F-IKOS: An Abstract Interpretation-based Static Analyzer for Fortran Programs

Sheng Zou[1,2], Liqian Chen[1,2,*], Guangsheng Fan[1,2], Renjie Huang[1,2] and Banghu Yin[3]

[1]*College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*
[2]*State Key Laboratory of Complex & Critical Software Environment, Changsha 410073, China*
[3]*College of Systems Engineering, National University of Defense Technology, Changsha 410073, China*

Abstract
The Fortran programming language is widely utilized in numerical computation and scientific computing. Fortran programs are prone to potential runtime errors related to numerical properties due to the large number of numerical operations. In this paper, we present `F-IKOS`, an abstract interpretation-based static analyzer for Fortran programs on top of `IKOS`, which soundly handles floating-point types in Fortran programs. Firstly, we translate Fortran programs to LLVM IR using compiler front-end `Flang`. After that, we extend `IKOS` to support sound floating-point analysis and then employ it to analyze the translated LLVM IR. Particularly, when analyzing floating-point types in programs, we first abstract floating-point expressions into real-number expressions with interval coefficients, and then linearize these expressions into real-number expressions with scalar coefficients. These linear expressions are subsequently handled by abstract domains originally designed for real-number types to produce sound analysis results. We have conducted experiments on representative Fortran programs to show the efficiency and effectiveness of `F-IKOS`. The experimental results are encouraging: `F-IKOS` soundly analyzes runtime errors in complex programs, outperforming other analyzers.

Keywords
Fortran, Static Analysis, Abstract Interpretation, Floating-point Program Analysis

1. Introduction

The Fortran programming language is one of the oldest high-level programming languages and one of the first to be widely adopted for scientific computing. Additionally, several numerical computation libraries, including BLAS[17], which are developed in Fortran, have significantly contributed to the widespread use of Fortran in domains such as numerical computing and high-performance computing. Compared with mainstream high-level programming languages such as C++ and Java, Fortran possesses a distinctive set of features, such as powerful array manipulation and abundant intrinsic functions for numerical computation.

However, Fortran programs are prone to potential runtime errors related to numerical aspects such as division-by-zero and arithmetic overflow due to the large number of numerical operations. Researchers have dedicated efforts to the analysis and verification of Fortran programs. Previous research on the analysis of Fortran programs can be mainly classified into three categories: Firstly, approaches such as `f2c` [3] and `FABLE` [11] convert Fortran programs to other high-level language programs and verify them using verifiers over the converted high-level language programs. Secondly, approaches such as `SMACK` [7] and `CIVL` [8] translate Fortran programs to Intermediate Representation (IR), which is then verified using verifiers for IR, mainly based on model checking. Additionally, some static analyzers such as `FORTRAN-lint` [13], `ftnchek` [12] and `Coverity` [14] detect certain generic defects, such as dead code and variable usage problems, using pre-defined patterns.

Fortran programs are characterized by the extensive use of floating-point operations, which are crucial for achieving high precision and scale in numerical computing tasks, such as solving differential equations. Every floating-point type has a specific finite precision and cannot represent all real numbers exactly, leading to inherent and pervasive rounding errors in Fortran programs. If rounding errors are not accounted for during the analysis of Fortran programs, the analysis results may be unsound. However, the aforementioned approaches neglect to account for rounding errors during the analysis of Fortran programs.

In this paper, we propose an approach to abstract floating-point operations soundly in Fortran programs, which considers rounding errors when analyzing programs. Our approach abstracts floating-point expressions in programs into expressions with interval coefficients under real-number semantics and then linearizes them into expressions with scalar coefficients. This abstraction aims to eliminate rounding errors by floating-point operations during program analysis, which enables analyzers to perform sound analysis under the abstract interpretation framework. We develop a Fortran program analyzer on top of the Inference Kernel for Open Static Analyzers (`IKOS` [10]) to implement the proposed approach, named `F-IKOS`. `F-IKOS` utilizes `Flang`[5] to translate Fortran programs into LLVM Intermediate Representation (LLVM IR) and then leverages `IKOS` to analyze the LLVM IR. The core of `F-IKOS` is `IKOS`, which is a static analyzer based on abstract interpretation. `IKOS` can use abstract domains from the Apron [6] numerical abstract domain library to analyze programs. However, it mainly detects runtime errors in machine integer types and cannot infer invariants on floating-point types [1]. In our implementation, we first extended `IKOS` to support floating-point types, and then applied the proposed approach to handle floating-point operations soundly. With these extensions, `F-IKOS` can analyze floating-point types in Fortran programs and obtain sound analysis results. We conducted experiments over benchmarks consisting of representative Fortran programs and the evaluation results demonstrate the efficiency, effectiveness, and utility of the analyzer `F-IKOS`. The main contributions of this work are as follows:

- We proposed an approach to soundly abstract floating-point operations in Fortran programs, accounting for rounding errors during program analy-

QuASoQ 2024: 12th International Workshop on Quantitative Approaches to Software Quality, December 03, 2024, Chongqing China
*Corresponding author.
zous@nudt.edu.cn (S. Zou); lqchen@nudt.edu.cn (L. Chen); guangshengfan@nudt.edu.cn (G. Fan); renjiehuang@nudt.edu.cn (R. Huang); bhyin@nudt.edu.cn (B. Yin)

sis.

