range of "x" in six sub-intervals
$ x in (0.5,1.9999999); # split the range of "x" in three sub-intervals, the middle one ↩
    being [0.5,~2]
$ x;                     # equivalent to: $ x in 4;
```

The third kind of bisection tries to find by dichotomy the best range split such that a goal of the logical formula holds true. This requires the range of this goal to be specified, and the enclosed expression has to be indicated on the left of the `$` symbol.

```
{ x in [0,3] -> |z| <= 1b-26 }
|z| $ x;
```

Contrarily to the first two kinds of bisection, this third one keeps the range improvements only if the goal was finally reached. If there was a failure in doing so, all the improvements are discarded. Gappa will display the sub-range on which the goal was not proved. There is a failure when Gappa cannot prove the goal on a range and this range cannot be split into two half ranges, either because its internal precision is not enough anymore or because the maximum depth of dichotomy has been reached. This depth can be set with the `-Edichotomy=999` option.

More than one bisection hint can be used. And hints of the third kind can try to satisfy more than one goal at once.

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

These two hints will be used one after the other. In particular, none will be used when Gappa is doing a bisection with the other one. By adding terms on the right of the `$` symbol, more complex hints can be built. Beware of combinatorial explosions though. The following hint is an example of a complex bisection: it asks Gappa to find a set of sub-ranges on `u` such that the goal on `b` is satisfied when the range on `v` is split into three sub-intervals.

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

Rather than mentioning simple terms on the left-hand side of hints, one can also write a logical formula. As a consequence, Gappa no longer infers the termination condition from the goal but instead performs a bisection until the formula is satisfied. This is especially useful if no enclosures of the terms appear in the goal, or if the termination criteria is not even an expression enclosure.

```
rnd(a) = a $ u
```

Chapter 4

Supported arithmetic

4.1 Rounding directions

Some of the classes of operators presented in the following sections are templated by a rounding direction. This is the direction chosen when converting a real number that cannot be exactly represented in the destination format.

There are eleven directions:

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

The rounding directions mandated by the IEEE-754 standard are `ne` (default mode, rounding to nearest), `zr`, `dn`, `up`, and `na` (introduced for decimal arithmetic).

4.2 Floating-point operators

This class of operators covers all the formats whose number sets are $F(p,d) = \{m \times 2^e; |m| < 2^p, e \geq d\}$. In particular, IEEE-754 floating-point formats (with subnormal numbers) are part of this class, if we set apart overflow issues. Both parameters p and d select a particular format. The last parameter selects the rounding direction.

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

Having to remember the precision and minimum exponent parameters may be a bit tedious, so an alternate syntax is provided: instead of these two parameters, a name can be given to the `float` class.

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

There are four predefined formats:

`ieee_32` IEEE-754 single precision

`ieee_64` IEEE-754 double precision

`ieee_128` IEEE-754 quadruple precision

`x86_80` extended precision on x86-like processors

4.3 Fixed-point operators

This class of operators covers all the formats whose number sets are $F(e) = \{m \times 2^e\}$. The first parameter selects the weight of the least significant bit. The second parameter selects the rounding direction.

```
fixed< lsb_weight, rounding_direction >(...)
```

Rounding to integer is a special case of fixed point rounding of weight 0. A syntactic shortcut is provided.

```
int< rounding_direction >(...)
```

4.4 Miscellaneous operators

The following operators are underspecified and therefore not suitable for formal proofs.

4.4.1 Functions with relative error

This set of functions is defined with related theorems on relative error. They can be used to express properties that cannot be directly expressed through unary rounding operators.

```
{add|sub|mul}_rel< precision [, minimum_exponent] >(..., ...)
```

If the minimum exponent is not provided, the bound on the relative error is assumed to be valid on the entire domain. Otherwise the interval $[-2^e, 2^e]$ is excluded from the domain.

Talking about the expression `add_rel<20,-60>(a,b)` is like talking about a fresh expression `c` such that `not |a + b| <= 1b-60 -> |c -/ (a + b)| <= 1b-20`.

4.4.2 Rounding operators with homogen properties

To be written.

Chapter 5

Examples

5.1 A simple example to start from: x * (1 - x)

5.1.1 The C program

Let us analyze the following function.

```
float f(float x)
{
  assert(0 <= x && x <= 1);
  return x * (1 - x);
}
```

This function computes the value `x * (1 - x)` for an argument `x` between 0 and 1. The float type is meant to force the compiler to use IEEE-754 single precision floating-point numbers. We also assume that the default rounding mode is used: rounding to nearest number, break to even on tie.

The function returns the value $x \otimes (1 \ominus x)$ instead of the ideal value $x \cdot (1 - x)$ due to the limited precision of the computations. If we rule out the overflow possibility (floating-point numbers are limited, not only in precision, but also in range), the returned value is also $\circ(x \cdot \circ(1 - x))$. This o function is a unary operator related to the floating-point format and the rounding mode used for the computations. This is the form Gappa works on.

5.1.2 First Gappa version

We first try to bound the result of the function. Knowing that `x` is in the interval `[0,1]`, what is the enclosing interval of the function result? It can be expressed as an implication: If `x` is in `[0,1]`, then the result is in ... something. Since we do not want to enforce some specific bounds yet, we use a question mark instead of some explicit bounds.

The logical formula Gappa has to verify is enclosed between braces. The rounding operator is a unary function `float< ieee_32, ne >`. The result of this function is a real number that would fit in a IEEE-754 single precision (`ieee_32`) floating-point number (`float`), if there was no overflow. This number is potentially a subnormal number and it was obtained by rounding the argument of the rounding operator toward nearest even (`ne`).

The following Gappa script finds an interval such that the logical formula describing our previous function is true.

```
{ x in [0,1] -> float<ieee_32,ne>(x * float<ieee_32,ne>(1 - x)) in ? }
```

Gappa answers that the result is between 0 and 1. Without any help from the user, they are the best bounds Gappa is able to prove.

```
Results for x in [0, 1]:
[rounding_float,ne,24,149](x * [rounding_float,ne,24,149](1 - x)) in [0, 1]
```

5.1.3 Defining notations

Directly writing the completely expanded logical formula is fine for small formulas, but it may become tedious once the problem gets big enough. For this reason, notations can be defined to avoid repeating the same terms over and over. These notations are all written before the logical formula.

For example, if we want not only the resulting range of the function, but also the absolute error, we need to write the expression twice. So we give it the name `y`.

```
y = float<ieee_32,ne>(x * float<ieee_32,ne>(1 - x));
{ x in [0,1] -> y in ? /\ y - x * (1 - x) in ? }
```

We can simplify the input a bit further by giving a name to the rounding operator too.

```
@rnd = float<ieee_32, ne>;
y = rnd(x * rnd(1 - x));
{ x in [0,1] -> y in ? /\ y - x * (1 - x) in ? }
```

These explicit rounding operators right in the middle of the expressions make it difficult to directly express the initial C code. So we factor the operators by putting them before the equal sign.

```
@rnd = float<ieee_32, ne>;
y rnd= x * (1 - x);
{ x in [0,1] -> y in ? /\ y - x * (1 - x) in ? }
```

Please note that this implicit rounding operator only applies to the results of arithmetic operations. In particular, `a rnd= b` is not equivalent to `a = rnd(b)`. It is equivalent to `a = b`.

Finally, we can also give a name to the infinitely precise result of the function to clearly show that both expressions have a similar arithmetic structure.

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

{ x in [0,1] -> y in ? /\ y - z in ? }
```

On the script above, Gappa displays:

```
Results for x in [0, 1]:
y in [0, 1]
y - z in [-1b-24 {-5.96046e-08, -2^(-24)}, 1b-24 {5.96046e-08, 2^(-24)}]
```

Gappa displays the bounds it has computed. Numbers enclosed in braces are approximations of the numbers on their left. These exact left numbers are written in decimal with a power-of-two exponent. The precise format will be described below.

5.1.4 Complete version

Previously found bounds are not as tight as they could actually be. Let us see how to expand Gappa's search space in order for it to find better bounds. Not only Gappa will be able provide a proof of the optimal bounds for the result of the function, but it will also prove a tight interval on the computational absolute error.

5.1.4.1 Notations

```
x = rnd(xx);                          # x is a floating-point number
y rnd= x * (1 - x);                   # equivalent to y = rnd(x * rnd(1 - x))
z = x * (1 - x);
```

The syntax for notations is simple. The left-hand-side identifier is a name representing the expression on the right-hand side. Using one side or the other in the logical formula is strictly equivalent. Gappa will use the defined identifier when displaying the results and generating the proofs though, in order to improve their readability.

The second and third notations have already been presented. The first one defines `x` as the rounded value of a real number `xx`. In the previous example, we had not expressed this property of `x`: it is a floating-point number. This additional piece of information will help Gappa to improve the bound on the error bound. Without it, a theorem like Sterbenz' lemma cannot apply to the `1 - x` subtraction.

