Verifying (In-)Stability in Floating-point Programs by Increasing Precision, using SMT Solving

Gabriele Paganelli Wolfgang Ahrendt
Chalmers University of Technology Gothenburg, Sweden {gabpag,ahrendt}@chalmers.se

Abstract—When computing with floating-point numbers, programmers choose a certain floating-point precision (like, for instance, float or double) upfront, for each variable. However, whether the chosen precision is appropriate for the computation at hand, and vice versa, is difficult to judge. One way is to increase the precision, and observe whether the result of the computation changes too much, in which case the computation with the original precisions is considered ‘unstable’. This effect may be exhibited with certain inputs, and not with others. With a classical testing approach, inputs that show instability can be very difficult to find. Moreover, testing can only show instability, not stability. In this paper, we present an approach, and its implementation, which can formally prove that an increased precision causes only a limited (quantified) change of the result. Alternatively, if the computation is not stable, the method returns inputs that exhibit this. We use methods from program verification, connecting to a novel SMT (satisfiability modulo theories) solver for floating-point number constraints. The user augments the program P with assertions on the expected stability bound. The system then creates a new program P', a certain kind of merge of P with a higher precision copy of P, computes the weakest precondition of P' w.r.t. these assertions, and feeds the resulting formula to the SMT solver, which then proves stability or alternatively returns data for a test exhibiting instability, to be used for further analysis. The implementation of the system targets a toy language but supports the IEEE standard in a realistic manner. The paper describes the method and its implementation, reports experiments, and discusses the results.

I. INTRODUCTION

“As the population of computer programmers has grown, proficiency in rounding-error analysis has dwindled. To compensate, better diagnostic aids should be incorporated into hardware, into program development environments, and into programming languages; but this is not happening.”

William Kahan [1], primary architect of the IEEE floating-point standards

Floating-point units have been spreading in those technology fields, such as digital signal processing, which were believed to be strongholds for fixed-point — if not just integer — computer arithmetics. Safety-critical (e.g., medical, automotive, avionics) software is running increasingly on general purpose hardware, which means that such software can, and does, make use of floating-point data types. Apart from strictly safety-critical applications, the ever more ubiquitous nature of software dramatically increases the society’s and the individual’s dependency on its reliability. In spite of this, many such applications make use of floating-point computations without programmers being fully aware of what they are doing. Even less spread are skills in testing, analysing, and debugging floating-point computations.

The work we present in this paper aims at making a contribution towards better diagnostic aids for rounding-error analysis. The recent advent of SMT (satisfiability modulo theories) solvers for floating-point number constraints, like Z31 [2], even if yet in an early stage, has great potential for the automated generation of test inputs that make a floating point computation ‘fail’. In the context of this paper, we are focusing on failures of a particular kind: unacceptably large roundoffs.

In particular, our method targets one of Kahan’s ‘five plausible schemes’ [1], which programmers can use to assess the effects of roundoff without the help of a human numerically trained expert scrutinising the code. The particular scheme works by repeating a computation in arithmetics of increased precision format. If this changes the computed result all too much, the accumulated rounding in the original precision was too high, which makes that precision inadequate for the computation at hand. (Further analysis might show that the precision would be adequate if only the computation was changed.) Using more digits of precision mitigates the effects of round-off in floating-point operations [3]. According to [1], the scheme is not a silver bullet but can often help the numerically unskilled programmer to assess the stability of an algorithm with respect to the size of the floating point type she is using. It does not replace the intervention of a numerical analyst. A computation $A$ is stable with respect to a tolerance $r$ if increasing the precision does not change the result more than $r$ allows. Stability is an approximation of adequacy, which measures how close $A$ on finite precision data is to $A$ on real numbers. Beware: proving stability does not prove accuracy because the computation on higher precision might not be accurate, but detecting instability does indicate inaccuracy thus the need of a deeper analysis. The biggest problem in applying the schema lies in finding a test input that exposes stability problems of the algorithm under analysis. To solve such problems, this work presents a formal verification method inspired by the Increase Precision assessment scheme. The method gives the user, depending on the outcome: a) counterexamples in form of failing test cases, b) a proof that an algorithm is accurate (with respect to a higher precision). It has the following relevant features: 1) all inputs allowed by the program are considered, thus removing the problem of finding test inputs; 2) it uses a weakest precondition calculus that produces verification conditions encoding the program properties; 3) it uses an SMT solver that has pure floating-point support to check the validity of such verification conditions, hence the corresponding properties; 4) it does not make any reference to mathematical reals: all computations and SMT reasoning are performed on boolean, machine integers, floating-point entities and nothing else.

The stability of an algorithm $A$ refers to the effects of the computational error on $A$ ’s result. $A$ is stable if the result it produces is relatively insensitive to perturbations made during the computation [4]. Stability can be measured in terms of units

\footnote{Available at http://z3.codeplex.com/}
in the last place (ulp), or in terms of relative error (which is proportional to ulps [5]).

