Eliminating Unstable Tests in Floating-Point Programs

Laura Titolo[1], César A. Muñoz[2], Marco A. Feliú[1], and Mariano M. Moscato[1]

[1] National Institute of Aerospace,
{laura.titolo,marco.feliu,mariano.moscato}@nianet.org *
[2] NASA Langley Research Center,
cesar.a.munoz@nasa.gov

Abstract. Round-off errors arising from the difference between real numbers and their floating-point representation cause the control flow of conditional floating-point statements to deviate from the ideal flow of the real-number computation. This problem, which is called test instability, may result in a significant difference between the computation of a floating-point program and the expected output in real arithmetic. In this paper, a formally proven program transformation is proposed to detect and correct the effects of unstable tests. The output of this transformation is a floating-point program that is guaranteed to return either the result of the original floating-point program when it can be assured that both its real and its floating-point flows agree or a warning when these flows may diverge. The proposed approach is illustrated with the transformation of the core computation of a polygon containment algorithm developed at NASA that is used in a geofencing system for unmanned aircraft systems.

Key Words: Floating-point numbers, round-off error, program transformation, test instability, formal verification.

1 Introduction

Floating-point numbers are widely used to represent real numbers in computer programs since they offer a good trade-off between efficiency and precision. The round-off error of a floating-point expression is the difference between the ideal computation in real arithmetic and the actual floating-point computation. These round-off errors accumulate during numerical computations. Besides having a direct effect on the result of mathematical operations, round-off errors may significantly impact the control flow of a program. This happens when the guard of a conditional statement contains a floating-point expression whose round-off error makes the actual Boolean value of the guard differ from the value that would

* Research by the first, the third, and the fourth authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

be obtained assuming real arithmetic. In this case, the conditional statement is called an *unstable test.* Unstable tests are an inherent feature of floating-point programs. In general, it is not possible to completely avoid them. However, it is possible to mitigate their effect by transforming the original program into another program that conservatively (and soundly) detects and corrects unstable tests.

This paper presents a program transformation technique to transform a given program into a new one that returns either the same result of the original program or a warning when the real and floating-point flows may diverge. This transformation is parametric with respect to two Boolean abstractions that take into consideration the round-off error in the expressions occurring in the guard. The transformation replaces the unstable conditions with more restrictive conditions that are guaranteed to preserve the control flow of stable tests. The correctness of the proposed transformation is formally verified in the Prototype Verification System (PVS) [16].

The remainder of the paper is organized as follows. Section 2 provides technical background on floating-point numbers and round-off errors. The proposed program transformation technique is presented in Section 3. Section 4 illustrates this technique by transforming the core logic of an algorithm for polygon containment that is part of a geofencing system developed by NASA. Section 5 discusses related work and Section 6 concludes the paper.

2 Round-Off Errors and Unstable Tests

A floating-point number can be formalized as a pair of integers $(m, e) \in \mathbb{Z}^2$, where m is called the *significand* and e the *exponent* of the float [10,1]. A floating-point *format* f is defined as a pair of integers (p, e_{min}), where p is called the *precision* and e_{min} is called the *minimal exponent*. For instance, IEEE single and double precision floating-point numbers are specified by the formats $(24, 149)$ and $(53, 1074)$, respectively. A *canonical* float is a float such that is either a normal or subnormal. A *normal* float is a float such that the significand cannot be multiplied by the radix and still fit in the format. A *subnormal* float is a float having the minimal exponent such that its significand can be multiplied by the radix and still fit in the format. Henceforth, $\mathbb{F}$ will denote the set of floating-point numbers in canonical form and the expression $\tilde{v}$ will denote a floating-point number (m, e) in $\mathbb{F}$. A conversion function $R : \mathbb{F} \to \mathbb{R}$ is defined to refer to the real number represented by a given float, i.e., $R((m, e)) = m \cdot \beta^e$.

The expression $F_f(r)$ denotes the floating-point number in format f *closest* to r. The format f will be omitted when clear from the context. Let $\tilde{v}$ be a floating-point number that represents a real number r, the difference $|R(\tilde{v}) - r|$ is called the *round-off error* (or *rounding error*) of $\tilde{v}$ with respect to r.

2.1 Unstable tests

Given a set $\widetilde{\Omega}$ of pre-defined floating-point operations, the corresponding set Ω of operations over real numbers, a finite set $\mathbb{V}$ of variables representing real values,

and a finite set $\widetilde{\mathbb{V}}$ of variables representing floating-point values, where $\mathbb{V}$ and $\widetilde{\mathbb{V}}$ are disjoint, the sets $\mathbb{A}$ and $\widetilde{\mathbb{A}}$ of arithmetic expressions over real numbers and over floating-point numbers, respectively, are defined by the following grammar.

$$A ::= d \mid x \mid op(A, \ldots, A), \qquad \widetilde{A} ::= \tilde{d} \mid \tilde{x} \mid \widetilde{op}(\widetilde{A}, \ldots, \widetilde{A}),$$

where $A \in \mathbb{A}$, $d \in \mathbb{R}$, $x \in \mathbb{V}$, $op \in \Omega$, $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\tilde{d} \in \mathbb{F}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, $\widetilde{op} \in \widetilde{\Omega}$. It is assumed that there is a function $\chi_r : \widetilde{\mathbb{V}} \to \mathbb{V}$ that associates to each floating-point variable $\tilde{x}$ a variable $x \in \mathbb{V}$ representing the real value of $\tilde{x}$. The function $R_{\mathbb{A}} : \widetilde{\mathbb{A}} \to \mathbb{A}$ converts an arithmetic expression on floating-point numbers to an arithmetic expression on real numbers. It is defined by simply replacing each floating-point operation with the corresponding one on real numbers and by applying R and χ_r to floating-point values and variables, respectively.

Boolean expressions are defined by the following grammar.

$$B ::= \mathit{true} \mid \mathit{false} \mid B \wedge B \mid B \vee B \mid \neg B \mid A < A \mid A = A \mid \widetilde{A} < \widetilde{A} \mid \widetilde{A} = \widetilde{A},$$

where $A \in \mathbb{A}$ and $\widetilde{A} \in \widetilde{\mathbb{A}}$. The conjunction $\wedge$, disjunction $\vee$, negation $\neg$, *true*, and *false* have the usual classical logic meaning. The symbols $\mathbb{B}$ and $\widetilde{\mathbb{B}}$ denote the domain of Boolean expressions over real and floating-point numbers, respectively. The function $R_{\mathbb{B}} : \widetilde{\mathbb{B}} \to \mathbb{B}$ converts a Boolean expression on floating-point numbers to a Boolean expression on real numbers. Given a variable assignment $\sigma : \mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{B}}(\sigma, B) \in \{\mathit{true}, \mathit{false}\}$ denotes the evaluation of the real Boolean expression B. Similarly, given $\widetilde{B} \in \widetilde{\mathbb{B}}$ and $\widetilde{\sigma} : \widetilde{\mathbb{V}} \to \mathbb{F}$, $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\widetilde{\sigma}, \widetilde{B}) \in \{\mathit{true}, \mathit{false}\}$ denotes the evaluation of the floating-point Boolean expression $\widetilde{B}$.