- We developed F-IKOS, a static analyzer for Fortran programs, to implement the proposed approach. F-IKOS can soundly analyze floating-point types in Fortran programs.
- Evaluation shows that F-IKOS can handle the complex syntax of Fortran programs and produce sound analysis results, outperforming other relevant Fortran analyzers.

The rest of the paper is organized as follows. Section 2 describes background. Section 3 presents the overview of our analyzer F-IKOS. Section 4 presents the proposed abstraction of floating-point expressions. Section 5 presents our analyzer implementation together with experimental results. Section 6 discusses some related work and Section 7 concludes.

2. Background

2.1. The Fortran Programming Language

The Fortran programming language is a well-established high-level language with many syntax standards, such as Fortran 77 and Fortran 90. Fortran has similarities to other high-level programming languages. For example, Fortran includes common control structures such as conditional statements and looping control structures. However, Fortran programs often emphasize numerical computation tasks more than logical control functions. This is reflected by the fact that Fortran has a wealth of intrinsic functions, such as the trigonometric functions **sin**, **cos**, **asin**, and **acos**.

Besides, Fortran has a rich set of convenient array operations. Fortran supports flexible array boundaries, such as $integer :: v(-3 : 3)$, indicating that the index of array v ranges from -3 to 3. It also provides a range of intrinsic functions for arrays, including **sum** and **product**, as well as **matmul** and **dot_product** for computing matrix products and dot products. Fortran inherently supports multidimensional arrays and offers mechanisms for array slicing and reshaping. For instance, the **reshape** function facilitates altering the number of dimensions and the size of each dimension within an array. Due to these characteristics, Fortran is widely used in scientific and high-performance computing.

2.2. The Floating-point Representation

The floating-point representation adheres to the IEEE-754 standard. Many high-precision real numbers, cannot be exactly represented by floating-point numbers. The IEEE-754 standard provides four rounding modes: *nearest, zero,* $-\infty$, and $+\infty$, to approximate real numbers using floating-point numbers. This approximation can introduce rounding errors, which result in inexactness.

Due to rounding errors, mathematical properties followed by real number operations do not hold in floating-point operations. We illustrate it with an example. To distinguish between real and floating-point numbers and their respective operators, we employ $rnd(\cdot)$ to represent floating-point numbers in machines and denote real number operators using symbols $+, -, \times, /$, while denoting floating-point operators using $\oplus_{f,r}, \ominus_{f,r}, \otimes_{f,r}, \oslash_{f,r}$ where the subscripts f, r denote different precision and rounding modes. Real numbers like 0.1, 0.2, and 0.3 cannot be exactly represented by machines. Under real number semantics, the equation $0.1 + 0.2 = 0.3$ holds. But in machines, $rnd(0.1) \oplus_{f,r} rnd(0.2)$ is not equal to $rnd(0.3)$. Furthermore, the law of association and distribution is not always true in floating-point operations. E.g., it may happen that

$$(rnd(a) \oplus_{f,r} rnd(b)) \oplus_{f,r} rnd(c) \neq rnd(a) \oplus_{f,r} (rnd(b) \oplus_{f,r} rnd(c))$$

2.3. Rounding Model of IEEE-754

A simplified rounding model of the IEEE-754 standard follows the equation below:

$$rnd(x) = x \times (1 + e) + d$$

where $|e| \leq \epsilon$, $|d| \leq \delta$, and $e \times d = 0$. When x is a normalized number, it holds that $d = 0$, and when x is a denormalized number, it holds that $e = 0$. Here, ϵ denotes the maximum relative error for normalized numbers, and δ describes the maximum absolute error for denormalized numbers for specific precision of floating-point numbers.

2.4. Linearization

An *interval linear expression* is an expression where the coefficients may be intervals instead of scalars. For instance,

$$z = [b_1, c_1]x + [b_2, c_2]y$$

Interval linear expressions, in which the coefficients are intervals, can be abstracted into linear expressions with scalar coefficients. This process is *linearization*, and it is defined as follows:

Definition 1 (Linearization). An interval linear expression $\sum_i [a_i, b_i] \times x_i + [c, d]$ can be linearized into a linear expression $\sum_i e_i \times x_i + [c', d']$, where $e_i \in [a_i, b_i]$ and satisfying $\sum_i [a_i, b_i] \times x_i + [c, d] \subseteq \sum_i e_i \times x_i + [c', d']$ for all $x_i \in [\underline{x_i}, \overline{x_i}]$ where $\underline{x_i} \leq x_i \leq \overline{x_i}$.

3. Overview

In this section, we give an overview of our approach. We develop a static analyzer on top of IKOS [10], named F-IKOS, to perform sound analysis of floating-point types in Fortran programs. The architecture and workflow of F-IKOS are illustrated in Fig. 1, highlighting our modifications to enable the sound analysis of floating-point types in Fortran programs. Initially, F-IKOS takes Fortran programs as input and uses the parser Flang [5] to translate programs into LLVM IR. Subsequently, after optimization and processing, F-IKOS uses extended IKOS together with numerical abstract domains from Apron [6] to analyze the LLVM IR and obtain invariants of programs. Finally, potential runtime errors in programs are checked by utilizing these invariants.