5.1.4.2 Logical formulas and numbers

```
{ x in [0,1] -> y in [0,0.25] /\ y - z in [-3b-27,3b-27] }
```

Numbers and bounds can be written either in the usual scientific decimal notation or by using a power-of-two exponent: `3b-27` means $3 \cdot 2^{-27}$. Numbers can also be written with the C99 hexadecimal notation: `0x0.Cp-25` is another way to express the bound on the absolute error.

5.1.4.3 Hints

Although we have given additional data through the type of `x`, Gappa is not yet able to prove the formula. It needs some user hints.

```
z -> 0.25 - (x - 0.5) * (x - 0.5);      # x * (1 - x) == 1/4 - (x - 1/2)^2
y $ x;                                  # bound y by splitting the interval on x
y - z $ x;                              # bound y - z by splitting ...
```

The first hint indicates to Gappa that both hand sides are equivalent: When it tries to bound the left hand side, it can use the bounds it has found for the right hand side. Please note that this rewriting only applies when Gappa tries to bound the expression, not when it tries to bound a bigger expression.

The two other hints indicate that Gappa should bound the left-hand-side values by doing a case split on the right-hand side. It only works if the logical proposition requires explicit bounds for the expression on the left hand side.

5.1.4.4 Full listing

As a conclusion, here is the full listing of this example.

```
# some notations
@rnd = float<ieee_32, ne>;
x = rnd(xx);                            # x is a floating-point number
y rnd= x * (1 - x);                     # equivalent to y = rnd(x * rnd(1 - x))
z = x * (1 - x);                        # the value we want to approximate

# the logical property
{ x in [0,1] -> y in [0,0.25] /\ y - z in [-3b-27,3b-27] }

# hints
z -> 0.25 - (x - 0.5) * (x - 0.5);      # x * (1 - x) == 1/4 - (x - 1/2)^2
y $ x;                                  # bound y by splitting the interval on x
y - z $ x;                              # bound y - z by splitting ...
```

Gappa gives the results below.

```
Warning: z and 25e-2 - (x - 5e-1) * (x - 5e-1) are not trivially equal.
         Difference: (5e-1)^2 - (25e-2) - 2 * (x) * (5e-1) + (x)

Results for x in [0, 1]:
y in [0, 1b-2 {0.25, 2^(-2)}]
y - z in [-3b-27 {-2.23517e-08, -2^(-25.415)}, 3b-27 {2.23517e-08, 2^(-25.415)}]
```

Note that Gappa checks the rewriting rules in order to provide early detection of mistyping. Since this check does not involve computations on decimal numbers, Gappa warns about the user hint on `z`. This false positive can be avoided by adding the special comment `#@ -Wno-hint-difference` before the offending rule.

5.2 Tang's exponential function

5.2.1 The algorithm

In *Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*, Ping Tak Peter Tang described an implementation of an almost correctly-rounded elementary function in single and double precision. John Harrison later did a complete formal proof in HOL Light of the implementation in *Floating point verification in HOL Light: the exponential function*.

Here we just focus on the tedious computation of the rounding error. We consider neither the argument reduction nor the reconstruction part (trivial anyway, excepted when the end result is subnormal). We want to prove that, in the C code below, the absolute error between `e` and the exponential `E0` of `R0` (the ideal reduced argument) is less than 0.54 ulp. Variable `n` is an integer and all the other variables are single-precision floating-point numbers.

```
r2 = -n * l2;
r = r1 + r2;
q = r * r * (a1 + r * a2);
p = r1 + (r2 + q);
s = s1 + s2;
e = s1 + (s2 + s * p);
```

Let us note `R` the computed reduced argument and `S` the stored value of the ideal constant `S0`. There are 32 such constants. For the sake of simplicity, we only consider the second constant: $2^{\frac{1}{32}}$. `E` is the value of the expression `e` computed with an infinitely precise arithmetic. `Z` is the absolute error between the polynomial $x + a_1 \cdot x^2 + a_2 \cdot x^3$ and $\exp(x) - 1$ for $|x| \leq \frac{\log 2}{64}$.

5.2.2 Gappa description

```
a1 = 8388676b-24;
a2 = 11184876b-26;
l2 = 12566158b-48;
s1 = 8572288b-23;
s2 = 13833605b-44;

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);

R = r1 + r2;
S = s1 + s2;

E0 = S0 * (1 + R0 + a1 * R0 * R0 + a2 * R0 * R0 * R0 + Z);

{ 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]
   ->
  e in ? /\ e - E0 in ? }
```

Gappa is unable to bound the expressions. This is not surprising for `e - E0`, since the tool is missing some of the implicit transformations hidden in the implementation. As for `e` itself, Gappa is missing the range of `r1`. Since Gappa has access to the ranges of `R` and `r2`, this issue can be solved by adding the hint: `r1 -> R - r2`.

Please note the way `Z` is introduced. Its definition is backward: `Z` is a real number such that `E0` is the product of `S0` and the exponential of `R0`. It makes for clearer definitions and it avoids having to deal with divisions.

Now to `e - E0`. We compute this error by splitting it between a pure rounding error `e - E` and another term which combines discretization and truncation errors. This term is `E - E0`. Unfortunately, Gappa is unable to compute tight bounds for this term since the syntactic structures of its sub-terms do not match.

So we introduce a new expression `Er` equal to `E` yet matching the structure of `E0`. Since `Z` has no equivalent term in `E`, we insert an artificial term `0` in the corresponding place in `Er`.

```
E = s1 + (s2 + S * (r1 + (r2 + R * R * (a1 + R * a2))));
Er = S * (1 + R + a1 * R * R + a2 * R * R * R + 0);
...
e - E0 -> (e - E) + (Er - E0);
```

5.2.3 Full listing

```
@rnd = float< ieee_32, ne >;

a1 = 8388676b-24;
a2 = 11184876b-26;
l2 = 12566158b-48;
s1 = 8572288b-23;
s2 = 13833605b-44;

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);

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);

{ 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] ->
  e in ? /\ e - E0 in ? }

e - E0 -> (e - E) + (Er - E0);
r1 -> R - r2;
```

Gappa answers that the error is bounded by 0.535 ulp. This is consistent with the bounds computed by Tang and Harrison.

```
Results for n in [-10176, 10176] and R in [0, 0.0217] and Z in [-1.00044e-10, 1.00044e-10] ↩
     and S - S0 in [-4.54747e-13, 4.54747e-13] and R - R0 in [-5.82077e-11, 5.82077e-11]:
e in [4282253b-22 {1.02097, 2^(0.0299396)}, 8768135b-23 {1.04524, 2^(0.0638374)}]
e - E0 in [-7580708276264878 5b-80 {-6.27061e-08, -2^(-23.9268)}, 154166255364809243b-81  ↩
    {6.37617e-08, 2^(-23.9027)}]
```

5.3 Fixed-point Newton division

5.3.1 The algorithm and its verification

Let us suppose we want to invert a floating-point number on a processor without a floating-point unit. The 24-bit mantissa has to be inverted from a value between 0.5 and 1 to a value between 1 and 2. For the sake of this example, the transformation is performed by Newton's iteration with fixed-point arithmetic.

The mantissa is noted `d` and its exact reciprocal is `R`. Newton's iteration is started with a first approximation `r0` taken from a table containing reciprocals at precision $\pm 2^{-8}$. Two iterations are then performed. The result `r1` of the first iteration is computed on 16-bit words in order to speed up computations. The result `r2` of the second iteration is computed on full 32-bit words. We want to prove that this second result is close enough to the infinitely precise reciprocal `R = 1/d`.

First, we define `R` as the reciprocal, and `d` and `r0` as two fixed-point numbers that are integer multiples of 2^{-24} and 2^{-8} respectively. Moreover, `r0` is an approximation of `R` and `d` is between 0.5 and 1.

```
R = 1 / d;

{ @FIX(d,-24) /\ d in [0.5,1] /\
  @FIX(r0,-8) /\ r0 - R in [-1b-8,1b-8] ->
  ... }
```

Next we have the two iterations. Gappa's representation of fixed-point arithmetic is high-level: the tool is only interested in the weight of the least significant bit. The shifts that occur in an implementation only have an impact on the internal representation of the values, not on the values themselves.

```
r1 fixed<-14,dn>= r0 * (2 - fixed<-16,dn>(d) * r0);
r2 fixed<-30,dn>= r1 * (2 - d * r1);
```

The property we are looking for is a bound on the absolute error between `r2` and `R`.

```
{ ... -> r2 - R in ? }
```

We expect Gappa to prove that `r2` is $R \pm 2^{-24}$. Unfortunately, this is not the case.

```
Results for d in [0.5, 1] and @FIX(d, -24) and @FIX(r0, -8) and r0 - R in [-0.00390625,  ↩
    0.00390625]:
r2 - R in [-1320985b-18 {-5.03916, -2^(2.33318)}, 42305669b-23 {5.04323, 2^(2.33435)}]
```

5.3.2 Adding hints

With the previous script, Gappa computes a range so wide for `r2 - R` that it is useless. This is not surprising: The tool does not know what Newton's iteration is. In particular, Gappa cannot guess that such an iteration has a quadratic convergence. Testing for `r1 - R` instead does not give results any better.