The expression language considered in this paper contains binary and n-ary conditionals, let expressions, arithmetic expressions, and a warning exceptional statement. Given a set Σ of function symbols, the syntax of program expressions in $\mathbb{S}$ is given by the following grammar.

$$\begin{aligned} S ::= & \widetilde{A} \mid \mathit{if}\ \widetilde{B}\ \mathit{then}\ S\ \mathit{else}\ S \mid \mathit{if}\ \widetilde{B}\ \mathit{then}\ S\ [\mathit{elsif}\ \widetilde{B}\ \mathit{then}\ S]_{i=1}^{n}\ \mathit{else}\ S \\ & \mid \mathit{let}\ \tilde{x} = \widetilde{A}\ \mathit{in}\ S \mid \mathit{warning}, \end{aligned} \tag{2.1}$$

where $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\widetilde{B} \in \widetilde{\mathbb{B}}$, $\tilde{d} \in \mathbb{F}$, $d \in \mathbb{R}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, $\widetilde{op} \in \widetilde{\Omega}$, $\tilde{f} \in \Sigma$, and $n \in \mathbb{N}^{>0}$. The notation $[\mathit{elsif}\ \widetilde{B}\ \mathit{then}\ S]_{i=1}^{n}$ denotes a list of n *elsif* branches.

A program is a *function declaration* of the form $\tilde{f}(\tilde{x}_1, \ldots, \tilde{x}_m) = S$, where $\tilde{x}_1, \ldots, \tilde{x}_m$ are pairwise distinct variables in $\widetilde{\mathbb{V}}$ and all free variables appearing in S are in $\{\tilde{x}_1, \ldots, \tilde{x}_m\}$. The natural number m is called the *arity* of $\tilde{f}$. The set of programs is denoted as $\mathbb{P}$.

When if-then-else guards contain floating-point expressions, the output of the considered program is not only directly influenced by rounding errors, but also by the error of taking the incorrect branch in the case of unstable tests.

Definition 1 (Conditional Instability). *A function declaration $\tilde{f}(\tilde{x}_1, \ldots, \tilde{x}_n) = S$ is said to have an* unstable conditional *when its body contains a conditional statement of the form if $\tilde{\phi}$ then S_1 else S_2 and there exist two assignments $\tilde{\sigma} : \{\tilde{x}_1, \ldots, \tilde{x}_n\} \to \mathbb{F}$ and $\sigma : \{\chi_r(\tilde{x}_1), \ldots, \chi_r(\tilde{x}_n)\} \to \mathbb{R}$ such that for all $i \in$*

$\{1,\ldots,n\}$, $\sigma(\chi_r(\tilde{x}_i)) = R(\tilde{\sigma}(\tilde{x}_i))$ *and* $eval_{\mathbb{B}}(\sigma, R_{\mathbb{B}}(\tilde{\phi})) \neq \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\phi})$. *Otherwise, the conditional expression is said to be* stable.

In other words, a conditional statement (or test) $\tilde{\phi}$ is unstable when there exists an assignments from the free variables $\tilde{x}_i$ in $\tilde{\phi}$ to $\mathbb{F}$ such that $\tilde{\phi}$ evaluates to a different Boolean value with respect to its real valued counterpart $R_{\mathbb{B}}(\tilde{\phi})$. In these cases, the program is said to follow an *unstable path*, otherwise, when the flows coincide, it is said to follow a *stable path*.

2.2 Floating-Point Denotational Semantics

This section presents a compositional denotational semantics for the expression language of Formula (2.1) that models both real and floating-point path conditions and outputs. This semantics is a modification of the one introduced in [13] and [21]. The proposed semantics collects for each combination of real and floating-point program paths: the real and floating-point path conditions, two symbolic expressions representing the value of the output assuming the use of real and floating-point arithmetic, respectively, and a flag indicating if the element refers to either a stable or an unstable path. This information is stored in a *conditional tuple*.

Definition 2 (Conditional Tuple). *A* conditional tuple *is an expression of the form* $\langle \eta, \tilde{\eta} \rangle_t \twoheadrightarrow (r, \tilde{r})$, *where* $\eta \in \mathbb{B}$, $\tilde{\eta} \in \widetilde{\mathbb{B}}$, $r \in \mathbb{A} \cup \{\perp_{\mathbf{u}}\}$, $\tilde{r} \in \widetilde{\mathbb{A}} \cup \{\perp_{\mathbf{u}}\}$, *and* $t \in \{\mathbf{s}, \mathbf{u}\}$.

Intuitively, $\langle \eta, \tilde{\eta} \rangle_t \twoheadrightarrow (r, \tilde{r})$ indicates that if the condition $\eta \wedge \tilde{\eta}$ is satisfied, the output of the ideal real-valued implementation of the program is r and the output of the floating-point execution is $\tilde{r}$. The sub-index t is used to mark by construction whether a conditional tuple corresponds to an unstable path, when $t = \mathbf{u}$, or to a stable path, when $t = \mathbf{s}$. The element $\perp_{\mathbf{u}}$ represents the output of the warning construct. Let $\mathbf{C}$ be the set of all conditional error bounds, and $\mathbb{C} := \wp(\mathbf{C})$ be the domain formed by sets of conditional error bounds.

An *environment* is defined as a function mapping a variable to a set of conditional tuples, i.e., $Env = \widetilde{\mathbb{V}} \to \mathbb{C}$. The empty environment is denoted as $\perp_{Env}$ and maps every variable to the empty set $\varnothing$.