4. Approach

Fortran programs involve numerous floating-point operations, and analyzing these programs within the abstract interpretation framework by simply treating floating-point expressions as real-number expressions may result in unsound results. This occurs because floating-point numbers in programs are rounded before being passed to abstract domains (e.g., Apron [6]). For example, within the polyhedra abstract domain, linear constraints between variables are collected

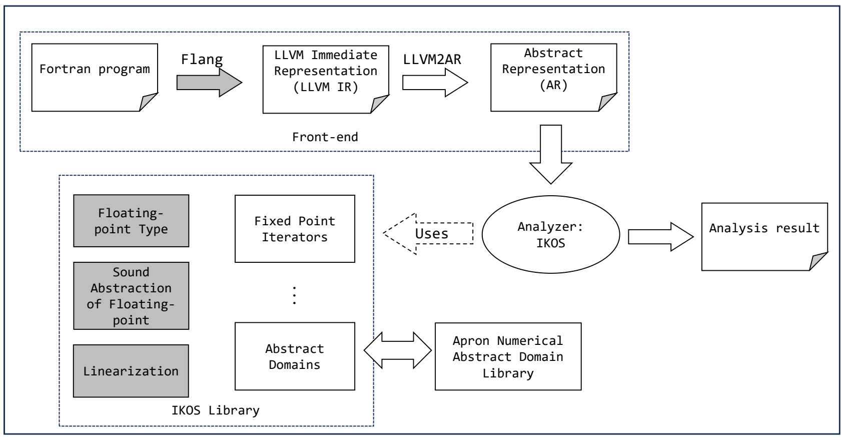

Figure 1: Overview of F-IKOS

and utilized to infer the program's invariants. Due to the presence of floating-point numbers and operators (e.g., $\oplus_{f,r}$, $\ominus_{f,r}$, $\otimes_{f,r}$, $\oslash_{f,r}$), these expressions are under floating-point semantics. Directly interpreting floating-point expressions as expressions under real-number semantics introduces unsoundness to the analysis. For instance, $(\mathrm{rnd}(x) \oplus_{f,r} \mathrm{rnd}(y))$ is not equivalent to $(x + y)$ under real-number semantics.

To address this challenge, Miné [19] proposes to over-approximately abstract floating-point expressions into real-number expressions. The approach includes the following three steps:

1. Abstract the deterministic semantics of floating-point expressions into non-deterministic semantics on real-number expressions.
2. Convert the non-deterministic semantics of real numbers into deterministic semantics for real numbers.
3. Analyze programs with real-number operations using abstract domains initially designed for programs with real-number types.

In this section, we present the following three steps: sound abstraction of floating-point expressions, linearization of interval linear expressions, and the analysis of programs using abstract domains.

4.1. Abstraction of Floating-point Expressions

Abstraction of floating-point numbers. Floating-point types in Fortran programs adhere to the IEEE-754 standard. Given the value of a floating-point number and its precision, using the rounding model outlined in Section 2.3, we can compute the relation of $rnd(x)$ and x (the interval range) as follows.

When x is a normalized number, the relation of x and $rnd(x)$ can be represented as:

$$rnd(x) = x \times [1 - \epsilon, 1 + \epsilon] \tag{1}$$

When x is a denormalized number (i.e., close to zero), the relation of x and $rnd(x)$ is given by:

$$rnd(x) = x + [-\delta, \delta] \tag{2}$$

where ϵ denotes the maximum relative error for normalized numbers, and δ represents the maximum absolute error for denormalized numbers for specific precision of x. Table 1 illustrates the values of ϵ and δ for various floating-point types. To enhance generality, we express the conversion between a floating-point number and its corresponding real interval range before rounding using Formula 3. This provides an over-approximation of x given by Formula 1 and 2 regardless of whether x is a normalized or denormalized number.

Precision	ϵ	δ
single (32 bits)	2^{-23}	2^{-149}
double (64 bits)	2^{-52}	2^{-1074}
quad (128 bits)	2^{-112}	2^{-16494}

Table 1
Related/Absolute rounding errors for various types

$$rnd(x) = x \times ([1 - \epsilon, 1 + \epsilon]) + [-\delta, \delta] \tag{3}$$

Abstraction of Expressions with Floating-Point Operators. Miné [19] proposes a method for abstracting expressions involving floating-point operators into real-number expressions. This approach captures rounding errors of floating-point arithmetic by over-approximating the behavior of floating-point operators using real-number semantics.

Assume that $rnd(x)$ and $rnd(y)$ are two floating-point expressions, with x and y representing the corresponding real-number expressions. Let a and b be real numbers, and let ϵ and δ denote the relative and absolute errors, respectively, which depend on the precision of the floating-point types.

Non-linear operators (such as $\otimes_{f,r}$ and $\oslash_{f,r}$) can be handled by applying the corresponding operator on intervals after "intervalizing" the arguments [19]. The operator $|\cdot|_\iota$ is used to "intervalize" the argument by a single interval [19]. When multiplying two linear forms that have not been reduced to an interval, the operator $|\cdot|_\iota$ can be applied to either argument. Similarly, operator $|\cdot|_\iota$ can be applied to the divisor to obtain a single interval before performing division.