Gappa does not find any useful relation between `r1` and `R`, as the first one is a rounded multiplication while the second one is an exact division. So we have to split the absolute error into two terms: a rounding error we expect Gappa to compute, and the convergence due to Newton's iteration.

```
{ ... ->
  r1 - r0 * (2 - d * r0) in ? /\ r0 * (2 - d * r0) - R in ? }
```

Gappa now gives the answer below. Notice that the range of the rounding error almost matches the precision of the computations.

```
Results for d in [0.5, 1] and @FIX(d, -24) and @FIX(r0, -8) and r0 - R in [-0.00390625,  ↩
    0.00390625]:
r0 * (2 - d * r0) - R in [-131585b-16 {-2.00783, -2^(1.00564)}, 263425b-17 {2.00977,  ↩
    2^(1.00703)}]
r1 - r0 * (2 - d * r0) in [-255b-22 {-6.07967e-05, -2^(-14.0056)}, 201456639b-40  ↩
    {0.000183224, 2^(-12.4141)}]
```

So Gappa was fine with the rounding error, but not with the algorithmic error. We can help Gappa by directly providing an expression of this error. So we add a rule describing the quadratic convergence of Newton's iteration. And since we also have to split `r1 - R`, we perform both operations in one single rewriting rule.

```
r1 - R -> (r1 - r0 * (2 - d * r0)) - (r0 - R) * (r0 - R) * d;
```

A similar rule is then used for `r2 - R`.

5.3.3 Full listing 1

```
R = 1 / d;

r1 fixed<-14,dn>= r0 * (2 - fixed<-16,dn>(d) * r0);
r2 fixed<-30,dn>= r1 * (2 - d * r1);

{ @FIX(d,-24) /\ d in [0.5,1] /\
  @FIX(r0,-8) /\ r0 - R in [-1b-8,1b-8] ->
  r2 - R in ? }

r1 - R -> (r1 - r0 * (2 - d * r0)) - (r0 - R) * (r0 - R) * d;
r2 - R -> (r2 - r1 * (2 - d * r1)) - (r1 - R) * (r1 - R) * d;
```

Gappa answers that $r_2 = R \pm 2^{-24.7}$.

```
Warning: although present in a quotient, the expression (d) may have not been tested for  ↩
    non-zeroness.

Results for d in [0.5, 1] and @FIX(d, -24) and @FIX(r0, -8) and r0 - R in [-0.00390625,  ↩
    0.00390625]:
r2 - R in [-41710608584542209b-80 {-3.45022e-08, -2^(-24.7887)}, 8356605b-52 {1.85554e-09, ↩
     2^(-29.0055)}]
```

5.3.4 Improving the rewriting rules

First of all, there is this warning message about `d` possibly being zero. Indeed, `R` is the reciprocal of `d` and we are using the fact that `R * d = 1`. So the rewriting rules cannot be proved on their own. (But they can be proved in the context of the problem, so there is no correctness issue.) In order to avoid this warning, we can give the precise hypotheses such that the left hand sides of the rewriting rules are equal to their right hand side without any other assumption. This is indicated at the end of the rule.

```
r1 - R -> (r1 - r0 * (2 - d * r0)) - (r0 - R) * (r0 - R) * d   {d <> 0};
```

Second, the rewriting rules are too specialized. For example, in the first one, the occurrences of `r1` could be replaced by any other real number and the rule would still be valid. Let us consider the problem again. We wanted to split `r1 - R` into a rounding error and an algorithmic error. So we could just say that `r1` is an approximation of `r0 * (2 - d * r0)`.

```
r1 ~ r0 * (2 - d * r0);
```

We are left with hinting Gappa at the quadratic convergence of Newton's iterations.

```
r0 * (2 - d * r0) - R -> (r0 - R) * (r0 - R) * -d   { d <> 0};
```

When generating a script for an external proof checker, Gappa will add this rewriting rule as a global hypothesis. For example, when selecting the Coq back-end with the option `-Bcoq`, the output contains the line below.

```
Hypothesis a1 : (_d <> 0)%R -> r9 = r2.
```

In this hypothesis, `_d` is the `d` variable of the example, while `r9` and `r2` are short notations for `r0 * (2 - d * r0) - R` and `(r0 - R) * (r0 - R) * -d` respectively. In order to access the generated proof, the user has to prove this hypothesis, which can be trivially done with Coq's `field` tactic.

5.3.5 Full listing 2

```
R = 1 / d;

r1 fixed<-14,dn>= r0 * (2 - fixed<-16,dn>(d) * r0);
r2 fixed<-30,dn>= r1 * (2 - d * r1);

{ @FIX(d,-24) /\ d in [0.5,1] /\
  @FIX(r0,-8) /\ r0 - R in [-1b-8,1b-8] ->
  r2 - R in ? }

r1 ~ r0 * (2 - d * r0);
r0 * (2 - d * r0) - R -> (r0 - R) * (r0 - R) * -d   { d <> 0 };
r2 ~ r1 * (2 - d * r1);
r1 * (2 - d * r1) - R -> (r1 - R) * (r1 - R) * -d   { d <> 0 };
```

The answer is the same.

```
Results for d in [0.5, 1] and @FIX(d, -24) and @FIX(r0, -8) and r0 - R in [-0.00390625,  ↩
    0.00390625]:
r2 - R in [-41710608584542209b-80 {-3.45022e-08, -2^(-24.7887)}, 8356605b-52 {1.85554e-09, ↩
     2^(-29.0055)}]
```

Another example of a Newton iteration is given in Section 6.1.

Chapter 6

Using Gappa from other tools

6.1 Why and Gappa

The Why[1] software verification platform can be used to translate code annotated with pre- and postconditions into proof obligations suitable for Gappa.

By installing Frama-C[2] first and then Why (in order to build the Jessie plugin), one gets a tool for directly certifying C programs with Gappa.

6.1.1 Example: floating-point square root

The example below demonstrates the usage of these tools. The C file defines a `sqrt` function that computes the square root with a relative accuracy of 2^{-43} for an input `x` between 0.5 and 2.

```
/*@
 requires 0.5 <= x <= 2;
 ensures \abs(\result - 1/\sqrt(x)) <= 0x1p-6 * \abs(1/\sqrt(x));
*/
double sqrt_init(double x);

/*@
 lemma quadratic_newton: \forall real x, t; x > 0 ==>
   \let err = (t - 1 / \sqrt(x)) / (1 / \sqrt(x));
   (0.5 * t * (3 - t * t * x) - 1 / \sqrt(x)) / (1 / \sqrt(x)) ==
   - (1.5 + 0.5 * err) * (err * err);
*/

/*@
 requires 0.5 <= x <= 2;
 ensures \abs(\result - \sqrt(x)) <= 0x1p-43 * \abs(\sqrt(x));
*/
double sqrt(double x)
{
  int i;
  double t, u;
  t = sqrt_init(x);

  /*@ loop pragma UNROLL 4;
    @ loop invariant 0 <= i <= 3; */
```

[1] http://why.lri.fr/
[2] http://frama-c.com/

```
  for (i = 0; i <= 2; ++i) {
    u = 0.5 * t * (3 - t * t * x);
    //@ assert \abs(u - 0.5 * t * (3 - t * t * x)) <= 1;
    /*@ assert \let err = (t - 1 / \sqrt(x)) / (1 / \sqrt(x));
          (0.5 * t * (3 - t * t * x) - 1 / \sqrt(x)) / (1 / \sqrt(x)) ==
          - (1.5 + 0.5 * err) * (err * err); */
    //@ assert \abs(u - 1 / \sqrt(x)) <= 0x1p-10 * \abs(1 / \sqrt(x));
    t = u;
  }

  //@ assert x * (1 / \sqrt(x)) == \sqrt(x);
  return x * t;
}
```

The code starts by calling the `sqrt_init` function. It returns an approximation of $x^{-1/2}$ with a relative accuracy of 2^{-6}. Only the specification of this auxiliary function is given. (Preconditions are introduced by the `requires` keyword, while postconditions are introduced by `ensures`.) Its implementation could use small tables for instance. Note that bounds on relative errors are expressed as $|approx - exact| \leq error \times |exact|$ in this setting.

The `sqrt` function then performs three Newton iterations in order to obtain an improved approximation of the reciprocal square root of `x`. Since Gappa only handles straight-line programs, a pragma annotation instructs Frama-C to completely unroll the loop before passing it to Jessie. Finally, once the reciprocal square root has been computed, it is multiplied by `x` to obtain the square root.

6.1.2 Passing hints through annotations

The `assert` annotations cause Frama-C/Jessie to generate additional proof obligations. These facts are then available to the following proof obligations as hypotheses. In this example, the actual content of the assertions does not matter from a certification point of view, they are only used as a way to pass information to Gappa. Indeed, as explained in Section 5.3, Gappa needs to know about Newton's relation and which expressions are approximations of what. So, if the program were to be directly expressed in Gappa syntax, the three loop assertions would instead have been written as follows.

```
rsqrt = 1 / sqrt(x);
err = (t - rsqrt) / rsqrt;
{ ... }
u ~ 0.5 * t * (3 - t * t * x);
(0.5 * t * (3 - t * t * x) - rsqrt) / rsqrt -> - (1.5 + 0.5 * err) * (err * err);
u ~ rsqrt;
```

When writing these assertions for guiding Gappa, one just as to make sure that they are easily provable; their actual accuracy is not relevant. For instance, if the relative distance between `u` and $1/\sqrt{x}$ had been 2^{-1} instead of 2^{-10}, Gappa would still have succeeded.