Given $\nu \in Env$, the semantics of program expressions is defined in Fig. 1 as a function $\mathcal{E} : \mathbb{S} \times Env \to \mathbb{C}$ that returns the set of conditional tuples representing the possible real and floating-point computations and their corresponding path conditions. The semantics of a variable $\tilde{x} \in \widetilde{\mathbb{V}}$ consists of two cases. If $\tilde{x}$ belongs to the environment, then the variable has been previously bound to a program expression S through a let-expression. In this case, the semantics of $\tilde{x}$ is exactly the semantics of S. If $\tilde{x}$ does not belong to the environment, then $\tilde{x}$ is a parameter of the function. Here, a new conditional error bound is added with a placeholder $\chi_r(\tilde{x})$ representing the real value of $\tilde{x}$. The semantics of a floating-point arithmetic operation $\widetilde{op}$ is computed by composing the semantics of its operands. The real and floating-point values are obtained by applying the

$$\mathcal{E}[\![\tilde{d}]\!]_\nu := \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (R(\tilde{d}), \tilde{d})\}$$

$$\mathcal{E}[\![warning]\!]_\nu := \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (\bot_{\mathbf{u}}, \bot_{\mathbf{u}})\}$$

$$\mathcal{E}[\![\tilde{x}]\!]_\nu := \begin{cases} \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (\chi_r(\tilde{x}), \tilde{x})\} & \text{if } \nu(\tilde{x}) = \varnothing \\ \nu(\tilde{x}) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\widetilde{op}(\widetilde{A}_i)_{i=1}^n]\!]_\nu := \bigsqcup\{ \langle \bigwedge_{i=1}^n \phi_i, \bigwedge_{i=1}^n \tilde{\phi}_i\rangle_{\mathbf{s}} \twoheadrightarrow (op(r_i)_{i=1}^n, \widetilde{op}(\tilde{r}_i)_{i=1}^n) \mid \forall 1 \le i \le n: \\ \langle \phi_i, \tilde{\phi}_i\rangle_{\mathbf{s}} \twoheadrightarrow (r_i, \tilde{r}_i) \in \mathcal{E}[\![\widetilde{A}_i]\!]_\nu, \bigwedge_{i=1}^n \phi_i \not\Rightarrow false, \bigwedge_{i=1}^n \tilde{\phi}_i \not\Rightarrow false\}$$

$$\mathcal{E}[\![let\ \tilde{x} = \widetilde{A}\ in\ S]\!]_\nu := \mathcal{E}[\![S]\!]_{\nu[\tilde{x} \mapsto \mathcal{E}[\![\widetilde{A}]\!]_\nu]}$$

$$\mathcal{E}[\![if\ \widetilde{B}\ then\ S_1\ else\ S_2]\!]_\nu := \mathcal{E}[\![S_1]\!]_\nu \Downarrow_{(R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})} \sqcup\ \mathcal{E}[\![S_2]\!]_\nu \Downarrow_{(\neg R_{\mathbb{B}}(\widetilde{B}), \neg \widetilde{B})} \sqcup \\ \bigsqcup\{\langle \phi_2, \tilde{\phi}_1\rangle_{\mathbf{u}} \twoheadrightarrow (r_2, \tilde{r}_1) \mid \langle \phi_1, \tilde{\phi}_1\rangle_{\mathbf{s}} \twoheadrightarrow (r_1, \tilde{r}_1) \in \mathcal{E}[\![S_1]\!]_\nu, \\ \langle \phi_2, \tilde{\phi}_2\rangle_{\mathbf{s}} \twoheadrightarrow (r_2, \tilde{r}_2) \in \mathcal{E}[\![S_2]\!]_\nu\} \Downarrow_{(\neg R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})} \sqcup \\ \bigsqcup\{\langle \phi_1, \tilde{\phi}_2\rangle_{\mathbf{u}} \twoheadrightarrow (r_1, \tilde{r}_2) \mid \langle \phi_1, \tilde{\phi}_1\rangle_{\mathbf{s}} \twoheadrightarrow (r_1, \tilde{r}_1) \in \mathcal{E}[\![S_1]\!]_\nu, \\ \langle \phi_2, \tilde{\phi}_2\rangle_{\mathbf{s}} \twoheadrightarrow (r_2, \tilde{r}_2) \in \mathcal{E}[\![S_2]\!]_\nu\} \Downarrow_{(R_{\mathbb{B}}(\widetilde{B}), \neg \widetilde{B})}$$

$$\mathcal{E}[\![if\ \widetilde{B}_1\ then\ S_1\ [elsif\ \widetilde{B}_i\ then\ S_i]_{i=2}^{n-1}\ else\ S_n]\!]_\nu := \\ \bigsqcup_{i=1}^{n-1} \mathcal{E}[\![S_i]\!]_\nu \Downarrow_{(\widetilde{B}_i \wedge \bigwedge_{j=1}^{i-1} \neg \widetilde{B}_j, R(\widetilde{B}_i) \wedge \bigwedge_{j=1}^{i-1} \neg R(\widetilde{B}_j))} \\ \sqcup\, \mathcal{E}[\![S_n]\!]_\nu \Downarrow_{(\bigwedge_{j=1}^{n-1} \neg \widetilde{B}_j, \bigwedge_{j=1}^{n-1} \neg R(\widetilde{B}_j))} \sqcup \\ \bigsqcup\{\langle \eta_i, \tilde{\eta}_j\rangle_{\mathbf{u}} \twoheadrightarrow (r_i, \tilde{r}_j) \mid i, j \in \{1, \ldots, n-1\}, i \neq j,\ \langle \eta_i, \tilde{\eta}_i\rangle_{\mathbf{s}} \twoheadrightarrow (r_i, \tilde{r}_i) \in \mathcal{E}[\![S_i]\!]_\nu, \\ \langle \eta_j, \tilde{\eta}_j\rangle_{\mathbf{s}} \twoheadrightarrow (r_j, \tilde{r}_j) \in \mathcal{E}[\![S_j]\!]_\nu\} \Downarrow_{(\widetilde{B}_j \wedge \bigwedge_{k=1}^{j-1} \neg \widetilde{B}_k, R(\widetilde{B}_i) \wedge \bigwedge_{k=1}^{i-1} \neg R(\widetilde{B}_k))} \sqcup \\ \bigsqcup\{\langle \eta_i, \tilde{\eta}_n\rangle_{\mathbf{u}} \twoheadrightarrow (r_i, \tilde{r}_n) \mid i \in \{1, \ldots, n-1\},\ \langle \eta_i, \tilde{\eta}_i\rangle_{\mathbf{s}} \twoheadrightarrow (r_i, \tilde{r}_i) \in \mathcal{E}[\![S_i]\!]_\nu, \\ \langle \eta_n, \tilde{\eta}_n\rangle_{\mathbf{s}} \twoheadrightarrow (r_n, \tilde{r}_n) \in \mathcal{E}[\![S_n]\!]_\nu\} \Downarrow_{(\bigwedge_{k=1}^{n-1} \neg \widetilde{B}_k, R(\widetilde{B}_i) \wedge \bigwedge_{k=1}^{i-1} \neg R(\widetilde{B}_k))} \sqcup \\ \bigsqcup\{\langle \eta_n, \tilde{\eta}_j\rangle_{\mathbf{u}} \twoheadrightarrow (r_n, \tilde{r}_j) \mid i \in \{1, \ldots, n-1\},\ \langle \eta_i, \tilde{\eta}_i\rangle_{\mathbf{s}} \twoheadrightarrow (r_i, \tilde{r}_i) \in \mathcal{E}[\![S_i]\!]_\nu, \\ \langle \eta_n, \tilde{\eta}_n\rangle_{\mathbf{s}} \twoheadrightarrow (r_n, \tilde{r}_n) \in \mathcal{E}[\![S_n]\!]_\nu\} \Downarrow_{(\widetilde{B}_i \wedge \bigwedge_{k=1}^{i-1} \neg \widetilde{B}_k, \bigwedge_{k=1}^{n-1} \neg R(\widetilde{B}_k))}$$