The abstraction is described as follows [19]:

$$
\begin{aligned}
& x \oplus_{f,r} y = (x+y) \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta] \\
& x \ominus_{f,r} y = (x-y) \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta] \\
& x \otimes_{f,r} [a_0, b_0] = x \times [1-\epsilon, 1+\epsilon] \cdot [a_0, b_0] + [-\delta, \delta] \\
& [a_0, b_0] \otimes_{f,r} x = x \otimes_{f,r} [a_0, b_0] \\
& x \otimes_{f,r} y = x \times [1-\epsilon, 1+\epsilon] \cdot |y|_\iota + [-\delta, \delta] \qquad (4) \\
& \textit{or} \\
& x \otimes_{f,r} y = y \times [1-\epsilon, 1+\epsilon] \cdot |x|_\iota + [-\delta, \delta] \\
& x \oslash_{f,r} [a_0, b_0] = x \times [1-\epsilon, 1+\epsilon] / [a_0, b_0] + [-\delta, \delta] \\
& x \oslash_{f,r} y = x \times [1-\epsilon, 1+\epsilon] / |y|_\iota + [-\delta, \delta]
\end{aligned}
$$

Abstraction of Floating-point Expressions. In Fortran programs, variables (or constants) and operators in expressions are under floating-point semantics. We abstract expressions that involve floating-point variables and perform floating-point arithmetic into real-number expressions, which involve real-number variables and real-number operators.

The floating-point expression $rnd(x) \oplus_{f,r} rnd(y)$ can be abstracted into a real-number expression as follows:

$$rnd(x) \oplus_{f,r} rnd(y)$$

where

$$rnd(x) = x \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta]$$

$$rnd(y) = y \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta]$$

Substituting these into the expression, we get

$$(x \times ([1-\epsilon, 1+\epsilon]) + [-\delta, \delta]) \oplus_{f,r} (y \times ([1-\epsilon, 1+\epsilon]) + [-\delta, \delta]) \quad (5)$$

Since $x \oplus_{f,r} y = rnd(x+y)$, we convert Formula 5 as:

$$rnd(((x+y)([1-\epsilon, 1+\epsilon]) + 2 \times [-\delta, \delta]))$$

Thus, the expression $rnd(x) \oplus_{f,r} rnd(y)$ can be abstracted as:

$$(x+y)[(1-\epsilon)^2, (1+\epsilon)^2] + [-(3+2\epsilon)\delta, (3+2\epsilon)\delta] \quad (6)$$

By following this approach, the floating-point expression $rnd(x) \ominus_{f,r} rnd(y)$ can be abstracted into its corresponding real-number expression, as illustrated:

$$(x-y)[(1-\epsilon)^2, (1+\epsilon)^2] + [-(3+2\epsilon)\delta, (3+2\epsilon)\delta] \quad (7)$$

When multiplying two linear forms, the operator $|\cdot|_\iota$ can be applied to either argument to obtain an interval range of the argument. In this case, we apply $|\cdot|_\iota$ to the second argument. The floating-point expression $rnd(x) \otimes_{f,r} rnd(y)$ can be abstracted through the following steps:

Assume $[a,b] = |rnd(y)|_\iota \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta]$, then $rnd(x) \otimes_{f,r} rnd(y)$ can be converted into

$$rnd(x) \otimes_{f,r} [a,b] \quad (8)$$

Given that $rnd(x) = x \times [1-\epsilon, 1+\epsilon] + [-\delta, \delta]$ and $x \otimes_{f,r} [a_0, b_0] = rnd(x \times [a_0, b_0]) = x \times [1-\epsilon, 1+\epsilon] \cdot [a_0, b_0] + [-\delta, \delta]$, we can express Formula 8 as follows:

$$rnd(x \times [1-\epsilon, 1+\epsilon] \cdot [a,b] + [-\delta, \delta] \cdot [a,b])$$

Thus, the expression $rnd(x) \otimes_{f,r} rnd(y)$ can be abstracted as:

$$x \times [(1-\epsilon)^2, (1+\epsilon)^2] \cdot [a,b] + [-(1+\epsilon)\delta, (1+\epsilon)\delta] \cdot [a,b] + [-\delta, \delta] \quad (9)$$

Similarly, we apply $|\cdot|_\iota$ to the divisor to obtain a single interval before performing the division. The floating-point expression $rnd(x) \oslash_{f,r} rnd(y)$ can then be abstracted into its corresponding real-number expression, as illustrated below:

$$x \times [(1-\epsilon)^2, (1+\epsilon)^2] / [a,b] + [-(1+\epsilon)\delta, (1+\epsilon)\delta] / [a,b] + [-\delta, \delta] \quad (10)$$

The coefficients of variables within the real-number expressions obtained by abstraction are represented as real-number intervals. However, many abstract domains cannot process such interval-coefficient forms, as they only support linear expressions.

4.2. Linearization of Interval Linear Expressions

To enable existing numerical abstract domains (e.g., polyhedra abstract domain) to handle interval linear expressions, Miné [19] proposes to linearize these interval linear expressions to linear expressions. The core idea is as follows: Supposing variable x_i ranges over the interval $[\underline{x_i}, \overline{x_i}]$, an interval linear expression $\Sigma_i [a_i, b_i] \times x_i + [c, d]$ can be over-approximated by a linear expression of the form $\Sigma_i e_i \times x_i + [c', d']$. This approach converts real-number interval linear expressions into real-number linear expressions with scalar coefficients. The existing numerical abstraction domain initially designed for real-number semantics can directly analyze these expressions. We present our approach to linearize interval linear expressions.