The method targets a toy language (FPHILE) with specification support in the form of assertion and assumption statements. The programming language has an IEEE floating-point support similar as Java, however with configurable precision.

This paper is structured as follows: Section II describes the workflow; Section III describes the FPHILE language and Section IV its annotations for verification support; Section V illustrates the program transformation at the core of the presented method; Section VI describes the experiments made with FPHILE; Section VII discusses the results and compares the approach with random testing; Section VIII describes related work and conclusions.

II. APPROACH AND WORKFLOW

The FPHILE verification system supports the following workflow:

1) Given an FPHILE program $P$, the user expresses the properties she is interested in directly in the program, using assertions.
2) Settings, including the low and the high precision used in the analysis, are configured via a file or a simple GUI.
3) FPHILE processes the program under analysis by creating a program $M$ that executes two versions of $P$ interleaved, differing in the precision of their floating-point type. The analysis continues on $M$.
4) FPHILE computes a weakest precondition of $M$ w.r.t. the specification.
5) FPHILE negates the weakest precondition, and uses an SMT solver to find a satisfying assignment for it.
6) If the formula is unsatisfiable, the program fulfills the specification.
7) Otherwise, the satisfying assignment is used to generate a test case guaranteed to fail.
8) The user can iterate the process after improving the specification, the program, or both.

A weakest precondition [6] is a predicate $wp(P, Q)$ that encodes the necessary assumptions needed for a program $P$ to satisfy $Q$ after termination. In FPHILE the calculation of the weakest precondition follows the one illustrated in the B-Book [7]. FPHILE encodes the weakest precondition of $P$ into a SMT-LIB language [8] script. FPHILE then uses the SMT solver as a black box. If the proof attempt returns a satisfying assignment, FPHILE can extract a test case or a specialized program. FPHILE also offers facilities to translate FPHILE programs to Java in order to execute the test cases. The test case can be used to test and debug the program.

III. TARGET LANGUAGE

The FPHILE language is a sequential imperative language. It features 1) the types $bool$, $int$, $float$ (plus $hifloat$, only to be used in the analysis) 2) the statements in Table I 3) the built-in functions in Table II. It is not possible to define more functions or types. Numeric types can be parametric in their precision.

**Parametric floating-point types:** As opposed to languages such as Java or C, the precision of the floating-point types in FPHILE is defined by the user in a configuration file, to decouple this aspect from programs. A precision is a pair of integers $e, m > 0$ indicating the number of bits allocated respectively for the exponent and mantissa of an IEEE-compliant binary floating-point system. Two precisions $\eta = (e, m)$ and $\rho = (e', m')$ are such that $\rho \geq \eta$ if and only if $e' \geq e$ and $m' \geq m$. Given $\rho \geq \eta$, a program $P$, a float or hifloat expression $e$, $P_\rho^\eta$ indicates a program whose float and hifloat types have respectively precisions $\eta$ and $\rho$; $e_\rho$ specifies that its type has precision $\eta$. Subscripts or superscript are omitted when not relevant or clear from the context.

**Floating-point details:** The semantics of floating-point types is similar to the one of the Java programming language. There is no access to the status flags; there is no difference between signalling and normal NaN. FPHILE programs behave as a Java class annotated with the strictfp modifier: no ex-

<table>
<thead>
<tr>
<th>Return type</th>
<th>Name(s)</th>
<th>Arguments' types</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>sqrt</td>
<td>( float a )</td>
</tr>
<tr>
<td>float</td>
<td>roundToInteger</td>
<td>( float a )</td>
</tr>
<tr>
<td>float</td>
<td>abs</td>
<td>( float a )</td>
</tr>
</tbody>
</table>

Table I: Program statements in FPHILE.

<table>
<thead>
<tr>
<th>type identifier;</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier = expression;</td>
</tr>
<tr>
<td>Assignment. An expression is either 1) a literal from any of the types, 2) an identifier, 3) an expression in Table II.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>if(bool b) { statements } else { statements }</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional statement. Its semantics is the same as in the corresponding statement in Java or C.</td>
</tr>
</tbody>
</table>

Table II: Functions in FPHILE.

2 Called there generalized substitutions.
tended precision is used in intermediate calculations. The only difference with Java is that the implicit base in literals is 2 in FPPhile, where in Java is 10. For instance, the literal 1.25e7 indicates the number 1.25 \cdot 10^7 in Java, 1.25 \cdot 2^7 in FPPhile. In both cases values are rounded to nearest, ties to even.

IV. ANNOTATIONS

FPPhile features also assumptions and assertions described in Table III. Such statements may contain boolean (sub-)expressions of the form \textit{stable(e@r)}, where \textit{e} and \textit{r} are \textit{float} typed expressions over the program state. FPPhile is thus not limited in proving stability properties, but anything that can be expressed in the assertions.