Fig. 1. Semantics of a program expression.

corresponding arithmetic operation to the values of the operands. The new conditions are obtained as the combination of the conditions of the operands. The semantics of the expression *let* $\tilde{x} = \widetilde{A}$ *in* S updates the current environment by associating with variable $\tilde{x}$ the semantics of expression $\widetilde{A}$.

The semantics of the conditional *if* $\widetilde{B}$ *then* S_1 *else* S_2 uses an auxiliary operator $\Downarrow$.

Definition 3 (Condition propagation operator). *Given* $b \in \mathbb{B}$ *and* $\tilde{b} \in \widetilde{\mathbb{B}}$, $\langle \phi, \tilde{\phi}\rangle_t \twoheadrightarrow (r, \tilde{r}) \Downarrow_{(b, \tilde{b})} = \langle \phi \wedge b, \tilde{\phi} \wedge \tilde{b}\rangle_t \twoheadrightarrow (r, \tilde{r})$ *if* $\phi \wedge b \wedge \tilde{\phi} \wedge \tilde{b} \not\Rightarrow false$, *otherwise it*

is undefined. The definition of $\Downarrow$ *naturally extends to sets of conditional tuples: given* $C \in \mathbb{C}$, $C \Downarrow_{(b,\tilde{b})} = \bigcup_{c \in C} c \Downarrow_{(b,\tilde{b})}$.

The semantics of S_1 and S_2 are enriched with the information about the fact that real and floating-point control flows match, i.e., both $\widetilde{B}$ and $R_{\mathbb{B}}(\widetilde{B})$ have the same value. In addition, new conditional tuples are built to model the unstable cases when real and floating-point control flows do not coincide and, therefore, real and floating-point computations diverge. For example, if $\widetilde{B}$ is satisfied but $R_{\mathbb{B}}(\widetilde{B})$ is not, the *then* branch is taken in the floating-point computation, but the *else* would have been taken in the real one. In this case, the real condition and its corresponding output are taken from the semantics of S_2, while the floating-point condition and its corresponding output are taken from the semantics of S_1. The condition $(\neg R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})$ is propagated in order to model that $\widetilde{B}$ holds but $R_{\mathbb{B}}(\widetilde{B})$ does not. The conditional tuples representing this case are marked with **u**.

Similarly, the semantics of an n-ary conditional is composed of stable and unstable cases. The stable cases are built from the semantics of all the program sub-expressions S_i by enriching them with the information stating that the correspondent guard and its real counter-part hold and all the previous guards and their real counterparts do not hold. All the unstable combinations are built by combining the real parts of the semantics of a program expression S_i and the floating-point contributions of a different program expression S_j. In addition, the operator $\Downarrow$ is used to propagate the information that the real guard of S_i and the floating-point guard of S_j hold, while the guards of the previous branches do not hold.

3 Program Transformation

In this section, a program transformation is proposed for detecting when round-off errors affect the evaluation of floating-point conditionals and for ensuring that when the floating-point control flow diverges from the real one a warning is issued. The proposed transformation takes into account round-off errors by abstracting the Boolean expressions in the guards of the original program. This is done by means of two Boolean abstractions $\beta^+, \beta^- : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{B}}$.

Given $\tilde{\phi} \in \widetilde{B}$, let $fv(\tilde{\phi})$ be the set of free variables in $\tilde{\phi}$. For all $\sigma : fv(\tilde{\phi}) \to \mathbb{R}$, $\tilde{\sigma} : fv(\tilde{\phi}) \to \mathbb{F}$, and $\tilde{x} \in fv(\tilde{\phi})$ such that $R(\tilde{\sigma}(\tilde{x})) = \sigma(\chi_r(\tilde{x}))$, β^+ and β^- satisfy the following properties.

1. $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \beta^+(\tilde{\phi})) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\phi}) \wedge eval_{\mathbb{B}}(\sigma, R(\tilde{\phi}))$.
2. $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \beta^-(\tilde{\phi})) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \neg\tilde{\phi}) \wedge eval_{\mathbb{B}}(\sigma, \neg R(\tilde{\phi}))$.

Property 1 states that for all floating-point Boolean expressions $\tilde{\phi}$, $\beta^+(\tilde{\phi})$ implies both $\tilde{\phi}$ and its real counterpart. Symmetrically, Property 2 ensures that $\beta^-(\tilde{\phi})$ implies both the negation of $\tilde{\phi}$ and the negation of its real counterpart.

Example 1. The Boolean abstractions β^+ and β^- can be instantiated as follows for conjunctions and disjunction of sign tests. Properties 1 and 2 are formally

proven in PVS to hold for the following definitions of β^+ and β^-. Let $\widetilde{expr} \in \widetilde{\mathbb{A}}$ and $\epsilon \in \mathbb{F}$ such that $|\widetilde{expr} - R_{\mathbb{A}}(\widetilde{expr})| \leq \epsilon$.

$$\beta^+(\widetilde{expr} \leq 0) = \widetilde{expr} \leq -\epsilon \qquad \beta^-(\widetilde{expr} \leq 0) = \widetilde{expr} > \epsilon$$
$$\beta^+(\widetilde{expr} \geq 0) = \widetilde{expr} \geq \epsilon \qquad \beta^-(\widetilde{expr} \geq 0) = \widetilde{expr} < -\epsilon$$
$$\beta^+(\widetilde{expr} < 0) = \widetilde{expr} < -\epsilon \qquad \beta^-(\widetilde{expr} < 0) = \widetilde{expr} \geq \epsilon$$
$$\beta^+(\widetilde{expr} > 0) = \widetilde{expr} > \epsilon \qquad \beta^-(\widetilde{expr} > 0) = \widetilde{expr} \leq -\epsilon$$
$$\beta^+(\tilde{\phi}_1 \wedge \tilde{\phi}_2) = \beta^+(\tilde{\phi}_1) \wedge \beta^+(\tilde{\phi}_2) \qquad \beta^-(\tilde{\phi}_1 \wedge \tilde{\phi}_2) = \beta^-(\tilde{\phi}_1) \vee \beta^-(\tilde{\phi}_2)$$
$$\beta^+(\tilde{\phi}_1 \vee \tilde{\phi}_2) = \beta^+(\tilde{\phi}_1) \vee \beta^+(\tilde{\phi}_2) \qquad \beta^-(\tilde{\phi}_1 \vee \tilde{\phi}_2) = \beta^-(\tilde{\phi}_1) \wedge \beta^-(\tilde{\phi}_2)$$