Drawing inspiration from [2, 19], we define the linearization of interval linear expressions within real-number semantics as follows:

Definition 2 (Linearization Operator). Given an interval linear expression $\varphi : (\sum_i [a_i, b_i] \times x_i + [c, d])$, and letting $\mathbf{x} := [\underline{x}, \overline{x}]$ be the bounding box of variable x, the linearization operator is defined as

$$\zeta(\varphi, \mathbf{x}) \overset{def}{=} \sum_i e_i \times x_i + [c', d']$$

where e_i is any real number in the interval $[a_i, b_i]$, and $[c', d']$ denotes the resulting interval of $\sum_i [a_i - e_i, b_i - e_i] \times [\underline{x_i}, \overline{x_i}] + [c, d]$. Generally, we choose the midpoint of the interval $e_i = (a_i + b_i) \times 0.5$.

We provide the proof of the soundness of the linearization operator through the following reasoning:

$$
\begin{aligned}
& \sum_i [a_i, b_i] \times x_i + [c, d] \\
\iff & \sum_i (e_i + [a_i - e_i, b_i - e_i]) \times x_i + [c, d] \\
\iff & \sum_i e_i \times x_i + \sum_i [a_i - e_i, b_i - e_i] \times x_i + [c, d]
\end{aligned}
$$

can be over-approximated as

$$\sum_i e_i \times x_i + \left([a_i - e_i, b_i - e_i] \times [\underline{x_i}, \overline{x_i}]\right) + [c, d]$$

since it holds that $[a_i - e_i, b_i - e_i] \times x_i \subseteq [a_i - e_i, b_i - e_i] \times [\underline{x_i}, \overline{x_i}]$, where $a_i \le e_i \le b_i$ and $\underline{x_i} \le x_i \le \overline{x_i}$.

Note that following the same principle, an interval linear inequality $\sum_i [a_i, b_i] \times x_i + [c, d] \le 0$ can be also linearized into a linear inequality in the form of $\sum_i e_i \times x_i + [c', d'] \le 0$ in the sense of *weak solution* [23]. It means that a weak solution of an interval linear inequality $\sum_i [a_i, b_i] \times x_i + [c, d] \le 0$ will be a solution of $\sum_i e_i \times x_i + [c', d'] \le 0$ (but the reverse does not hold). This approach over-approximates interval linear expressions (or inequalities) up into linear expressions (or inequalities).

4.3. Analyze programs with existing abstract domains

Due to the sound handling of rounding errors inherent in floating-point operations through the above abstraction and linearization procedure, we can now obtain sound results using abstract domains initially designed for real-number semantics.

5. Implementation and Evaluation

5.1. Implementation

We have implemented a static analyzer for Fortran programs, named F-IKOS, with over 4K LOC of C++ code by extending IKOS. F-IKOS is endowed with the capability to perform sound analysis of floating-point types in Fortran programs.

5.2. Research Questions and Experimental Setup

To evaluate F-IKOS, we compare it with two most relevant Fortran program analyzers, SMACK [7] and CIVL [8]. Both SMACK and CIVL translate Fortran programs into IR for subsequent verification using IR-compatible verifiers, mainly based on model checking. They are designed to verify certain program properties but do not directly detect potential errors in Fortran programs.

We investigate the following three research questions across the analyzers:

- **RQ1**: How effective is F-IKOS in analyzing simple Fortran programs?
- **RQ2**: How well does F-IKOS handle complex features of Fortran programs?
- **RQ3**: How does F-IKOS perform when applied to real-world Fortran programs?

To address these questions, we conducted three experiments to evaluate the capabilities of F-IKOS. The benchmarks employed in our experiments are categorized into three distinct classes:

- 36 Fortran programs used by SMACK [7] and CIVL[8], which are used to evaluate F-IKOS's ability to handle simple syntaxs and accomplish verification tasks.
- 45 real-world Fortran programs extracted from open-source repositories [22, 21], encompassing various Fortran syntax standards.
- 10 artificially constructed programs, derived from the repository [20], designed to evaluate the capability of F-IKOS in detecting runtime errors associated with floating-point types.

All experiments were conducted on a PC running Ubuntu 20.04 (16GB Memory) in the Oracle VirtualBox 6.1.30 with a 3.3GHz Intel Core i9 CPU. The abstract domain used is Polka [6], which is an implementation of the Polyhedra abstract domain in Apron.

5.3. RQ1: Verifying simple Fortran programs

We analyzed 36 simple Fortran programs from the first benchmark [7, 8], excluding parallel and recursive program instances. The experimental results are presented in Table 2, where "F-IKOS Time (s)", "SMACK Time(s)" and "CIVL Time (s)" denote the execution time of F-IKOS, SMACK, and CIVL, respectively. In the last row of Table 2, the average execution time of programs is recorded.

Specifically, SMACK successfully verified only 19 out of 36 programs. In contrast, CIVL correctly verified all 36 programs, and F-IKOS successfully completed the analysis of the majority (30 out of 36). Further analysis of the unverified programs by F-IKOS reveals that most required disjunctive invariants for successful verification, are out of the expressiveness of the used polyhedra abstract domain, whereas CIVL, with the help of the SMT solver (i.e., Z3), can address them effectively.