Let \( \eta, \rho \) be two precisions, \( \eta \leq \rho \). An occurrence of the expression \textit{stable(e@r)} in program \( P_\eta \) evaluates to true iff the relative error between \( e_\eta \) and \( e_\rho \) is smaller than \( r_\rho \). Here, the indices \( \eta, \rho \) indicate that an expression is interpreted in the indicated precision. Otherwise, the expressions are syntactically identical up to floating-point literals, which are adjusted to fit the precision. Intuitively, \textit{stable(e@r)} means that recomputing \( P \) with the higher precision does not change the result (relatively) more than \( r \). We can say that \( e \) is \( r \)-stable whenever \textit{stable(e@r)} evaluates to true. Informally, a program is \( r \)-stable whenever its result expression is \( r \)-stable. What is meant by result will be clear from the context.

The \textit{stable} assertion compares a value arising in a low-precision run with a value arising in a high-precision run \textit{at the same control point}. However, running the programme in different precisions can lead to different control flows. Therefore, for guaranteeing that the \textit{stable} assertion has the aforementioned meaning, it is allowed only in statements that are not nested in a loop body nor in a conditional branch. For instance consider program \( P \) depicted in Listing 1. Assuming \( \eta = (8, 24) \) and \( \rho = ((11, 53)) \), the value assigned to \( f \) in line 1 always overflows in \( P_\eta \) but not in \( P_\rho \). Therefore in line 3 no comparison can be made because no run of \( P_\eta \) reaches it.

```plaintext
float f = 1.0e12f * 1.0e20f;
if(f < INFF){
    assert stable(f @ 0.0f);
} else{
    ...
}
```

Listing 1: Where \textit{stable} makes no sense.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Name(s)</th>
<th>Arguments’ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>stable</td>
<td>(float e @ float r)</td>
</tr>
</tbody>
</table>

Returns \textit{true} if the value of expression \textit{e} is \( r \)-stable, \textit{false} otherwise.

Table IV: Annotation functions in FPPhile.

V. CLONING AND Merging Programs

We now define the transformation of the original program into a new program which executes the original code twice, in an interleaved manner, on two copies of the state space, one with original precision, and one with the higher precision. While all conventional statements are doubled, the assumption/assertion statements are not doubled, neither are the loop invariants. These ‘embedded specification’ constructs only talk about the original copy of the state space, with one exception: (sub)expressions of the kind \textit{stable(e@r)}, and only those, relate the original state with the higher precision state, see below.

We assume the original program to not contain any primed variable names (like \( x' \)), nor should it contain \textit{hifloat} declarations. In the following, priming expressions or statement lists is a shortcut for an operation which \( a \) replaces each occurrence of a variable with its primed version, and \( b \) replaces each occurrence of a \textit{float} literal, like \( 1.9e-10f \), with the corresponding \textit{hifloat} literal, like \( 1.9e-10h \).

The \textit{cloneAndMerge} function takes a statement list as input and returns another statement list.

\[
\begin{align*}
\text{cloneAndMerge}(x = e; R) &= x = e; x' = e'; \text{cloneAndMerge}(R) \\
\text{cloneAndMerge}(\text{if} \ T \ \text{else} \ E; R) &= \text{if} \ (T') \ \text{else} \ (E); \text{cloneAndMerge}(R) \\
\text{cloneAndMerge}(\text{if} \ (\beta) \ T \ \text{else} \ E; R) &= \text{if} \ ((\beta)) \ T' \ \text{else} \ E; \text{cloneAndMerge}(R)
\end{align*}
\]

The pattern \( 'x = e; R' \) matches any statement list that starts with an arbitrary assignment, followed by a \textit{Remaining} list of statements. For other first statements, this works accordingly.

When cloning declarations, \textit{float} becomes \textit{hifloat}. Variables of other types are cloned in the same type.

\[
\begin{align*}
\text{cloneAndMerge}(\text{float } x; R) &= \text{float } x; \text{hifloat } x'; \text{cloneAndMerge}(R) \\
\text{cloneAndMerge}(\text{int } x; R) &= \text{int } x; \text{cloneAndMerge}(R)
\end{align*}
\]

The extension to multiple variable declarations is obvious. For instance, \textit{hifloat} \( x,y \) becomes \textit{float} \( x,y \); \textit{hifloat} \( x',y' \).

If-statements are doubled in the following way:

\[
\begin{align*}
\text{cloneAndMerge}(\text{if} \ (\beta) \ (T) \ \text{else} \ (E); R) &= \text{if} \ ((\beta)) \ T' \ \text{else} \ (E); \text{cloneAndMerge}(R) \\
\text{cloneAndMerge}(\text{if} \ (\beta) \ (T \ \text{else} \ E; R) &= \text{if} \ ((\beta)) \ (T') \ \text{else} \ (E); \text{cloneAndMerge}(R)
\end{align*}
\]

As \( \beta \) and \( \beta' \) evaluate on the two copies of the state with different precision, they may or may not evaluate to the same truth value. Note that the different branches of the if are not tightly interleaved. For instance, the entire \( T \) or \( E \) is executed before \( T' \) or \( E' \) is. The fact that we only interleave on the top-level would be a problem if \textit{stable(e@r)} expressions were allowed in nested statements. But they are not, precisely because we would not know which ‘twin’ state to compare to in the case the cloned program has a different control flow. When translating a (top-level) while loop, however, we can allow for an unequal number of iterations in both copies, as the reader can see below.