The abstractions performed for sign tests are not correct for generic inequalities of the form $a \leq b$. In this case, to compensate for the round-off errors of both expressions, additional floating-point operations must be performed. Thus, the round-off error generated by such operations needs to be considered as well to obtain a sound approximation. The naive application of this strategy leads to a non-terminating transformation. The design of an effective approximation for these generic inequalities is left as future work.

The program transformation is defined as follows.

Definition 4 (Program Transformation). *Let $\tilde{f}(\tilde{x}_1, \ldots, \tilde{x}_n) = S \in \mathbb{P}$ be a floating-point program that does not contain any warning statements, the transformed program is defined as $\tilde{f}(\tilde{x}_1, \ldots, \tilde{x}_n) = \tau(S)$ where τ is defined as follows.*

$\tau(\tilde{A}) = \tilde{A}$

$\tau(\mathit{if}\ \tilde{\phi}\ \mathit{then}\ S_1\ \mathit{else}\ S_2) =$
$\quad \mathit{if}\ \beta^+(\tilde{\phi})\ \mathit{then}\ \tau(S_1)\ \mathit{elseif}\ \beta^-(\tilde{\phi})\ \mathit{then}\ \tau(S_2)\ \mathit{else\ warning}$

$\tau(\mathit{if}\ \tilde{\phi}_1\ \mathit{then}\ S_1\ [\mathit{elsif}\ \tilde{\phi}_i\ \mathit{then}\ S_i]_{i=2}^{n-1}\ \mathit{else}\ S_n) =$
$\quad \mathit{if}\ \beta^+(\tilde{\phi}_1)\ \mathit{then}\ \tau(S_1)\ [\mathit{elsif}\ \beta^+(\tilde{\phi}_i) \wedge \bigwedge_{j=1}^{i-1} \beta^-(\tilde{\phi}_j)\ \mathit{then}\ \tau(S_i)]_{i=2}^{n-1}$
$\quad \mathit{elsif}\ \bigwedge_{j=1}^{n-1} \beta^-(\tilde{\phi}_j)\ \mathit{then}\ \tau(S_n)$
$\quad \mathit{else\ warning}$

$\tau(\mathit{let}\ \tilde{x} = \tilde{A}\ \mathit{in}\ S) = \mathit{let}\ \tilde{x} = \tilde{A}\ \mathit{in}\ \tau(S)$

In the case of the binary conditional statement, the *then* branch of the transformed program is taken when $\beta^+(\tilde{\phi})$ is satisfied. By Property 1, this means that in the original program both $\tilde{\phi}$ and $R(\tilde{\phi})$ hold and, thus, the *then* branch is taken in both real and floating-point control flows. Similarly, the *else* branch of the transformed program is taken when $\beta^-(\tilde{\phi})$ holds. This means, by Property 2, that in the original program the else branch is taken in both real and floating-point control flows. In the case real and floating-flows diverge, neither $\beta^+(\tilde{\phi})$ nor $\beta^-(\tilde{\phi})$ is satisfied and a warning is returned.

In the case of the n-ary conditional statements, the guard $\tilde{\phi}_i$ of the i-th branch is replaced by the conjunction of $\beta^+(\tilde{\phi}_i)$ and $\beta^-(\tilde{\phi}_j)$ for all the previous branches $j < i$. By properties 1 and 2, it follows that the transformed program takes the i-th branch only when the same branch is taken in both real and floating-point control flows of the original program. Additionally, a warning is issued by the transformed program when real and floating-point control flows of the original program differ.

The following theorem states the correctness of the program transformation τ. If the transformed program $\tau(P)$ returns an output $\tilde{r}$ different from *warning*, then the original program follows a stable path and returns the floating-point output $\tilde{r}$. Furthermore, in the case the original program presents an unstable behavior, the transformed program returns *warning*.

Theorem 1 (Program Transformation Correctness). *Given* $\tilde{f}(\tilde{x}_1,\ldots,\tilde{x}_n) = S \in \mathbb{P}$, $\sigma : \{\chi_r(\tilde{x}_1)\ldots\chi_r(\tilde{x}_n)\} \to \mathbb{R}$, *and* $\tilde{\sigma} : \{\tilde{x}_1\ldots\tilde{x}_n\} \to \mathbb{F}$, *such that for all* $i \in \{1,\ldots,n\}$, $R(\tilde{\sigma}(\tilde{x}_i)) = \sigma(\tilde{x}_i)$*:*

1. *for all* $\langle \eta', \tilde{\eta}' \rangle_{t'} \twoheadrightarrow (r', \tilde{r}') \in \mathcal{E}[\![\tau(S)]\!]_{\perp_{Env}}$ *such that* $\tilde{r} \neq \perp_{\mathbf{u}}$, *there exists* $\langle \eta, \tilde{\eta} \rangle_{\mathbf{s}} \twoheadrightarrow (r, \tilde{r}) \in \mathcal{E}[\![S]\!]_{\perp_{Env}}$ *such that* $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\eta}') \Rightarrow eval_{\mathbb{B}}(\sigma, \eta) \wedge \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\eta})$ *and* $\tilde{r} = \tilde{r}'$*;*
2. *for all* $\langle \eta, \tilde{\eta} \rangle_{\mathbf{u}} \twoheadrightarrow (r, \tilde{r}) \in \mathcal{E}[\![S]\!]_{\perp_{Env}}$, *there exists* $\langle \eta', \tilde{\eta}' \rangle_{t'} \twoheadrightarrow (r', \perp_{\mathbf{u}}) \in \mathcal{E}[\![\tau(S)]\!]_{\perp_{Env}}$ *such that* $eval_{\mathbb{B}}(\sigma, \eta) \wedge \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\eta}) \Rightarrow \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \tilde{\eta}')$.