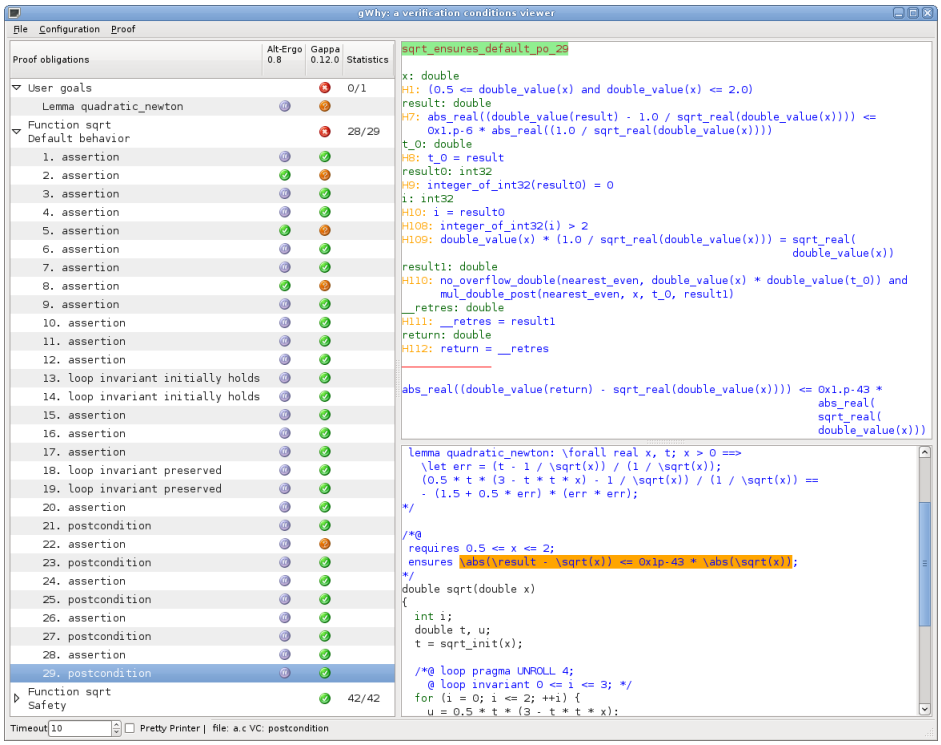
6.1.3 Execution results

Passing the program above to the Frama-C/Jessie tool produces the following console output...

```
$ frama-c -jessie a.c
[kernel] preprocessing with "gcc -C -E -I.  -dD a.c"
[jessie] Starting Jessie translation
[kernel] No code for function sqrt_init, default assigns generated
[jessie] Producing Jessie files in subdir a.jessie
[jessie] File a.jessie/a.jc written.
[jessie] File a.jessie/a.cloc written.
[jessie] Calling Jessie tool in subdir a.jessie
Generating Why function sqrt
[jessie] Calling VCs generator.
```

```
gwhy-bin [...] why/a.why
Computation of VCs...
Computation of VCs done.
Reading GWhy configuration...
Loading .gwhyrc config file
GWhy configuration loaded...
Creating GWhy Tree view...
```

...and displays the following user interface.



On the left of the window are the proof obligations. Once all of them are proved, the code is guaranteed to match its specification. Green marks flag proof obligations that were automatically proved. Selected proof obligations are displayed on the right; here it is the postcondition of the `sqrt` function.

Gappa is not able to prove Newton's relation nor does it know that $x \times \sqrt{x}^{-1} = \sqrt{x}$ holds. These assertions are therefore left unproved. Due to loop unrolling, Newton's relation appears three times. To factor these occurrences, a lemma describing the relation has been added to the C code. The Alt-Ergo[3] prover is used to check that the three occurrences indeed match this lemma.

In the end, we have 72 proof obligations and only two of them are left unproved by the combination of Gappa and Alt-Ergo. They are mathematical identities on real-valued expressions, so they could easily be checked with an interactive proof assistant or a computer algebra system. (And they should be, at least for Newton's relation, because of its error-prone expression.)

6.2 Coq and Gappa

The Gappa Coq Library[4] adds a `gappa` tactic to the Coq Proof Assistant[5]. This tactic invokes Gappa to solve properties about floating-point or fixed-point arithmetic. It can also solve simple inequalities over real numbers.

The tactic is provided by the `Gappa_tactic` module. It expects to find a Gappa executable (called `gappa`) in the user program path.

[3] http://alt-ergo.lri.fr/
[4] http://gappa.gforge.inria.fr/
[5] http://coq.inria.fr/

```
Require Import Reals.
Require Import Fcore.
Require Import Gappa_tactic.
Open Scope R_scope.

Goal
  forall x y : R,
  3/4 <= x <= 3 ->
  0 <= sqrt x <= 1775 * (powerRZ 2 (-10)).
Proof.
  gappa.
Qed.
```

The tactic recognizes fully-applied `rounding_fixed` and `rounding_float` functions as rounding operators.

The script below proves that the difference between two double precision floating-point numbers in $[\frac{52}{16}, \frac{53}{16}]$ and $[\frac{22}{16}, \frac{30}{16}]$ is exactly representable as a double-precision floating-point number. (Rounding direction does not matter for this example; it has been arbitrarily chosen as rounding toward zero.)

```
Definition rnd := rounding_float rndZR 53 (-1074).

Goal
  forall a_ b_ a b : R,
  a = rnd a_ ->
  b = rnd b_ ->
  52 / 16 <= a <= 53 / 16 ->
  22 / 16 <= b <= 30 / 16 ->
  rnd (a - b) = (a - b).
Proof.
  unfold rnd; gappa.
Qed.
```

The tactic handles goals and hypotheses that are either equalities of real numbers, $e_1 = e_2$, or pairs of inequalities on real numbers, $b_1 \leq e \leq b_2$, or inequalities expressing relative errors, $|e_1 - e_2| \leq b \cdot |e_2|$. For inequalities, the b bounds have to be explicit dyadic numbers. The tactic also recognizes properties written as $|e| \leq b$ as syntactic sugar for $0 \leq |e| \leq b$.

The tactic is built on Flocq's formalism[6] and uses the same rounding operators and formats. The previous goal could therefore have been written in a slightly more natural way.

```
Definition format :=
  generic_format radix2 (FLT_exp (-1074) 53).

Goal
  forall a b : R,
  format a -> format b ->
  52 / 16 <= a <= 53 / 16 ->
  22 / 16 <= b <= 30 / 16 ->
  format (a - b).
Proof.
  intros a b Ha Hb Ia Ib.
  refine (sym_eq (_ : rnd (a - b) = a - b)).
  revert Ia Ib.
  replace a with (rnd a).
  replace b with (rnd b).
  unfold rnd ; gappa.
Qed.
```

[6] http://flocq.gforge.inria.fr/

Chapter 7

Customizing Gappa

These sections explain how rounding operators and back-ends are defined in the tool. They are meant for developers rather than users of Gappa and involve manipulating C++ classes defined in the `src/arithmetic` and `src/backends` directories.

7.1 Defining a generator for a new formal system

To be written.

7.2 Defining rounding operators for a new arithmetic

7.2.1 Function classes

A function derives from the `function_class` class. This class is an interface to the name of the function, its associated real operator, and six theorems.

```
struct function_class
{
  function_class(real_op_type t, int mask);
  virtual interval round                          (interval const &, std::string &) const;
  virtual interval enforce                        (interval const &, std::string &) const;
  virtual interval absolute_error                                     (std::string &) const;
  virtual interval relative_error                                     (std::string &) const;
  virtual interval absolute_error_from_exact_bnd (interval const &, std::string &) const;
  virtual interval absolute_error_from_exact_abs (interval const &, std::string &) const;
  virtual interval absolute_error_from_approx_bnd(interval const &, std::string &) const;
  virtual interval absolute_error_from_approx_abs(interval const &, std::string &) const;
  virtual interval relative_error_from_exact_bnd (interval const &, std::string &) const;
  virtual interval relative_error_from_exact_abs (interval const &, std::string &) const;
  virtual interval relative_error_from_approx_bnd(interval const &, std::string &) const;
  virtual interval relative_error_from_approx_abs(interval const &, std::string &) const;
  virtual std::string description() const = 0;
  virtual std::string pretty_name() const = 0;
  virtual ~function_class();
};
```

The `description` function should return the internal name of the rounding operator. It will be used when generating the notations of the proof. When the generated notation cannot be reduced to a simple name, comma-separated additional parameters can be appended. The back-end will take care of formatting the final string. This remark also applies to names

returned by the theorem methods (see below). The `pretty_name` function returns a name that can be used in messages displayed to the user. Ideally, this string can be reused in an input script.