But first, we translate specification statements.

\[
\begin{align*}
\text{cloneAndMerge}((\text{assert} \ \text{assume}) \ i; R) &= (\text{assert} \ \text{assume}) \ i'; \text{cloneAndMerge}(R)
\end{align*}
\]
Note that the `assert` resp. `assume` statement is not doubled. Only one copy of it remains in the merged program, but with the condition \( \iota \) instead of \( \iota^* \). The "-ed version of a condition is constructed by replacing all sub-expressions of the form \( \text{stable}(e@r) \) with \( \text{abs}(e - e')/e' \) \( <= r' \). This way, the \( \iota^* \) talks about the original precision state everywhere but in the stability predicate where it finally relates both states. The whole effort of computing on a cloned, higher precision state is only done for the sake of referring to it in the (translation of) the stability predicate!

Why is \( \text{stable}(e@r) \) translated to \( \text{abs}(e - e')/e' \) \( <= r' \)? Because it intuitively says that \( r \) is an upper bound of the relative difference of the versions of \( e \) in the lower and the higher precision. Note that the bound, and all operations \( (-, /, <=) \) and expressions, are interpreted in the floating-point type, not in mathematical reals, because there is no SMT solver supporting constraints with mixed floating-point/real numbers and operations. Note also that the operations in the translation of \( \text{stable} \) are interpreted in the higher precision. This way, we can represent errors that may not be representable in the lower precision.

Outside the `assert` resp. `assume` statements, the stability predicate can only appear in the loop-invariant of a (top-level) predicate. An invariant \( \iota \) is translated to \( \iota^* \) in the same way as in the `assert` resp. `assume` conditions.

\[
\text{cloneAndMerge}(\text{keep}(\iota)\text{while}(\beta)(S) \ R) = \\
\text{keep}(\iota^*)\text{while}(\beta \lor \beta')(S)\text{else}() \\
\text{if}(\beta')(\text{S'}\text{else}()) \\
\text{cloneAndMerge}(R)
\]

The slightly more intricate way in which we merge `while` statements (\( \lor \)-ing the \( \beta \)s and conditionally executing the loop body copies) allows us to reason about stability of loop computations even if the loops iterate differently often in both precisions. Both loop bodies alternate until one loop condition becomes false, after which the other loop continues in case—and for as long as—its condition is still true. The loop invariant of the merged program, \( \iota^* \), must always hold after executing both \( \text{if}s \). For the non-stability part of \( \iota^* \), executing \( S' \) does not matter anyway. As for the (translated) stability predicates in \( \iota^* \), the proof scheme implemented by this transformation allows to verify those cases where one precision, e.g., the higher one, iterates further, approximating some ideal mathematical value even better, but still not diverging from the lower precision value all too much. This way, stability after exiting the loop can be proved in more cases as when insisting on equally many iterations. Still, there are programs that enjoy stability after exiting the loop that cannot be proved stable with this scheme. The situation is similar in inductive theorem proving, where each induction scheme fails to prove certain theorems.

As an example, \( P \) is given in Listing 2. The transformed program appears in Listing 3.

```plaintext
Listing 2: Original program

bool singlePrecision;  
singlePrecision = true;  
float g, f, diff;  
diff = 2048.0;  
g = 1.0e35;  
f = g + diff;  
if(f <= g){  
singlePrecision = true;  
}  
else{  
singlePrecision = false;  
}

assert stable(f @ 1.0e-24);
```

```plaintext
Listing 3: Program after transformation

bool singlePrecision;  
singlePrecision = true;  
float g, f, diff;  
if (f <= g){  
singlePrecision = true;  
}  
else{  
singlePrecision = false;  
}

assert stable(f @ 1.0e-24);
```

The merge phase produces the program in Listing 3. This example also suggests the rationale of forbidding stability assertions inside scopes that could be executed depending on the float precision available. Let the types `float` and `hifloat` be respectively the IEEE single and double precision format. The guard of the first `if` statement evaluates to true, whereas the second to false. Having a comparison in any of the `if` branches would refer to a comparison with a computation that has not happened. In general one cannot assume that the execution traces of a two same program with different precisions are the same. Thus the comparison is generally meaningful in those statements that are executed regardless of the precision used, and top-level statements have this property.