The program transformation defined in Definition 4 has been formalized and Theorem 1 has been proven correct in PVS.[3]

It is important to remark that the intended semantics of the floating-point transformed program is the real-valued semantics of the original one, i.e., the real-valued semantics of the transformed program is irrelevant. Therefore, even if the transformed program presents unstable tests, Theorem 1 ensures that its floating-point control flow preserves the control flow of stable tests in the original program.

Example 2. Consider the program *eps_line*, which is part of the ACCoRD conflict detection and resolution algorithm [11]. This function is used to compute an implicitly coordinated horizontal resolution direction for the aircraft involved in a pair-wise conflict.

$$eps\_line(\tilde{v}_x, \tilde{v}_y, \tilde{s}_x, \tilde{s}_y) = \textit{if } \widetilde{expr} > 0 \textit{ then } -1 \quad \textit{elsif } \widetilde{expr} < 0 \textit{ then } 1 \quad \textit{else } 0,$$

where $\widetilde{expr} = (\tilde{s}_x * \tilde{v}_y) - (\tilde{s}_y * \tilde{v}_x)$ and $\tilde{v}_x, \tilde{v}_y, \tilde{s}_x, \tilde{s}_y$ are floating-point variables. For example, if the values of such variables are assumed to lie in the range $[-100, 100]$, the tool PRECiSA [13,21] can be used to compute the round-off error estimation $\epsilon = 6.4801497501321145 \times 10^{-12}$ for $\widetilde{expr}$. PRECiSA is a tool that over-approximates the round-off error of floating-point programs. It is fully automatic and generates PVS proof certificates that guarantee the correctness of the error estimations with respect to the floating-point IEEE-754 standard. The

[3] This formalization is available at https://shemesh.larc.nasa.gov/fm/PRECiSA.

following program is obtained by using the transformation τ with the Boolean approximations of Example 1.

$$\tau(\mathit{eps\_line}(\tilde{v}_x, \tilde{v}_y, \tilde{s}_x, \tilde{s}_y)) = \mathit{if}\ \widetilde{\mathit{expr}} > \epsilon\ \mathit{then}\ -1\ \ \mathit{elsif}\ \widetilde{\mathit{expr}} < -\epsilon\ \mathit{then}\ 1 \\ \mathit{elsif}\ \widetilde{\mathit{expr}} \geq \epsilon \wedge \widetilde{\mathit{expr}} \leq -\epsilon\ \mathit{then}\ 0\ \ \mathit{else\ warning}$$

The condition $\widetilde{\mathit{expr}} \geq \epsilon \wedge \widetilde{\mathit{expr}} \leq -\epsilon$ never holds since ϵ is a positive number. Therefore, the transformed program never returns 0. Indeed, when $\widetilde{\mathit{expr}}$ is close to 0, the test is unstable. The transformed program detects these unstable cases and returns a warning.

4 Case Study: PolyCARP algorithm

PolyCARP[4] (Algorithms for Computations with Polygons) [14,15] is a suite of algorithms for geo-containment applications. One of the main applications of PolyCARP is to provide geofencing capabilities to unmanned aerial systems (UAS), i.e., detecting whether a UAS is inside or outside a given geographical region, which is modeled using a 2D polygon with a minimum and a maximum altitude. Another application of PolyCARP is the detection of weather cells, modeled as moving polygons, along an aircraft trajectory.

A core piece of logic in PolyCARP is the polygon containment algorithm, i.e., the algorithm that checks whether or not a point lies in the interior of a polygon. Algorithms for polygon containment have to be carefully implemented since numerical errors may lead to wrong answers, even in cases where the point is far from the boundaries of the polygon. PolyCARP uses several techniques to detect if a point is contained in a polygon. One of these techniques relies on the computation of the *winding number*. This number corresponds to the number of times the polygon winds around p.

Consider two consecutive vertices v and v' of the polygon in the Cartesian plane with the point p as the origin. The function *winding_number_edge* checks in which quadrants v and v' are located and counts how many axes are crossed by the edge (v, v'). If v and v' belong to the same quadrant, the contribution of the edge to the winding number is 0 since no axis is crossed. If v and v' lie in adjacent quadrants, the contribution is 1 (respectively -1) if moving from v to v' along the edge is in counterclockwise (respectively clockwise) direction. In the case v and v' are in opposite quadrants, the determinant is computed for checking the direction of the edge. If it is counterclockwise the contribution is 2, otherwise it is -2. The winding number is obtained as the sum of the contributions of all the edges of the polygon. If the result is 0 or 4, the point is inside the polygon, otherwise, it is outside.

$$\mathit{winding\_number\_edge}(v_x, v_y, v'_x, v'_y, p_x, p_y) = \\ \mathit{let}\ t_x = v_x - p_x\ \mathit{in}\ \mathit{let}\ t_y = v_y - p_y\ \mathit{in}\ \mathit{let}\ n_x = v'_x - p_x\ \mathit{in}\ \mathit{let}\ n_y = v'_y - p_y\ \mathit{in}$$

[4] PolyCARP is available at https://github.com/nasa/polycarp.

$$
\begin{array}{l}
\quad \mathit{if}\ \mathit{same\_quad}\ \mathit{then}\ 0 \\
\quad \mathit{elsif}\ \ \mathit{adj\_quad\_ctrclock}\ \mathit{then}\ 1 \\
\quad \mathit{elsif}\ \ \mathit{adj\_quad\_clock}\ \mathit{then}\ -1 \\
\quad \mathit{elsif}\ \ \mathit{det\_pos}\ \mathit{then}\ 2 \\
\quad \mathit{else}\ \ -2
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{same\_quad} = \\
\quad (t_x \geq 0 \wedge t_y \geq 0 \wedge n_x \geq 0 \wedge n_y \geq 0) \vee (t_x \leq 0 \wedge t_y \geq 0 \wedge n_x \leq 0 \wedge n_y \geq 0) \vee \\
\quad (t_x \geq 0 \wedge t_y \leq 0 \wedge n_x \geq 0 \wedge n_y \leq 0) \vee (t_x \leq 0 \wedge t_y \leq 0 \wedge n_x \leq 0 \wedge n_y \leq 0) \\
\mathit{adj\_quad\_ctrclock} = \\
\quad (t_x \geq 0 \wedge t_y \leq 0 \wedge n_x \geq 0 \wedge n_y \geq 0) \vee (t_x \geq 0 \wedge t_y \geq 0 \wedge n_x \leq 0 \wedge n_y \geq 0) \vee \\
\quad (t_x \leq 0 \wedge t_y \geq 0 \wedge n_x \leq 0 \wedge n_y \leq 0) \vee (t_x \leq 0 \wedge t_y \leq 0 \wedge n_x \geq 0 \wedge n_y \leq 0), \\
\mathit{adj\_quad\_clock} = \\
\quad (t_x \geq 0 \wedge t_y \geq 0 \wedge n_x \geq 0 \wedge n_y \leq 0) \vee (t_x \leq 0 \wedge t_y \geq 0 \wedge n_x \leq 0 \wedge n_y \geq 0) \vee \\
\quad (t_x \leq 0 \wedge t_y \leq 0 \wedge n_x \leq 0 \wedge n_y \geq 0) \vee (t_x \geq 0 \wedge t_y \leq 0 \wedge n_x \leq 0 \wedge n_y \leq 0), \\
\mathit{det\_pos} = (n_x - t_x) * t_y - (n_y - t_y) * t_x \leq 0.
\end{array}
$$