The `real_op_type` value is the associated real operator. This will be `UOP_ID` (the unary identity function) for standard rounding operators. But it can be more complex if needed:

```
enum real_op_type { UOP_ID, UOP_NEG, UOP_ABS, BOP_ADD, BOP_SUB, BOP_MUL, BOP_DIV, ... };
```

The type will indicate to the parser the number of arguments the function requires. For example, if the `BOP_DIAM` type is associated to the function `f`, then `f` will be parsed as a binary function. But the type is also used by the rewriting engines in order to derive default rules for this function. These rules involve the associated real operator (the diamond in this example). $f(a,b)-c\diamond d \quad\longrightarrow\quad (f(a,b)-a\diamond b)+(a\diamond b-c\diamond d)\frac{f(a,b)-c\diamond d}{c\diamond d} \quad\longrightarrow\quad \frac{f(a,b)-a\diamond b}{a\diamond b}+\frac{a\diamond b-c\diamond d}{c\diamond d}+\frac{f(a,b)-a\diamond b}{a\diamond b}\cdot\frac{a\diamond b-c\diamond d}{c\diamond d}$ For these rules and the following theorems to be useful, the expressions $f(a,b)$ and $a\diamond b$ have to be close to each other. Bounding their distance is the purpose of the last ten theorems. The first two theorems compute the range of $f(a,b)$ itself.

It is better for the proof engine not to consider theorems that never return a useful range. The `mask` argument of the `function_class` constructor is a combination of the following flags. They indicate which theorems are known. The corresponding methods should therefore have been overloaded.

```
struct function_class
{
  static const int TH_RND, TH_ENF, TH_ABS, TH_REL,
    TH_ABS_EXA_BND, TH_ABS_EXA_ABS, TH_ABS_APX_BND, TH_ABS_APX_ABS,
    TH_REL_EXA_BND, TH_REL_EXA_ABS, TH_REL_APX_BND, TH_REL_APX_ABS;
};
```

All the virtual methods for theorems have a similar specification. If the result is the undefined interval `interval()`, the theorem does not apply. Otherwise, the last parameter is updated with the name of the theorem that was used for computing the returned interval. The proof generator will then generate an internal node from the two intervals and the name. When defining a new rounding operator, overloading does not have to be comprehensive; some functions may be ignored and the engine will work around the missing theorems.

`round` Given the range of $a\diamond b$, compute the range of $f(a,b)$.

`enforce` Given the range of $f(a,b)$, compute a stricter range of it.

`absolute_error` Given no range, compute the range of $f(a,b)-a\diamond b$.

`relative_error` Given no range, compute the range of $\frac{f(a,b)-a\diamond b}{a\diamond b}$.

`absolute_error_from_exact_bnd` Given the range of $a\diamond b$, compute the range of $f(a,b)-a\diamond b$.

`absolute_error_from_exact_abs` Given the range of $|a\diamond b|$, compute the range of $f(a,b)-a\diamond b$.

`absolute_error_from_approx_bnd` Given the range of $f(a,b)$, compute the range of $f(a,b)-a\diamond b$.

`absolute_error_from_approx_abs` Given the range of $|f(a,b)|$, compute the range of $f(a,b)-a\diamond b$.

`relative_error_from_exact_bnd` Given the range of $a\diamond b$, compute the range of $\frac{f(a,b)-a\diamond b}{a\diamond b}$.

`relative_error_from_exact_abs` Given the range of $|a\diamond b|$, compute the range of $\frac{f(a,b)-a\diamond b}{a\diamond b}$.

`relative_error_from_approx_bnd` Given the range of $f(a,b)$, compute the range of $\frac{f(a,b)-a\diamond b}{a\diamond b}$.

`relative_error_from_approx_abs` Given the range of $|f(a,b)|$, compute the range of $\frac{f(a,b)-a\diamond b}{a\diamond b}$.

The `enforce` theorem is meant to trim the bounds of a range. For example, if this expression is an integer between 1.7 and 3.5, then it is also a real number between 2 and 3. This property is especially useful when doing a dichotomy resolution, since some of the smaller intervals may be reduced to a single exact value through this theorem.

Since the undefined interval is used when a theorem does not apply, it cannot be used by `enforce` to flag an empty interval in case of a contradiction. The method should instead return an interval that does not intersect the initial interval. Due to formal certification considerations, it should however be in the rounded outward version of the initial interval. For example, if the expression is an integer between 1.3 and 1.7, then the method should return an interval contained in [1,1.3[or]1.7,2]. For practical reasons, [1,1] and [2,2] are the most interesting answers.

7.2.2 Function generators

Because functions can be templated by parameters. They have to be generated by the parser on the fly. This is done by invoking the functional method of an object derived from the `function_generator` class. For identical parameters, the same `function_class` object should be returned, which means that they have to be cached.

```
struct function_generator {
  function_generator(const char *);
  virtual function_class const *operator()(function_params const &) const = 0;
  virtual ~function_generator() {}
};
```

The constructor of this class requires the name of the function template, so that it gets registered by the parser. `operator()` is called with a vector of encoded parameters.

If a function has no template parameters, the `default_function_generator` class can be used instead to register it. The first parameter of the constructor is the function name. The second one is the address of the `function_class` object.

```
default_function_generator::default_function_generator(const char *, function_class const  ↩
    *);
```

Chapter 8

Bibliography

8.1 Description of Gappa

[1] Marc Daumas and Guillaume Melquiond, Certification of bounds on expressions involving rounded operators[1], *ACM Transactions on Mathematical Software*, 37(1):1-20, 2010.

[2] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres, *Handbook of Floating-Point Arithmetic*[2], Birkhäuser, 2010.

8.2 Applications of Gappa

[3] Guillaume Melquiond and Sylvain Pion, Formally certified floating-point filters for homogeneous geometric predicates[3], *Theoretical Informatics and Applications*, 41(1):57-70, 2007.

[4] Arnaud Tisserand, High-performance hardware operators for polynomial evaluation[4], *International Journal of High Performance Systems Architecture*, 1(1):14-23, 2007.

[5] Sylvie Boldo, Marc Daumas, and Pascal Giorgi, Formal proof for delayed finite field arithmetic using floating point operators[5], *Proceedings of the 8th Conference on Real Numbers and Computers*, 113-122, Santiago de Compostella, 2008.

[6] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond, Combining Coq and Gappa for certifying floating-point programs[6], *Proceedings of the 16th Calculemus Symposium*, 59-74, Grand Bend, ON, Canada, 2009.

[7] Claude-Pierre Jeannerod and Guillaume Revy, Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores[7], *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, 731-735, Pacific Grove, CA, USA, 2009.

[8] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy, and Gilles Villard, A new binary floating-point division algorithm and its software implementation on the ST231 processor[8], *Proceedings of the 19th Symposium on Computer Arithmetic*, 95-103, Portland, OR, USA, 2009.

[1] http://dx.doi.org/10.1145/1644001.1644003
[2] http://dx.doi.org/10.1007/978-0-8176-4705-6_15
[3] http://dx.doi.org/10.1051/ita:2007005
[4] http://dx.doi.org/10.1504/IJHPSA.2007.013288
[5] http://hal.archives-ouvertes.fr/hal-00135090/
[6] http://dx.doi.org/10.1007/978-3-642-02614-0_10
[7] http://dx.doi.org/10.1109/ACSSC.2009.5469948
[8] http://dx.doi.org/10.1109/ARITH.2009.19

[9] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan, Towards program optimization through automated analysis of numerical precision[9], *Proceedings of the 8th International Symposium on Code Generation and Optimization*, 230-237, Toronto, ON, Canada, 2010.

[10] Vincent Lefèvre, Philippe Théveny, Florent de Dinechin, Claude-Pierre Jeannerod, Christophe Mouilleron, David Pfannholzer, and Nathalie Revol, LEMA: towards a language for reliable arithmetic[10], *ACM SIGSAM Bulletin*, 44(1/2):41-52, 2010.

[11] Sylvie Boldo and Thi Minh Tuyen Nguyen, Hardware-independent proofs of numerical programs[11], *Proceedings of the 2nd NASA Formal Methods Symposium*, 14-23, Washington DC, USA, 2010.

[12] Ali Ayad and Claude Marché, Multi-prover verification of floating-point programs[12], *Proceedings of the 5th International Joint Conference on Automated Reasoning*, 127-141, Edinburgh, Scotland, 2010.

[13] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond, Certifying the floating-point implementation of an elementary function using Gappa[13], *IEEE Transactions on Computers*, 60(2):242-253, 2011.

[9] http://dx.doi.org/10.1145/1772954.1772987
[10] http://dx.doi.org/10.1145/1838599.1838622
[11] http://shemesh.larc.nasa.gov/NFM2010/papers/nfm2010_14_23.pdf
[12] http://dx.doi.org/10.1007/978-3-642-14203-1_11
[13] http://dx.doi.org/10.1109/TC.2010.128

Appendix A

Gappa language

A.1 Comments and embedded options

Comments begin with a sharp sign # and go till the end of the line. Comments beginning by #@ are handled as embedded command-line options. All these comments are no different from a space character.