Furthermore, a comparative analysis of the average execution time reveals that F-IKOS exhibits shorter analysis time, approximately 5% of SMACK's and 10% of CIVL's. The experimental results underscore F-IKOS's ability to achieve a delicate balance between analysis efficiency and effectiveness, demonstrating its strengths compared to existing state-of-the-art Fortran analyzers.

RQ-1 Answer: F-IKOS undergoes comparison with SMACK and CIVL for the analysis of 36 Fortran programs. It successfully verified 30 out of 36 programs, exhibiting a shorter average execution time compared with state-of-the-art approach and about 10% of CIVL's. The results highlight the efficiency and effectiveness of F-IKOS in analyzing simple Fortran programs.

5.4. RQ2: Handling complex feature operations

We analyzed 45 Fortran programs from the second benchmark. The programs in this benchmark utilize features and intrinsic functions in Fortran that have not been previously examined. Some programs exemplify common Fortran programming conventions, while others involve algorithmic implementations. The inclusion of integer and floating-point types, along with arrays, increases the complexity of analyzing the programs.

In addition to the Fortran 90 standard programs, some programs following Fortran 77, and Fortran 95 standards are also included in bold in the table. The evaluation results in the efficiency and effectiveness of the analyzers are shown in Table 3. "F-IKOS (s)" and "F-IKOS FP" illustrate the Execution Time and False Positives (FP) of F-IKOS, respectively.

The experimental results demonstrate that both SMACK and CIVL to analyze the 45 programs in benchmark and find that SMACK can parse 37 out of 45 programs, whereas CIVL can only parse 3 out of 45. In contrast, F-IKOS can analyze all programs within an average time of 1.79s while maintaining an acceptable average of 3 false positives per program. In particular, when analyzing programs that use intrinsic functions to manipulate Fortran's arrays, F-IKOS issues some false positives. The reason for these false positives lies in the differences between Fortran arrays and common high-level language arrays.

RQ-2 Answer: F-IKOS maintains analytical capabilities when dealing with intrinsic functions. F-IKOS successfully analyzed all Fortran programs within an average time of 1.79s. Moreover, F-IKOS can be used to analyze Fortran programs adhering to multiple syntax standards, and the results show that F-IKOS performs better than SMACK and

ID	Program Name	Loc	F-IKOS Time (s)	SMACK Time (s)	CIVL Time (s)
P1	array	15	**0.14**	3.22	2.9
P2	arrary_fail	11	**0.14**	2.79	3.19
P3	compound	19	**0.26**	3.23	1.41
P4	compound_fail	19	**0.31**	2.77	1.48
P5	compound_fail_2	19	**0.30**	3.64	1.71
P6	compute	13	**0.12**	3.14	1.57
P7	compute_fail	13	**0.15**	2.67	1.50
P8	forloop	15	**0.16**	3.62	1.5
P9	forloop_fail	15	**0.21**	2.99	1.89
P10	function	36	**0.18**	4.10	1.82
P11	function_fail	36	**0.19**	3.83	1.62
P12	function_fail_2	35	**0.23**	6.29	1.74
P13	function_fail_3	35	**0.25**	7.39	1.65
P14	hello	14	**0.14**	3.07	1.39
P15	hello_fail	13	**0.16**	2.63	1.62
P16	inout	19	**0.17**	2.20	1.54
P17	inout_fail	19	**0.20**	2.70	1.51
P18	pointer	15	**0.18**	3.28	1.40
P19	pointer_fail	15	**0.22**	2.75	1.46
P20	abs	15	-	-	**1.46**
P21	abs_bad	15	-	-	**1.54**
P22	array_section	29	-	-	**3.63**
P23	array_section_bad	30	-	-	**3.88**
P24	intent_inout	24	**0.19**	-	2.02
P25	intent_out	25	**0.17**	-	2.66
P26	intent_out_bad	24	**0.21**	-	2.28
P27	mod_impl	25	**0.17**	-	2.05
P28	mod_impl_bad	25	**0.18**	-	2.11
P29	mod_spec	19	**0.12**	-	1.48
P30	mult_impl	25	**0.13**	-	1.73
P31	mult_impl_bad	25	**0.13**	-	1.86
P32	mult_spec	21	**0.16**	-	1.46
P33	short_circuit	37	-	-	**3.57**
P34	short_circuit_bad	35	-	-	**3.53**
P35	truncate	14	**0.12**	-	1.42
P36	truncate_bad	14	**0.14**	-	1.63
	Average	22	0.18	3.49	1.98

Table 2
Experimental results on 36 Fortran programs

CIVL.

5.5. RQ3: Handling real-world programs

We evaluated the capability of F-IKOS to detect runtime errors in larger programs from the third benchmark. To objectively demonstrate F-IKOS's capabilities, we intentionally injected 10 division-by-zero bugs into some of these programs. Our evaluation metrics include both analysis time and accuracy. The accuracy of the analysis is defined as follows:

$$Accuracy = \frac{TP}{TP + FP}$$

where TP represents the number of true positives, and FP denotes the number of false positives.