### VI. Experiments with FPhile

In the following paragraphs some experiments of FPhile are illustrated\(^3\). The tests are run on a machine mounting an Intel\textsuperscript{\textregistered} Core\textsuperscript{TM} i5 CPU 760 at 2.80 Ghz, with 12 Gib RAM running a 64-bit Ubuntu 12.04. The SMT solver used here is Z3, using the latest build (4.3.2).

The stability analysis compares precisions \( \eta < \rho \), with \( \eta = (8, 24) \) and \( \rho = (11, 53) \). \( \eta \) and \( \rho \) correspond respectively to the IEEE single and double precisions, which in turn correspond to the `float` and `double` types in programming languages like C or Java. Whenever possible the experiments finding a counterexample are compared with random testing, being it

---

\(^3\)FPhile and the presented inputs can be found at http://www.cse.chalmers.se/~gabpag/devel/fphile/
the closest form of bug finding in that requires a comparable amount of engineering work and analysis from the developer\textsuperscript{4}. Test cases were run using the TestNG framework \cite{10} and a Java translation of the programs. It is reasonable to include the overhead of a testing framework since they are widely used in practice.

A deeper comparison and analysis of a selection of the experiment is given in Section VII.

If \( r_0 \) mantissa bits are available, an ulp is \( 2^{-r_0} \). Let \( r_0 = 2^{-24} \). Proving \( r_0\)-stability of a program \( P \) means proving that the float result differs from the \texttt{hifloat} result for at most half a unit in the last place. For this reason \( r_0\)-stability is often used in the following experiments. In each of the following paragraphs a roman numeral indexed list describes the experiments.

\textbf{A. Sterbenz’s lemma}

Sterbenz’s lemma states that if two floating-point numbers \( x \) and \( y \) are such that \( y/2 \leq x \leq 2y \), then \( x - y \) is an exact floating-point number: no round-off error is made.

\begin{verbatim}
float x,y,z;
assume y/2.0 <= x && x <= 2.0*y &&
normal(y/2.0) && stable(y @ 0.0) && stable(x @ 0.0) &&
abs(2.0*y) < INFF;
z = (x-y);
assert stable(z @ 0.0);
Listing 4: Proving Sterbenz’s theorem.
\end{verbatim}

The experiments show that \( i \) the program in Listing 4 is 0-stable.

\textbf{B. Integer multiplication}

The following program multiplies two floats, treated as integers, by repeatedly summing 1. The result is then compared with the actual multiplication, showing that under the program’s assumptions \( i \) a sequence of additions is the same as multiplying, and \( ii \) the result is 0-stable.

\begin{verbatim}
float x,y,z;
assume y/2.0 <= x && x <= 2.0*y &&
normal(y/2.0) && stable(y @ 0.0) && stable(x @ 0.0) &&
abs(2.0*y) < INFF;
z = (x-y);
assert stable(z @ 0.0);
Listing 4: Proving Sterbenz’s theorem.
\end{verbatim}

The experiments show that \( i \) the program in Listing 4 is 0-stable.

\textbf{C. Subtraction}

Results of operations like addition or subtraction diverge from the real counterparts whenever the difference of the orders of magnitude of the operands is bigger than the available mantissa precision. The experiments show that the program \( i \) is \( r_0\)-stable (Listing 6) \( ii \) it is not \( +1.9999998 \cdot 2^{-25}\)-stable, yielding a counterexample.

\begin{verbatim}
float x, y, z;
assume stable(x @ 0.0f) && stable(y @ 0.0f);
assume x > y && abs(x-y)<INFF;
z = (x-y);
assert stable(z @ 1.0e-24);
Listing 6: A subtraction.
\end{verbatim}

\textbf{D. Non-Termination}

Determining termination is subtle when loops are guarded by comparisons of floating-point expressions. Listing 7 is the input of this experiment that shows \( i \) the non-termination of the program. \textsc{FPhile} returns the counterexample \( i = +1.5312576290453125e-86 \) and \( j = +1.53125e-3f \), which causes the program to loop indefinitely: the program enters the loop, but \( \) variable \( i \) does not increase because \( i + 0.25 = i \).

\begin{verbatim}
float i, j;
assume stable(i @ 0.0) && stable(j @ 0.0);
assume i < j && i > 0.0;
keep(true) shrinking(j-i) while(i < j){
  i = i + 0.25;
} assert true;
Listing 7: A non-terminating program.
\end{verbatim}

\textbf{E. Approximation of π}

The perimeter of a convex regular polygon with \( N \) edges inscribed inside the unit circumference can be used to approximate \( π \): let \( L_N \) be the side of such a polygon, then \( π \approx \frac{NL_N}{2} \) \cite{11}. The recursive expression

\begin{equation}
L_{2N}^2 = 2 \left(1 - \sqrt{1 - L_N^2/4}\right) \quad (1)
\end{equation}

suggests an algorithm to compute the approximation. An alternative form is the following:

\begin{equation}
\frac{L_{2N}^2}{4} = \frac{L_N^2/4}{2 \left(1 + \sqrt{1 - L_N^2/4}\right)} \quad (2)
\end{equation}

The algorithm starts from \( N = 4 \), thus \( L_N = \sqrt{2} \). The experiments show that \( i \) Equation 1 is not anymore \( r_0\)-stable after 1 iteration \( ii \) Equation 2 is not anymore \( r_0\)-stable after 11 iterations. Thus Equation 2 is better behaved.

\begin{verbatim}
float x, y, z;
assume stbal(x @ 0.0f) && stbal(y @ 0.0f);
assume x > y && abs(x-y)<INFF;
z = (x-y);
Listing 6: A subtraction.
\end{verbatim}

\textbf{Comparison with testing:} Testing can detect this instability in less time than proving with \textsc{FPhile}, as it is just a constant evaluation.