Space, tabulation, and line-break characters are not significant, they at most act as identifier separators. In the definition part of a script, the GE is never matched, so no separator is needed between operators > and =.

A.2 Tokens and operator priority

There are five composite operators: /\ (AND) and \/ (OR) and -> (IMPL) and <= (LE) and >= (GE). And three keywords: in (IN) and not (NOT) and sqrt (SQRT).

Numbers are either written with a binary representation, or with a decimal representation. In both representations, the {integer} parts are radix-10 natural numbers.

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

These three expressions represent the same number: 57.5e-1, 23b-2, 0x5.Cp0. They do not have the same semantic for Gappa though and a different property will be proved in the decimal case. Indeed, some decimal numbers cannot be expressed as a dyadic number and Gappa will have to harden the proof and add a layer to take this into account. In particular, the user should refrain from being inventive with the constant 1. For example, writing this constant as 00100.000e-2 may prevent some rewriting rules to be applied.

Identifiers (IDENT) are matched by {alpha}({alpha}|{digit}|_)*.

The associativity and priority of the operators in logical formulas is as follows. It is meant to match the usual conventions.

```
%right IMPL
%left OR
%left AND
%left NOT
```

For the mathematical expressions, the priority are as follows. Note that NEG is the priority of the unary minus; this is the operator with the highest precedence.

```
%left '+' '-'
%left '*' '/'
%left NEG
```

A.3 Grammar

```
 0 $accept: BLOB $end

 1 BLOB: BLOB1 HINTS

 2 BLOB1: PROG '{' PROP '}'

 3 PROP: REAL LE SNUMBER
 4     | FIX '(' REAL ',' SINTEGER ')'
 5     | FLT '(' REAL ',' INTEGER ')'
 6     | REAL IN '[' SNUMBER ',' SNUMBER ']'
 7     | REAL IN '?'
 8     | REAL GE SNUMBER
 9     | REAL RDIV REAL IN '[' SNUMBER ',' SNUMBER ']'
10     | '|' REAL RDIV REAL '|' LE NUMBER
11     | REAL RDIV REAL IN '?'
12     | REAL '=' REAL
13     | PROP AND PROP
14     | PROP OR PROP
15     | PROP IMPL PROP
16     | NOT PROP
17     | '(' PROP ')'

18 SNUMBER: NUMBER
19        | '+' NUMBER
20        | '-' NUMBER

21 INTEGER: NUMBER

22 SINTEGER: SNUMBER

23 FUNCTION_PARAM: SINTEGER
24               | IDENT

25 FUNCTION_PARAMS_AUX: FUNCTION_PARAM
26                    | FUNCTION_PARAMS_AUX ',' FUNCTION_PARAM

27 FUNCTION_PARAMS: /* empty */
28                | '<' FUNCTION_PARAMS_AUX '>'

29 FUNCTION: FUNID FUNCTION_PARAMS

30 EQUAL: '='
31      | FUNCTION '='

32 PROG: /* empty */
33     | PROG IDENT EQUAL REAL ';'
34     | PROG '@' IDENT '=' FUNCTION ';'
35     | PROG VARID
36     | PROG FUNID
37     | PROG '@' VARID
38     | PROG '@' FUNID

39 REAL: SNUMBER
40     | VARID
41     | IDENT
42     | FUNCTION '(' REALS ')'
43     | REAL '+' REAL
44     | REAL '-' REAL
```

```
45      | REAL '*' REAL
46      | REAL '/' REAL
47      | '|' REAL '|'
48      | SQRT '(' REAL ')'
49      | FMA '(' REAL ',' REAL ',' REAL ')'
50      | '(' REAL ')'
51      | '+' REAL
52      | '-' REAL

53 REALS: REAL
54       | REALS ',' REAL

55 DPOINTS: SNUMBER
56        | DPOINTS ',' SNUMBER

57 DVAR: REAL
58     | REAL IN INTEGER
59     | REAL IN '(' DPOINTS ')'

60 DVARS: DVAR
61      | DVARS ',' DVAR

62 PRECOND: REAL NE SINTEGER
63        | REAL LE SINTEGER
64        | REAL GE SINTEGER
65        | REAL '<' SINTEGER
66        | REAL '>' SINTEGER

67 PRECONDS_AUX: PRECOND
68             | PRECONDS_AUX ',' PRECOND

69 PRECONDS: /* empty */
70         | '{' PRECONDS_AUX '}'

71 HINTS: /* empty */
72      | HINTS REAL IMPL REAL PRECONDS ';'
73      | HINTS REALS '$' DVARS ';'
74      | HINTS PROP '$' DVARS ';'
75      | HINTS '$' DVARS ';'
76      | HINTS REAL '~' REAL ';'
```

A.4 Logical formulas

These sections describe some properties of the logical fragment Gappa manipulates. Notice that this fragment is sound, as the generated formal proofs depend on the support libraries, and these libraries are formally proved by relying only on the axioms of basic arithmetic on real numbers.

A.4.1 Undecidability

First, notice that the equality of two expressions is equivalent to checking that their difference is bounded by zero: `e - f in [0,0]`. Second, the property that a real number is a natural number can be expressed by the equality between its integer part `int<dn>(e)` and its absolute value `|e|`.

Thanks to classical logic, a first-order formula can be written in prenex normal form. Moreover, by skolemizing the formula, existential quantifiers can be removed (although Gappa does not allow the user to type arbitrary functional operators in order to prevent mistyping existing operators, the engine can handle them).

As a consequence, a first-order formula with Peano arithmetic (addition, multiplication, and equality, on natural numbers) can be expressed in Gappa's formalism. It implies that Gappa's logical fragment is not decidable.

A.4.2 Expressiveness

Equality between two expressions can be expressed as a bound on their difference: `e - f in [0,0]`. For inequalities, the difference can be compared to zero: `e - f >= 0`. The negation of the previous propositions can also be used. Checking the sign of an expression could also be done with bounds; here are two examples: `e - |e| in [0,0]` and `e in [0,1] \/ 1 / e in [0,1]`. Logical negations of these formulas can be used to obtain strict inequalities. They can also be defined by discarding only the zero case: `not e in [0,0]`.

Disclaimer: although these properties can be expressed, it does not mean that Gappa is able to handle them efficiently. Yet, if a proposition is proved to be true by Gappa, then it can be considered to be true even if the previous "features" were used in its expression.

Appendix B

Warning and error messages

B.1 Syntax error messages

B.1.1 Error: foobar at line 17 column 42.

The Bison front-end has detected a syntax error at the given location. The error message is usually unusable, so let us hope the location is enough for you to find what the problem is.

```
f(x) = x + 1;
```

```
Error: syntax error, unexpected '(', expecting FUNID or '=' at line 1 column 2
```

B.1.2 Error: unrecognized option 'bar'.

Gappa was invoked with an unknown option.

Variant: unrecognized option 'bar' at line 42. This error is displayed for options embedded in the script.

B.1.3 Error: the symbol foo is redefined...

A symbol cannot be defined more than once, even if the right hand sides of every definitions are equivalent.

```
a = 1;
a = 1;
```

Nor can it be defined after being used as an unbound variable.

```
b = a * 2;
a = 1;
```

B.1.4 Error: foo is not a rounding operator...

Only rounding operators (unary function close to the identity function) can be prepended to the equal sign in a definition.

```
x add_rel<25,-100>= 1;
```

```
Error: relative,add,25,-100 is not a rounding operator at line 1 column 20
```

B.1.5 Error: invalid parameters for foo...

A function template has been instantiated with an incorrect number of parameters or with parameters of the wrong type.

```
x = float<ieee_32,0>(y);
```

```
Error: invalid parameters for float at line 1 column 20
```

B.1.6 Error: incorrect number of arguments when calling foo...

There are either less or more expressions between parentheses than expected by the function.

```
x = int<zr>(y, z);
```

```
Error: incorrect number of arguments when calling rounding_fixed,zr,0 at line 1 column 17
```

B.2 Error messages

B.2.1 Error: undefined intervals are restricted to conclusions.

You are not allowed to use an interrogation mark for an interval that appears as an hypothesis in the logical formula.

```
{ x in ? -> x + 1 in [0,1] }
```

Notice that this condition is checked after the proposition has been reduced to several elementary propositions. As a consequence, an interrogation mark on the right hand side of an implication can still cause Gappa to fail.

```
{ x + 1 in [0,1] \/ not (x in ?) }
```

B.2.2 Error: the range of foo is an empty interval.

An interval has been interpreted as having reverted bounds.

```
{ x in [2,1] }
```

```
Error: the range of x is an empty interval.
```

Interval bounds are replaced by binary floating-point numbers. As a consequence, Gappa can encounteer an empty interval when it tightens a range appearing in a goal. For example, the empty set is the biggest representable interval that is a subset of the singleton $\{1.3\}$. So Gappa fails to prove the following property.

```
{ 1.3 in [1.3,1.3] }
```

```
Error: the range of 13e-1 is an empty interval.
```

B.2.3 Error: a zero appears as a denominator in a rewriting rule.