The experimental results are represented in Table 4, where "F-IKOS TP" and "F-IKOS FP" denote true positives and false positives of F-IKOS, respectively. SMACK successfully parsed 7 programs, but it failed to detect runtime errors within them. CIVL couldn't parse any of the 10 programs. In contrast, F-IKOS detected all runtime errors, achieving an accuracy of 22.2%. The programs include both custom and Fortran intrinsic functions, exhibiting more complex numerical characteristics. The results demonstrate the effectiveness of F-IKOS to detect runtime errors in numerical computation programs. Regardless of the rounding mode used by machines, our analysis of Fortran programs remains sound, ensuring consistent and sound analysis results across different computational environments.

We find that some programs require more time for analysis. Two primary factors contribute to the long execution time. Firstly, our experiments utilize the polyhedra abstract domain, which can be computationally expensive for certain programs. Additionally, real-world programs often exhibit distinct numerical characteristics, particularly due to the presence of loops and arrays, which demand more time-intensive processing.

RQ-3 Answer: The results show that SMACK parses 7 out of 10 Fortran programs, while CIVL fails to parse any. Moreover, neither tools can detect runtime errors in the programs they parse. In contrast, F-IKOS soundly analyzes all programs and successfully detects runtime errors, demonstrating its effectiveness in handling Fortran numerical programs.

6. Related Work

In the literature, there exists several tools to analyze or verify Fortran programs. Some tools such as f2c [3] and FABLE[11], involve converting Fortran programs into other high-level languages (such as C++) programs, and then use

ID	Program	Loc	F-IKOS (s)	F-IKOS FP	SMACK (s)	CIVL (s)
P1	arguments	41	**2.40**	0	2.43	-
P2	associate_bounds	9	**0.99**	0	2.71	-
P3	bounds	12	**0.88**	8	3.24	-
P4	boz	11	**1.83**	1	2.85	-
P5	case_insensitivity	15	**0.18**	1	2.43	-
P6	column_major	18	**0.12**	0	2.9	-
P7	compare_floats	15	**2.14**	0	3.25	-
P8	data	21	**0.18**	5	2.91	-
P9	derived_type_composition	21	**1.31**	0	2.73	-
P10	derived_type_implied_do	19	**1.69**	5	3.34	-
P11	dimension	13	**0.36**	0	2.67	-
P12	direct_access	28	**5.16**	15	-	-
P13	do_loop_index	18	**0.17**	0	3.14	0.99
P14	do_while	32	**1.78**	12	3.27	-
P15	error_stop	20	**0.26**	1	-	-
P16	get_command	23	**1.49**	3	3.27	-
P17	implicit_save	41	**0.39**	7	3.17	-
P18	intrinsic	50	**1.18**	0	2.93	-
P19	list_directed_read	20	**1.65**	0	3.23	-
P20	loop	10	**0.16**	4	2.91	1.59
P21	loop_bound	32	**0.59**	0	3.28	-
P22	loop_index	22	**0.34**	5	3.05	-
P23	loop_label	34	**4.45**	15	3.64	-
P24	merge	10	**0.82**	0	3.26	-
P25	module	16	**0.12**	0	2.7	-
P26	module_parameter	35	**0.17**	0	2.62	-
P27	open_file	27	**1.93**	2	2.98	-
P28	overlapping_arg	31	**1.69**	0	2.84	-
P29	print_implied_do_loop	22	**3.10**	16	4.27	-
P30	protected	29	**0.34**	0	2.87	-
P31	recursive_io	37	**0.51**	0	-	-
P32	scratch	19	**1.27**	6	3.47	-
P33	select_case	21	**0.99**	0	3.56	-
P34	slash	15	**0.85**	0	2.9	-
P35	sum_exit	14	**0.23**	5	3.11	1.52
P36	trim	9	**0.26**	0	2.63	-
P37	type_constructor_optional	23	**0.75**	0	2.75	-
P38	value	45	**0.41**	4	2.73	-
P39	write_char	15	**0.76**	0	2.67	-
P40	xrandom_int	17	**4.60**	4	-	-
P41	**swap_arrays**	51	**5.12**	2	-	-
P42	**average**	40	**17.81**	8	-	-
P43	**submod**	41	**0.56**	0	-	-
P44	**linear_equations**	27	**7.29**	3	-	-
P45	**temp_converter**	40	**1.47**	2	2.19	-

Table 3
Experimental results on 45 Fortran programs

verifiers to conduct verification. Alternatively, FORTRAN-lint [13], ftnchek [12] and Coverity [14] specializes in predefined, general-purpose defect detection within Fortran programs, lacking the comprehensive capability to analyze program properties semantically. In contrast, CamFort [15] incorporates a lightweight declarative specification language capable of both checking and inferring specifications.

SMACK [7] and CIVL [8] translate Fortran programs into Intermediate Representation (IR) for subsequent verification using IR-compatible verifiers. Specifically, SMACK converts Fortran programs to LLVM IR and then verifies LLVM IR via Corral [9], wherein Corral restricts the syntax of expressions in this language to one that can be efficiently decided by a SMT solver. CIVL [8] converts Fortran programs to CIVL-C which is an Intermediate Verification Language (IVL), which is subsequently verified using model checking and symbolic execution. However, their work mainly focuses on the verification of Fortran programs, without addressing runtime error detection. In this paper, we use F-IKOS to support the sound analysis of Fortran programs with complex features.