The function *winding_number_edge* has been verified in PVS using real arithmetic. However, due to floating-point errors, taking the incorrect branch for one of the edges in the computation of the winding number may result in an incorrect conclusion about the position of the point with respect to the polygon. In order to overcome this problem, the transformation τ of Definition 4 is applied to the function *winding_number_edge* resulting in the following function. Given initial bounds for the input variables, PRECiSA [13,21] can be used to compute the round-off error estimations for n_x, n_y, t_x, t_y and the determinant, which are denoted ϵ_{t_x}, ϵ_{t_y}, ϵ_{n_x}, ϵ_{n_y}, and ϵ_{det}, respectively.

$$
\begin{array}{l}
\tau(\mathit{winding\_number\_edge}(v_x, v_y, v'_x, v'_y, p_x, p_y)) = \\
\quad \mathit{let}\ t_x = v_x - p_x\ \mathit{in}\ \mathit{let}\ t_y = v_y - p_y\ \mathit{in}\ \mathit{let}\ n_x = v'_x - p_x\ \mathit{in}\ \mathit{let}\ n_y = v'_y - p_y\ \mathit{in} \\
\quad\quad \mathit{if}\ \ \mathit{same\_quad}^{\beta}\ \mathit{then}\ 0 \\
\quad\quad \mathit{elsif}\ \ \mathit{adj\_quad\_ctrclock}^{\beta}\ \mathit{then}\ 1 \\
\quad\quad \mathit{elsif}\ \ \mathit{adj\_quad\_clock}^{\beta}\ \mathit{then}\ -1 \\
\quad\quad \mathit{elsif}\ \ \mathit{det\_pos}^{\beta}\ \mathit{then}\ 2 \\
\quad\quad \mathit{elsif}\ \ \mathit{original\_else}^{\beta}\ \mathit{else}\ -2 \\
\quad\quad \mathit{else}\ \ \mathit{warning},
\end{array}
$$

where

$$
\begin{aligned}
same\_quad^{\beta} = \beta^{+}(same\_quad) = {} & (t_x \geq \epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \geq \epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \geq \epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \geq \epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}),
\end{aligned}
$$

$$
adj\_quad\_ctrclock^{\beta} = \beta^{+}(adj\_quad\_counterclock) \wedge \beta^{-}(same\_quad),
$$

$$
\begin{aligned}
adj\_quad\_clock^{\beta} = {} & \beta^{+}(adj\_quad\_clock) \wedge \beta^{-}(adj\_quad\_ctrclock) \wedge \\
& \beta^{-}(same\_quad),
\end{aligned}
$$

$$
\begin{aligned}
det\_pos^{\beta} = {} & (n_x - t_x) * t_y - (n_y - t_y) * t_x \leq -\epsilon_{det} \wedge \beta^{-}(adj\_quad\_clock) \wedge \\
& \beta^{-}(adj\_quad\_ctrclock) \wedge \beta^{-}(same\_quad),
\end{aligned}
$$

$$
\begin{aligned}
original\_else^{\beta} = {} & (n_x - t_x) * t_y - (n_y - t_y) * t_x > \epsilon_{det} \wedge \beta^{-}(adj\_quad\_clock) \wedge \\
& \beta^{-}(adj\_quad\_ctrclock) \wedge \beta^{-}(same\_quad),
\end{aligned}
$$

$$
\begin{aligned}
\beta^{-}(same\_quad) = {} & (t_x < -\epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x < -\epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x < -\epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x < -\epsilon_{n_x} \vee n_y > \epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y > \epsilon_{n_y}),
\end{aligned}
$$

$$
\begin{aligned}
\beta^{+}(adj\_quad\_ctrclock) = {} & (t_x \geq \epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \geq \epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \geq \epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \geq \epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}),
\end{aligned}
$$

$$
\begin{aligned}
\beta^{-}(adj\_quad\_ctrclock) = {} & (t_x < -\epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x < -\epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x < -\epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y > \epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x < -\epsilon_{n_x} \vee n_y > \epsilon_{n_y}),
\end{aligned}
$$

$$
\begin{aligned}
\beta^{+}(adj\_quad\_clock) = {} & (t_x \geq \epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \geq \epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \geq \epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \leq -\epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \geq \epsilon_{n_y}) \vee \\
& (t_x \geq \epsilon_{t_x} \wedge t_y \leq -\epsilon_{t_y} \wedge n_x \leq -\epsilon_{n_x} \wedge n_y \leq -\epsilon_{n_y}),
\end{aligned}
$$

$$
\begin{aligned}
\beta^{-}(adj\_quad\_clock) = {} & (t_x < -\epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x < -\epsilon_{n_x} \vee n_y > \epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y < -\epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x > \epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y < -\epsilon_{n_y}) \wedge \\
& (t_x < -\epsilon_{t_x} \vee t_y > \epsilon_{t_y} \vee n_x > \epsilon_{n_x} \vee n_y > \epsilon_{n_y}).
\end{aligned}
$$

Consider a polygonal geofence and a set of randomly generated points in the square that circumscribes it. For each edge of the polygon and each generated point, the original function *winding_number_edge* is executed by using both exact real arithmetic and double-precision floating-point arithmetic. Additionally, the transformed function $\tau(winding\_number\_edge)$ is executed with double-precision floating-point arithmetic. For these randomly generated points, both the original and the transformed program return the same result. However, the closer the generated point is to the border of the polygon, the more likely is for the original program to take an unstable path. By considering a set of randomly generated points very close to the edges of the polygon, the transformed program always returns a warning, showing that these are the cases for which the floating-point computation may diverge from the real one. Since an over-approximation of the round-off error is used, not all the generated warnings reflect an actual problem. In fact, false warnings occur when the compensated error computed by the abstraction is larger than the round-off error that actually occurs in the computation. The amount of false warnings converges to the 50% of the number of total warnings as the distance to the edge decreases.