Gappa has a detected that a divisor is trivially equal to zero in an expression that appears in a rewriting rule. This is most certainly an error.

```
{ 1 in ? }
y -> y * (x - x) / (x - x);
```

B.3 Warning messages

B.3.1 Warning: the hypotheses on foo are trivially contradictory, skipping.

If an expression is enclosed in several hypotheses, and if the intersection of their ranges is empty, then the proposition is trivially true. As a consequence, Gappa skips the proposition instead of proving it.

```
{ x <= 2 /\ x >= 3 -> x in ? }
```

```
Warning: the hypotheses on x are trivially contradictory, skipping.
```

B.3.2 Warning: foo is being renamed to bar.

When a definition `toto = expr` is given to Gappa, the name `toto` is associated to the expression `expr`. This name is then used whenever Gappa needs to output `expr`. If another definition `titi = expr` is later provided, the new name supersedes the name given previously.

```
a = 42;
b = 42;
{ a - b in ? }
```

```
Warning: a is being renamed to b at line 2 column 7

Results:
b - b in [0, 0]
```

B.3.3 Warning: although present in a quotient, the expression foo may not have been tested for non-zeroness.

When Gappa verifies that both sides of a user rewriting rule are equivalent, it does not generate additional constraints to verify that denominators appearing in the expressions are not zero. For example, the rule `1 / (1 / x) -> x` only applies when `x` is not zero; yet Gappa does not test for it.

No warning messages are displayed when constraints are added to the rewriting rule (whether these constraints are sufficient or not). E.g. `1 / (1/ x) -> x { x <> 0 }`.

Option switch: `-W[no-]null-denominator`.

B.3.4 Warning: foo and bar are not trivially equal.

When Gappa verifies the rule `toto -> titi`, it first performs the subtraction of both sides. Then it normalizes this expression in the field of polynomial fractions with integer coefficients. If this result is not zero, Gappa warns about it.

As the normalization only deals with polynomials and integers, false positive may appear when the expressions involve numerical values or square roots or absolute values.

```
{ 1 in ? }
x * (y - 2) -> x * y - x;  # not equal
1b-2 -> 0.2 + 0.05;        # equal, but needs numerical computations
sqrt(x * x) -> |x|;        # equal, but involves square root and absolute value
```

```
Warning: x * (y - 2) and x * y - x are not trivially equal.
         Difference: -(x)
Warning: 1b-2 and 2e-1 + 5e-2 are not trivially equal.
         Difference: -(2e-1) - (5e-2) + (1b-2)
Warning: sqrt(x * x) and |x| are not trivially equal.
         Difference: (sqrt(x * x)) - (|x|)
```

Option switch: `-W[no-]hint-difference`.

B.3.5 Warning: bar is a variable without definition, yet it is unbound.

Neither an expression is associated to the identifier `titi` nor it is a sub-term of any of the bounded expressions of the logical property. This may mean an identifier was in fact mistyped.

```
{ x - x in ? }
```

```
Warning: x is a variable without definition, yet it is unbound.
```

Option switch: `-W[no-]unbound-variable`.

B.4 Warning messages during proof computation

B.4.1 Warning: no path was found for foo.

The expression `toto` appears in one of the goals, yet Gappa does not have any theorem that could be used to compute this expression or one of its sub-terms.

B.4.2 Warning: hypotheses are in contradiction, any result is true.

Gappa has found a contradiction among the hypotheses. So, instead of proving the goals, it will just prove `False`, since it implies any possible goal.

```
{ x in [1,2] /\ x - 1 in [3,4] -> x in [5,6] }
```

```
Results for x in [1, 2] and x - 1 in [3, 4]:
Warning: hypotheses are in contradiction, any result is true.
```

B.4.3 Warning: no contradiction was found.

Gappa had no specific goal to prove, and as such was expecting to find a contradiction and prove it. Yet none were found.

```
{ x in [1,2] -> not x + 1 in [2,3] }
```

```
Results for x in [1, 2] and x + 1 in [2, 3]:
Warning: no contradiction was found.
```

B.4.4 Warning: some enclosures were not satisfied.

Only part of a conjunction of goals was proved to be true. Gappa was unable to prove some other expressions or formulas, which are displayed after the message.

```
{ x in [1,2] -> x + 1 in ? /\ x + 2 in [2,3] }
```

```
Results for x in [1, 2]:
x + 1 in [1b1 {2, 2^(1)}, 3]
Warning: some enclosures were not satisfied.
Missing x + 2
```

B.4.5 Warning: when foo is in i1, bar is in i2 potentially outside of i3.

When Gappa applies a case splitting `titi $ toto`, it splits the interval of `toto` until the goal containing `titi` holds for any sub-interval. If the maximal dichotomy depth has been reached yet the property still does not hold, the warning message displays the failing sub-interval (the leftmost one) and the computed ranges.

```
#@-Edichotomy=3
{ x in [1,2] -> x + 1 in [2,2.75] }
x + 1 $ x;
```

```
Results for x in [1, 2]:
Warning: when x is in [7b-2 {1.75, 2^(0.807355)}, 15b-3 {1.875, 2^(0.906891)}], x + 1 is ↩
    in [11b-2 {2.75, 2^(1.45943)}, 23b-3 {2.875, 2^(1.52356)}] potentially outside of [1b1 ↩
    {2, 2^(1)}, 11b-2 {2.75, 2^(1.45943)}].
Warning: some enclosures were not satisfied.
Missing x + 1
```

Variant: when foo is in i1, bar is not computable. This warning is displayed if no bounds on `titi` can even be computed.

```
{ x in [-2,1] -> x / x in [1,1] }
x / x $ x;
```

```
Results for x in [-2, 1]:
Warning: when x is in [-1b-99 {-1.57772e-30, -2^(-99)}, 1b-100 {7.88861e-31, 2^(-100)}], x ↩
     / x is not computable.
Warning: some enclosures were not satisfied.
Missing x / x
```

Option switch: `-W[no-]dichotomy-failure`.

B.4.6 Warning: case split on foo has not produced any interesting new result.

This warning is displayed when Gappa is successful in finding a case split that satisfies the goals, yet the results obtained on the sub-intervals are not interesting: they have already been proved through a simpler analysis.

```
{ x in [1,2] -> x + 1 in ? }
$ x;
```

```
Results for x in [1, 2]:
Warning: case split on x has not produced any interesting new result.
x + 1 in [1b1 {2, 2^(1)}, 3]
```

Option switch: `-W[no-]dichotomy-failure`.

B.4.7 Warning: case split on foo has no range to split.

This warning is displayed when Gappa is unable to find the initial range on which to split cases.

```
{ x in [-1,1] -> 1 in ? }
$ 1 / x;
```

```
Results for x in [-1, 1]:
Warning: case split on 1 / x has no range to split.
1 in [1, 1]
```

Option switch: `-W[no-]dichotomy-failure`.

B.4.8 Warning: case split on foo is not goal-driven anymore.

This warning is displayed when Gappa is supposed to find the best case split for proving a property, yet it does not know the range for this property or it has already proved it.

```
{ x in [-1,1] -> x + 1 in ? }
x + 1 $ x;
```

```
Results for x in [-1, 1]:
Warning: case split on x is not goal-driven anymore.
x + 1 in [0, 1b1 {2, 2^(1)}]
```

Option switch: `-W[no-]dichotomy-failure`.

Appendix C

Changes

- Version 0.15.1
  - fact database:
    - fixed broken simplification of a*b -/ c*b
- Version 0.15.0
  - fact database:
    - added EQL predicate: e1 = e2
    - changed user rewriting to use the EQL predicate
    - improved rewriting rules to handle ABS, FIX, FLT, NZR, REL in addition to BND predicate
  - syntax:
    - added "@FLT(e,k)" and "@FIX(e,k)" for inputting FLT and FIX properties
    - added "e1 = e2" for inputting EQL properties
    - allowed arbitrary logical propositions as termination condition for bisection
- Version 0.14.1
  - build system
    - fixed some platform-specific issues
- Version 0.14.0
  - Coq back-end
    - added support for Coq support library 0.14
- Version 0.13.0
  - Coq lambda back-end
    - simplified generated proofs
  - proof graph
    - disabled sequent generation
    - disabled proof tracking for the null back-end
    - improved handling of deep logic negations
    - handled disjunctions by dichotomies (null back-end only)
  - main interface
    - removed option -Monly-failure since there is only one proposition

  - documentation
    * switched from jade to dblatex
- Version 0.12.3
  - Coq lambda back-end
    * fixed incorrect invocation of some theorems
  - arithmetic
    * fixed incorrect proofs for floating-point error near powers of two
- Version 0.12.2
  - back-ends
    * fixed output of underspecified REL goals
- Version 0.12.1
  - proof graph
    * fixed sequents with empty goals not being recognized as proved
  - main interface
    * added option -Msequent to display goals as Gappa scripts
    * added option -Monly-failure to limit output to failing goals
    * improved display of extremely small/big bounds
    * improved display of rounding operators
  - proof paths
    * fixed inequalities (lower bound) on absolute values being ignored
  - arithmetic
    * improved relative operators handling when exponents are not constrained
- Version 0.12.0
  - back-ends
    * added back-end for producing Coq lambda terms
  - proof graph
    * fixed handling of complicated goals
  - main interface
    * added output of failed subgoals
- Version 0.11.3
  - Coq back-end
    * applied correct theorems for intervals with infinite bounds
- Version 0.11.2
  - parser
    * fixed handling of CRLF end of line
- Version 0.11.1
  - main interface
    * removed error code on options --help and --version
- Version 0.11.0