7. Conclusion

In this paper, we present F-IKOS, an abstract interpretation-based static analyzer designed for Fortran programs. Particularly, F-IKOS provides a sound analysis for floating-point types in programs. F-IKOS first abstracts floating-point expressions into real-number expressions with interval coefficients, then linearizes these expressions into real-number expressions with scalar coefficients. These linear expressions are subsequently handled by abstract domains originally designed for real-number types to produce sound analysis results. Evaluation of three benchmarks demonstrates F-IKOS's efficiency and effectiveness than other relevant analyzers in the analysis of Fortran programs.

ID	Program Name	Loc	F-IKOS (s)	F-IKOS TP	F-IKOS FP	SMACK (s)	CIVL (s)
P1	converter	41	1.06	0	2	2.95	-
P2	bubblesort	62	169.81	0	10	-	-
P3	libconstants	107	0.38	0	0	2.17	-
P4	simpson	53	11.62	0	5	3.65	-
P5	differentiation	39	4.48	0	5	-	-
P6	div	111	0.15	6	0	2.98	-
P7	expr	110	0.12	2	0	3.08	-
P8	function	112	12.01	2	0	3.05	-
P9	palindrome	51	63.05	0	7	-	-
P10	trapezodial	102	18.79	0	6	3.27	-

Table 4
Experimental results on 10 Fortran programs

8. Acknowledgments

We thank the reviewers for their constructive feedback. This work is supported by the National Key R&D Program of China (No.2022YFA1005101) and the National Natural Science Foundation of China (Nos.62032024,62102432).

References

[1] Response to issues about floating-point program analysis in IKOS, 2023. URL: https://github.com/NASA-SW-VnV/ikos/issues/224.
[2] Liqian Chen, Sound floating-point and non-convex static analysis using interval linear abstract domains, 2010. National University of Defense Technology.
[3] Feldman, Stuart I, A Fortran to C converter, ACM SIGPLAN Fortran Forum. Vol. 9. No. 2. New York, NY, USA: ACM, 1990.
[4] LLVM homepage, 2000. URL: https://llvm.org/.
[5] Flang Fortran language front-end homepage. URL: https://github.com/flang-compiler/flang.
[6] Jeannet, B., Miné, A., Apron: A library of numerical abstract domains for static analysis, in: Computer Aided Verification, Springer, Berlin, Heidelberg, 2009, pp. 661-667.
[7] Garzella, J.J., Baranowski, M., He, S., Rakamaric, Z., Leveraging compiler intermediate representation for multi- and cross-language verification, in: Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings, Springer, Berlin, Heidelberg, 2020, pp. 90-111.
[8] Wu, W., Hückelheim, J., Hovland, P.D., Siegel, S.F., Verifying Fortran Programs with CIVL, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, 2022, pp. 106-124.
[9] Lal, A., Qadeer, S., Lahiri, S.K., A solver for reachability modulo theories, in: Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012, Proceedings, Springer, Berlin, Heidelberg, 2012, pp. 427-443.
[10] Brat, G., Navas, J.A., Shi, N., Venet, A., IKOS: A framework for static analysis based on abstract interpretation, in: Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1–5, 2014, Proceedings, Springer, Berlin, Heidelberg, 2014, pp. 271-277.
[11] Grosse-Kunstleve, R.W., Terwilliger, T.C., Sauter, N.K., Adams, P.D., Automatic Fortran to C++ conversion with FABLE, Source Code for Biology and Medicine, 7 (2012), pp. 1-11.
[12] Mak, L., Taheri, P., An Automated Tool for Upgrading Fortran Codes, Software, 1(3) (2022), pp. 299-315.
[13] FORTRAN-lint: a pre-compile analysis tool, URL: https://stellar.cleanscape.net/docs_lib/data_F-lint2.pdf.
[14] Coverity Fortran Syntax Analysis, URL: https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/webhelp-files/fortran_start.html#introduction.
[15] Orchard, D., Contrastin, M., Danish, M., Rice, A., Verifying spatial properties of array computations, Proceedings of the ACM on Programming Languages, 1(OOPSLA) (2017), pp. 1-30.
[16] Rakamarić, Z., Emmi, M., SMACK: Decoupling source language details from verifier implementations, in: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014, Proceedings, Springer, Berlin, Heidelberg, 2014, pp. 106-113.
[17] BLAS (Basic Linear Algebra Subprograms), URL: https://www.netlib.org/blas/.
[18] Cousot, P., Cousot, R., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1977, pp. 238-252.
[19] Miné, A., Relational abstract domains for the detection of floating-point run-time errors, in: European Symposium on Programming, Springer, Berlin, Heidelberg, 2004, pp. 3-17.
[20] fortranlib2024 homepage, URL: https://github.com/astrofrog/fortranlib.
[21] FortranTip homepage, URL: https://github.com/Beliavsky/FortranTip.
[22] Fortran4Researchers homepage, URL: https://github.com/WarwickRSE/Fortran4Researchers.
[23] Fiedler, M., Nedoma, J., Ramík, J., Rohn, J., Zimmermann, K., Solvability of systems of interval linear equations and inequalities, in: Linear Optimization Problems with Inexact Data, Springer, Berlin, Heidelberg, 2006, pp. 35-77.