\textbf{F. Inaccurate calculation}

This example shows how \textsc{FPhile} can be used to detect an inaccurate calculation due to roundoff. Consider the function

\begin{equation}
p(x) = \begin{cases}
\frac{(x-c)^2 - (x-c_0)^2}{c_0^2} & \text{if } 2^7 \leq x \leq 2^7 + \frac{25}{486} \\
0 & \text{otherwise}
\end{cases} \quad (3)
\end{equation}
where $\epsilon_1 = 2^{-15}$ and $\epsilon_2 = 2^{-16}$. The program $P$ in Listing 8 computes Equation 3 as it is defined. The expression is inaccurate as figure 1 shows; in fact, $p(128) \approx 409.6$, but on that input $P_E$ produces the value 256.0, whereas $P_P$ produces $\approx 409.0$. The experiments show that Listing 8 $i$) is not $r_0$-stable.

![Figure 1: The function defined in Equation 3. Dashed line: single precision. Solid line: double precision.](image)

Comparison with testing: It takes around $10^4$ random tests and 1.6 minutes to find 100 inputs showing instability of an implementation of Equation 4. FPHILE finds a counterexample in 2.68 minutes. Kahan’s version is empirically better than that: performing experiment $ii$ by running $10^7$ tests found no counterexamples in 27 hours, whereas FPHILE finds one in 6.5 minutes.

H. Trisection of an angle

The problem of trisecting an acute angle $\alpha$ using only a compass and an unmarked straightedge [13], [14] can be expressed as the solution to the equation

$$f(t, T) = (3 - t^2)t - (1 - 3t^2)T = 0$$

where $T = \tan \alpha$. The equation is solved approximately by some iterations of Newton’s method. The experiments show that for $i)$ 2 and $ii)$ 10 times is not $r_0$-stable. Both the attempts show that the algorithm is unstable even for values of the angles $\frac{\pi}{9} < \alpha < \frac{\pi}{4}$ where less sensitivity would be expected by common sense, i.e. without a formal analysis.

Comparison with testing: Running $10^4$ tests on the two experiments above results in around 3500 failing inputs; in the case of experiment $ii$ it does so in 4.7 minutes, whereas FPHILE finds a counterexample in 7 minutes.

1. Variance estimation

The variance estimator

$$s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

is found in a more efficient form in some books [15]:

$$s_n^2 = \frac{1}{n-1} \left( \sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left( \sum_{i=1}^{n} x_i \right)^2 \right)$$

Equation 8 needs only one pass in the data and is obtained by simple rearrangement. The experiments were run on the two methods. The number of arbitrary samples was 10 to limit the number of iterations and allow loop unwinding. Input values are constrained so that no overflows occur. Experiments show that $i)$ Variance computed with Equation 8 can be negative $ii)$ Variance computed with Equation 7 is always positive $iii)$ Equation 8’s implementation is $r_0$-unstable. $iv)$ Equation 7’s implementation is $r_0$-unstable.

Comparison with testing: Testing in experiment $iii$ took 3.75 minutes and $10^4$ tests showing $\approx 2500$ failures. FPHILE took around 40 minutes to deliver a counterexample. In experiment $iv$ testing took 2.6 minutes and $10^4$ tests showing $\approx 3500$ failures. FPHILE took around 20 minutes to deliver a counterexample. Interestingly, in experiment $i$ running $10^7$ tests took 1.26 hours and could not show that the variance computed with Equation 8 can be negative, which FPHILE shows in 18.5 minutes.

VII. Performance analysis

A. FPHILE’s performances

In the following the experiments are identified with the shorthand section-subsection-item. For instance, the experiment that shows the stability of Sterbenz’s formula in subsection VI-A , experiment $i$ is referenced as VI-A-i. The experiments’ measures

G. Heron’s formula

The formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a + b + c}{2}$$

computes the area $A$ of the triangle $abc$. This textbook expression is not numerically stable and a better form is given in [12]:

$$A = \frac{1}{4} \sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}$$

The experiments show that assuming the area not to be infinite, $i)$ Equation 4 and $ii)$ Equation 5 are not $r_0$-stable.