  - proof graph
    * avoided splitting provably-singleton intervals
    * added score system for favoring successful schemes
  - arithmetic
    * tightened rounding error when applied to short values
  - syntax
    * recognized lhs of user rewriting as potential user approximate
    * added "x -/ y in ..." and "|x -/ y| <= ..." for REL properties
  - build system
    * fixed compilation on Cygwin

- Version 0.10.0
  - proof graph
    * avoided infinite dichotomy on some unprovable propositions
  - back-ends
    * fixed generation of subset facts
  - fact database
    * reduced cycles in theorems
  - main interface
    * added -Mschemes option for generating .dot scheme graphs
    * allowed input filename on command line

- Version 0.9.0
  - syntax
    * added constraints on user rewriting, e.g. "x -> 1/(1/x) { x <> 0 }"
  - whole engine
    * added detection of assumed facts in -Munconstrained mode
  - fact database
    * added relative error propagation through division
  - back-ends
    * fixed cache collisions between theorems
  - proof graph
    * fixed intersection of relative errors
    * enabled bound simplification through rewriting rules
    * fixed handling of half-bounded goals

- Version 0.8.0
  - HOL Light back-end
    * added new back-end
  - proof graph
    * added option -Mexpensive to select best paths on length
  - fact database
    * added predicate REL: x = y * (1 + e)
    * replaced rewriting rules on relative error by computations on REL
    * enhanced path finder to fill holes in theorems

    * put back rewriting rules to theorem level
    * fixed incorrect equality of variables
    * added predicate NZR: x <> 0
    * added propagation of FIX and FLT through rounding operators

- Version 0.7.3
  - fact database
    * generalized rounding theorems to any combination of errors and predicates
  - whole engine
    * removed dependencies between sequents
  - parser
    * removed automatic rounding of negated expressions
    * equated numbers with exponent zero but different radices
    * fixed grammar for multiple splits
  - proof graph
    * improved quality of fixed split
- Version 0.7.2
  - parser
    * fixed incorrectly rounded intervals on input
  - fact database
    * restricted domain of some rewriting rules
- Version 0.7.1
  - fact database
    * added error propagation through opposite, division, and square root
  - Coq back-end
    * fixed confusion between nodes from different proof graphs
    * optimized away trivial goal nodes
- Version 0.7.0
  - Coq back-end
    * updated to support Gappa library version 0.1
  - proof graph
    * added optimization pass for bound precision
  - whole engine
    * simplified back-end handling
- Version 0.6.5
  - Coq back-end
    * fixed square root generation
  - fact database
    * disabled square root of negative numbers
  - syntax
    * added fma(a,b,c) as a synonym for a*b+c
  - arithmetic

    * added fma_rel
  - main interface
    * added -Ereverted-fma option to change the meaning of fma(a,b,c) to c+a*b
- Version 0.6.4
  - arithmetic
    * fixed influence zone again for floating-point absolute error
- Version 0.6.3
  - proof graph
    * fixed failure when an inequality leads to a contradiction
    * added support for intersecting ABS predicates
  - hint parser
    * added detection of useless hints
  - parser
    * normalized numbers with fractional part ending by zero
- Version 0.6.2
  - fact database
    * fixed dependencies of rewriting rules relying on non-zero values; a race could lead to failed theorems
    * added formulas to compute the relative error of a sum
  - arithmetic
    * added new directed rounding: to odd, away from zero
    * added new rounding to nearest with tie breaking: to odd, to zero, away from zero, up, down
    * fixed influence zone for floating-point absolute error
- Version 0.6.1
  - proof paths
    * improved detection of dead paths
  - fact database
    * fixed patterns for generalized rounding operators
    * improved rewriting rules for approx/accurate pairs
    * renamed rewriting rules
- Version 0.6.0
  - syntax
    * added ways of specifying how interval splitting is done
    * added detection of badly-formed intervals in propositions
    * removed limitation on multiple hypotheses about the same expression
    * improved heuristic for detecting approx/accurate pairs
    * removed limitation on number of accurate expressions for a given approximate
    * removed hack for accurate expressions of rounded expressions (potentially disruptive)
  - whole engine
    * added inequalities; as hypotheses, they cannot imply theorems but they can restrict computed ranges
  - proof paths
    * put rewriting schemes to a higher level to remove constraints on approx/accurate pairs

    * added rewriting rules for replacing an accurate expression by an approximate and an error

- Version 0.5.6
  - fact database
    * cleaned theorems and removed redundant ones
  - arithmetic
    * enabled test to zero with relative rounding, it can still be disabled by -Munconstrained
  - Coq back-end
    * improved script generation
  - proof graph
    * fixed premature halt when a case split was a failure
    * fixed case split not noticing newly discovered bounds
  - main interface
    * simplified display of hypotheses and sorted display of goals
- Version 0.5.5
  - whole engine
    * added square root (no rule checking though)
    * modified rewriting rules to apply to approximates instead of just rounded values
    * added ABS predicate to workaround abuse of absolute values in theorems
  - syntax
    * added syntax to define user approximates
  - fact database
    * added option to disable constraint checking around zero
  - arithmetic
    * generalized fixed-point range enforcing to any expression
  - proof paths
    * enhanced the detection of dead paths that contain cycles
- Version 0.5.4
  - syntax
    * reduced the number of false-positive for unbound identifiers
    * merged variables and functions anew in order to correctly detect define-after-uses errors
  - proof graph
    * added fifo processing of proof schemes
    * handled case splitting on empty goals
    * added more general schemes for case splitting
  - hint parser
    * shut up warning messages about trivial integers as denominator
- Version 0.5.3
  - proof graph
    * fixed goal removal: undefined goals require "optimal" solutions
- Version 0.5.2
  - proof graph

    * simplified node memory handling
    * simplified graph handling
    * put dichotomy at a higher level, outside of proof schemes
    * replaced hypothesis property vectors by bit vectors, stored in-place if possible
  - syntax
    * added detection of unbound identifiers
  - whole engine
    * added support for complex logical properties
- Version 0.5.1
  - whole engine
    * added FIX and FLT predicates in addition to the original BND predicate
- Version 0.5.0
  - whole engine
    * generalized rounding modes as functions
    * generalized functions with rounding theorems
    * removed default rounding theorems
  - syntax
    * split variables and functions
    * simplified rounding and function syntax: purely C++-template-like syntax
    * added NOT and OR logical operators
  - arithmetic
    * replaced relative rounding by functions
    * factored number rounding handling
    * added fixed-point arithmetic
    * generalized floating-point rounding modes to any triplet (precision, subnormal smallest exponent, direction)
  - proof graph
    * reduced the number of node types by demoting theorem and axiom nodes to internal facts
- Version 0.4.12
  - syntax
    * added a way to define new names for rounding operators
    * simplified the handling of negative numbers
  - Coq back-end
    * fixed representation of relative rounding
- Version 0.4.11
  - proof graph
    * fixed a missing dependency cleanup for an owned union node
  - main interface
    * added parsing of embedded options
- Version 0.4.10
  - proof graph
    * changed the node hypotheses to be graph hypotheses

  - fact database
    * switched some other facts to the absolute value of the denominators
    * added an explicit exclusion pattern for the rewriting rules (e.g. "x * x" is no more excluded)
  - main interface
    * added basic command-line parsing for warnings and parameters
- Version 0.4.9
  - numbers and arithmetic
    * separated number handling from special arithmetic
    * added "relative", a format for handling varying precision rounding
  - formulas
    * implemented absolute value
  - fact database
    * made relative error depends on absolute value of the range
  - proof graph
    * fixed a bug related to the clean-up of the last graph of a dichotomy
    * strengthened the role of modus nodes
- Version 0.4.8
  - fact database
    * tightened intervals for "a + b + a * b"
    * discarded multiple occurrences of the same term in the rewriting rules
    * cleaned up rewriting rules
- Version 0.4.7
  - hint parser
    * sped up verification
- Version 0.4.6
  - floating-point numbers
    * disabled relative error for subnormal numbers (potentially disruptive)
  - homogen numbers
    * cleaned up
    * added non-homogen double rounded format
  - hint parser
    * added pseudo-support for quotient in hint
  - parser
    * added support for C99 hexadecimal floating-point format
  - proof graph
    * replaced useless empty intersections by contradiction proofs
- Version 0.4.5
  - proof graph
    * reworked modus ponens creation to fix an assertion failure in lemma invocation
  - fact database
    * added conditional rules (potentially disruptive)
  - homogen numbers
    * split roundings between initialization and computation
  - Coq back-end
    * implemented union