5 Related Work

Recently, several program transformations have been proposed with the aim of improving accuracy and efficiency of floating-point computations. It is possible to distinguish two kinds of approaches: precision allocation tools and program optimization ones. Precision allocation (or tuning) tools aim at selecting the lowest floating-point precision that is necessary to achieve a desired accuracy. This approach avoids using more precision than needed and improves the performance of the program. Rosa [8,9] uses a compilation algorithm that, from an ideal real-valued implementation, produces a finite-precision version (if it exists) that is guaranteed to meet the desired overall precision. Rosa soundly deals with unstable tests and with bounded loops. Similarly, FPTuner [3] implements a rigorous approach to precision allocation of mixed-precision arithmetic expressions. Precimonius [18] is a dynamic tool able to identify parts of a program that can be performed at a lower precision. It generates a transformed program where each floating-point variable is typed to the lowest precision necessary to meet a set of given accuracy and performance constraints. Hence, the transformed program uses variables of lower precision and performs better than the original program.

Program optimization tools aim at improving the accuracy of floating-point programs by rewriting arithmetic expressions in equivalent ones with a lower accumulated round-off error. Herbie [17] is a tool that automatically improves the accuracy of floating-point programs though a heuristic search. Herbie detects the expressions where rounding-errors occur and it applies a series of rewriting and simplification rules. It generates a set of transformed programs that are equivalent to the original one but potentially more accurate. The rewriting and simplification process is then applied recursively to the generated transformed

programs until the most accurate program is obtained. CoHD [19] is a source-to-source transformer for C code that automatically compensates for the round-off errors of some basic floating-point operations. SyHD [20] is a C code optimizer that explores a set of programs generated by CoDH and selects the one with the best accuracy and computation-time trade-off. The tool Sardana [12], given a Lustre [2] program, produces a set of equivalent programs with simplified arithmetic expressions. Then, it selects the ones for which a better accuracy bound can be proved. Salsa [4] combines Sardana with techniques for intra-procedure [5] and inter-procedure [6,7] program transformation in order to improve the accuracy of a target variable in larger pieces of code containing assignments and control structures.

To the best of the authors' knowledge, the program transformation proposed in this work is the only approach that addresses the problem of conditional instability for floating-point programs.

6 Conclusion

This paper presents a formally verified program transformation to detect instability in floating-point programs. The transformed program is guaranteed to return a warning when real and floating-point flows may diverge. Otherwise, it behaves as the original program when real and floating-point control flows coincide. The proposed approach is parametric with respect to two Boolean expression abstractions that return more restrictive Boolean conditions using an over-approximation of the round-off error occurring in the guard. These abstractions cause a loss of precision since the guards occurring in the transformed program are more restrictive and, therefore, some stable original traces may be lost in the transformed program. This leads to the possibility of having false instability warnings. However, it is ensured that all the unstable paths of the original program are detected.

This transformation has been formalized and formally proven correct in the interactive theorem prover PVS. The PVS tool PVSio can be used to execute the program transformation. However, a full integration with PRECiSA is the missing step to compute the round-off error approximations and to make the presented approach fully automatic.

The program transformation presented in this paper is the first step towards the much broader goal of improving the quality and reliability of floating-point programs. Future work includes the extension of the formalization to a more expressive language where conditionals are allowed inside Boolean expressions and function calls and loops are supported. This extension is not straightforward since it involves several changes in the formalization. In fact, in such setting, the evaluation of the expressions in the guards can also present unstable behaviors. Additionally, an extensive experimental evaluation is needed in order to assess the quality of the approach and its applicability to real-world applications. Another interesting future direction is the integration of the proposed approach with tools such as Salsa [4] and Herbie [17]. This integration will improve the

accuracy of the mathematical expressions used inside a program and, at the same time, prevent unstable tests that may cause unexpected behaviors.

References

1. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS. Tech. Rep. CR-2006-214298, NASA (2006)
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: Conference Record of the 14th ACM Symposium on Principles of Programming Languages, POPL 1987. pp. 178–188. ACM (1987)
3. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017. pp. 300–315. ACM (2017)
4. Damouche, N., Martel, M.: Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. 6th Workshop on Automated Formal Methods, AFM 2017 (2017)
5. Damouche, N., Martel, M., Chapoutot, A.: Optimizing the accuracy of a rocket trajectory simulation by program transformation. In: Proceedings of the 12th ACM International Conference on Computing Frontiers (CF'15). pp. 40:1–40:2. ACM (2015)
6. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. International Journal on Software Tools for Technology Transfer 19(4), 427–448 (2017)
7. Damouche, N., Martel, M., Chapoutot, A.: Numerical accuracy improvement by interprocedural program transformation. In: Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2017. pp. 1–10. ACM (2017)
8. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014. pp. 235–248. ACM (2014)
9. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Transactions on Programming Languages and Systems 39(2), 8:1–8:28 (2017)
10. Daumas, M., Rideau, L., Théry, L.: A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics. pp. 169–184. Springer Berlin Heidelberg (2001)
11. Dowek, G., Muñoz, C., Carreño, V.: Provably safe coordinated strategy for distributed conflict resolution. In: Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047 (2005)
12. Ioualalen, A., Martel, M.: Synthesizing accurate floating-point formulas. In: 24th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2013. pp. 113–116. IEEE Computer Society (2013)
13. Moscato, M.M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Proceedings of the 36th International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017. Springer (2017)
14. Narkawicz, A., Hagen, G.: Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance. In: Proceedings of the AIAA Aviation Conference (2016)

15. Narkawicz, A., Muñoz, C., Dutle, A.: The MINERVA software development process. In: 6th Workshop on Automated Formal Methods, AFM 2017 (2017)
16. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. vol. 607, pp. 748–752. Springer (1992)
17. Panchekha, P., Sanchez-Stern, A., Wilcox, J., Z., T.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015. pp. 1–11. ACM (2015)
18. Rubio-González, C., Nguyen, C., Nguyen, H., Demmel, J., Kahan, W., Sen, K., Bailey, D., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13. p. 27. ACM (2013)
19. Thévenoux, L., Langlois, P., Martel, M.: Automatic source-to-source error compensation of floating-point programs. In: 18th IEEE International Conference on Computational Science and Engineering, CSE 2015. pp. 9–16. IEEE Computer Society (2015)
20. Thévenoux, L., Langlois, P., Martel, M.: Automatic source-to-source error compensation of floating-point programs: code synthesis to optimize accuracy and time. Concurrency and Computation: Practice and Experience 29(7) (2017)
21. Titolo, L., Feliú, M., Moscato, M., Muñoz, C.: An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2018. vol. 10747, pp. 516–537. Springer (2018)