Listing 8: Implementation of Equation 3

```c
float f, a, c, low, high, eps1, eps2, result;
low = 0.25e9;
high = 0.25001220703125e9;
eps1 = 0.1e-15;
eps2 = 0.5e-15;
assume abs(f) < INFINITY;
assume normal(f) & stable(f @ 0.0);
if(f<low){
    result = 200.0f;
} else{
    a = f - eps1;
c = f - eps2;
    result = (a*a - c*c) / (eps2);
    result = 200.0f;
}
assert stable(result @ 1.0e-24f);
```

I. Variance estimation

The variance estimator

$$s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

is found in a more efficient form in some books [15]:

$$s_n^2 = \frac{1}{n-1} \left( \sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left( \sum_{i=1}^{n} x_i \right)^2 \right)$$

Equation 8 needs only one pass in the data and is obtained by simple rearrangement. The experiments were run on the two methods. The number of arbitrary samples was 10 to limit the number of iterations and allow loop unwinding. Input values are constrained so that no overflows occur. Experiments show that $i)$ Variance computed with Equation 8 can be negative $ii)$ Variance computed with Equation 7 is always positive $iii)$ Equation 8’s implementation is $r_0$-unstable. $iv)$ Equation 7’s implementation is $r_0$-unstable.

Comparison with testing: Testing in experiment $iii$ took 3.75 minutes and $10^4$ tests showing $\approx 2500$ failures. FPHILE took around 40 minutes to deliver a counterexample. In experiment $iv$ testing took 2.6 minutes and $10^4$ tests showing $\approx 3500$ failures. FPHILE took around 20 minutes to deliver a counterexample. Interestingly, in experiment $i$ running $10^7$ tests took 1.26 hours and could not show that the variance computed with Equation 8 can be negative, which FPHILE shows in 18.5 minutes.
showed correlation between running time and: 1) Insatisfiability of formula. A program’s validity accounts to prove its weakest precondition unsatisfiable, which often can be harder than proving satisfiability. 2) Precision of data types. Since the prover is based on bit-blasting, wider data types translate to more variables in the problem delivered to the SAT solver by the SMT solver. 3) Number of SMT variables. Since modern SAT solvers are mainly based on DPLL [16] the more the variables, the more decisions are made. For this reason proving stability takes more time to run compared to proofs involving only one floating-point type, as twice as much variables are declared and half of them have higher precision. The number of SMT variables is affected by the unwinding of loops. 4) Expensive operations such as rounding to integral, square root, or multiplication.

The current implementation of Z3 follows the one described in such as rounding to integral, square root, or multiplication. the textbook Heron formula in Equation 4: no bug was found after 107 tests. Instead FPhile found the counterexample $a = +1.751953125e109$ $b = +1.751953125e109$ $c = +1e-126$, where edge $c$ is the minimum normal number in the (8, 24) precision. This is a clue that the program might not work properly when one edge is close to subnormals, or the intermediate results are subnormals.

Random testing can be more efficient than verification-based failure detection such as FPhile when 1) the density of failure-inducing inputs is such that a failure is found after a reasonable amount of time and guesses 2) the constraints describing the input are easy to fulfill. In the case where the latter does not hold, a considerable amount of time can be spent in finding the inputs fulfilling the preconditions of the tests: compare in Table VI the differences in execution time in rows VI-F-i, VI-G-ii, VI-I-i due to input generation. Input generation is what constraint solvers and theorem provers are good at.

### VIII. Related Work and Conclusions

This work is inspired by Kahan [1]. He proposes several ways for debugging a floating-point computation, called mindless because they do not require deep knowledge of numerical analysis. Among those, he proposes an aware debugger that monitors a program while executing a more precise copy of it. Benz et al. [18] implemented such debugger. The test cases provided by the analysis presented in the present paper can direct such debugging. The ongoing CORVETTE project and the PRECIMONIOUS tool [19] have as objective the optimisation of the tradeoff speed/accuracy in programs which depend on the floating point precision: the idea is to detect which are the parts of programs that need a wider precision, by means of a collection of code analyses. Theorem proving seems not to be part of the project, unlike in FPhile. The tools F-SOFT [20] and ARIADNE [21] use model checking and SMT solvers. ARIADNE uses a modified version of Z3. ARIADNE and F-SOFT reason about floating-point numbers in terms of reals: that is, floating-point program variables are translated to real functions in the SMT solver. FPhile does instead directly reason on floating-point numbers.

Until recently, SMT solvers were lacking a floating-point theory. Rümmer and Wahl [22] proposed a floating-point SMT theory based on the IEEE standard [9]. This theory is implemented in the Z3 SMT solver. The experiments proved also to be an effective testbed for the solver, helping detecting several bugs in it. Floating-point verification traditionally uses interval analysis [23], or refinements/generalisations of it such as affine arithmetics [24]. The abstract interpretation tool FLUCTUAT [25] uses such approaches to verify properties of floating-point C programs. FPhile can reason about intervals since the stable predicate implicitly introduces them; yet it is not as developed as the mentioned tools on that direction. The Frama-C framework [26] is the result of a long tradition of software analysis, often focused on floating-point analysis [27], [28]. It uses several different theorem provers and tools to deduce properties of programs, among which Gappa [29] which is also based on an interval model of the computation. Gappa needs concrete bounds as preconditions to compute the interval of values that the function under analysis will return, whereas FPhile can work with symbolic values by virtue of the underlying SMT

---

**Table V: Performance: FPhile.**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>vars</th>
<th>/</th>
<th>(\sqrt{\text{\textbackslash{}vars}})</th>
<th>Memory</th>
<th>⏰</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI-F-i, unstable</td>
<td>2</td>
<td>54</td>
<td>37</td>
<td>0</td>
<td>674MB</td>
</tr>
<tr>
<td>VI-C-ii, unstable</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>506MB</td>
</tr>
<tr>
<td>VI-A-i, stable</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>507MB</td>
</tr>
<tr>
<td>VI-G-ii, unstable</td>
<td>6</td>
<td>52</td>
<td>4</td>
<td>13</td>
<td>885MB</td>
</tr>
<tr>
<td>VI-H-ii, unstable</td>
<td>22</td>
<td>216</td>
<td>38</td>
<td>16</td>
<td>4.7GB</td>
</tr>
<tr>
<td>VI-I-i, non-valid</td>
<td>30</td>
<td>13</td>
<td>2</td>
<td>0</td>
<td>209MB</td>
</tr>
<tr>
<td>VI-C-i, stable</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>593MB</td>
</tr>
</tbody>
</table>

The experiments VI-B-i (7 hours) and VI-B-ii (several days) were not included in the statistics being outliers: their running times were strongly impaired by the presence of 10 rounding to integral operation, thus confirming this operation being the one that creates harder problems. Approaches with loop invariants in principle should make the number of variables constant. During the experiments it was realized that the simple use of invariant yields, as it is expectable, less tight bounds.

**A. FPhile and testing**

Table VI compares FPhile and random testing on some of the experiments discussed above.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>(\square_{\phi\text{\textbackslash{}\text{phile}}})</th>
<th>#tests</th>
<th>#failed</th>
<th>(\square_{\text{\textbackslash{}tests}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI-F-i</td>
<td>22s</td>
<td>(10^7)</td>
<td>0</td>
<td>23m</td>
</tr>
<tr>
<td>VI-C-ii</td>
<td>2m</td>
<td>(10^5)</td>
<td>2</td>
<td>1h</td>
</tr>
<tr>
<td>VI-G-ii</td>
<td>6.5m</td>
<td>(10^7)</td>
<td>0</td>
<td>27h</td>
</tr>
<tr>
<td>VI-H-ii</td>
<td>7m</td>
<td>(10^4)</td>
<td>3262</td>
<td>4.7m</td>
</tr>
<tr>
<td>VI-I-i</td>
<td>18.5m</td>
<td>(10^7)</td>
<td>0</td>
<td>1.26h</td>
</tr>
</tbody>
</table>

A posteriori\(^5\) the testing performance can be motivated by some probabilistic reasoning. For instance in the case of VI-F-i an upper bound to the probability of finding an error over \(n\) tests is

\[ p_n = \left(1 - \frac{\phi}{2^{32}} \right)^n \]

\(^5\)That is, once one knows the set of failure-inducing inputs for the program under test.
solver. Unlike FPhile, the (in-)stability cannot be determined automatically by Frama-C. The aim of those tools is different. FPhile targets programmers, Frama-C targets verification and numerical experts (due to the interactivity).

The FPhile language was inspired by the programming language and verification system Dafny[30], although the latter is much more expressive. The FPhile framework aims at investigating the application and the points of strength of a pure floating-point approach to automated verification. Many of the pitfalls for numerically inexperienced programmers are rooted in the erroneous assumption that the property of reals hold also for floating-point numbers. Integrating FPhile in programming environments would hopefully widen the numerical awareness of users. A hurdle in automation in formal methods and automated reasoning is handling loops with invariants. In floating-point verification — for the reasons explained above — this becomes challenging even for experts, and experiments with FPhile confirmed this. Future work will concentrate on understanding how strong loop invariants can be in FPhile, and what kind of properties one can write without referring to reals. The risk is to add to the intricacies of numerical analysis those of verification technology. More language features such as arrays are planned for the near future, decreasing the gap to mainstream languages. Without doubt, the work can, at its current stage, not replace conventional usage of reals. The risk is to add to the intricacies of numerical analysis to floating-point numbers. Integrating FPhile in programming environments would hopefully widen the numerical awareness of users. A hurdle in automation in formal methods and automated reasoning is handling loops with invariants.

**Acknowledgements**

The authors would like to thank Dr. Christoph Wintersteiger of Microsoft Research for his decisive help and support with the Z3 SMT solver.

**References**


