Interval Type Inference: Improvements and Evaluations

by

Di Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Di Wang 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Interval analysis estimates the run-time values of numerical expressions in the source code by computing a lower bound and an upper bound. Interval analysis for integral types is useful in providing facts of the target program to help developers find issues such as unsafe narrowing casts, out-of-bound array indices, numerical overflows/underflows, divisions-by-zero, and dead branches.

Various approaches have been developed to achieve this goal. Pluggable type systems such as the Checker Framework allow developers to customize type checkers of their own interest by associating a type with a particular property and defining specific type rules that restrict the program behaviors. However, the type checkers are intra-procedural, which require manual annotations on all the subroutines invoked in the method being checked. This annotation effort can be a heavy burden to the development of large-scale projects.

A solution to reduce the human effort is inter-procedural, whole-program type inference. Whole-program type inference takes an unannotated program as input and outputs an entire typing for the program that type-checks. If no such typing exists, the reason is either a real type error or a false positive.

Checker Framework Inference is a framework for whole-program type inference built upon the Checker Framework. Constraint variables and constraints are created throughout the whole program based on syntactical type rules. Then the constraints are encoded and solved by a solver.

Value Range Inference is a whole-program inference approach for integral range (interval) analysis, which is implemented with Checker Framework Inference.

This thesis proposes Interval Type Inference, which improves Value Range Inference in the following aspects:

- simplify the interval type hierarchy and the representation of interval types. Thereby reduce the size of the SMT encoding.
- redefine certain type rules as well as the flow-sensitive refinement on comparison, especially in the context of a loop.
- redefine the SMT encoding of constraints including well-formedness constraints, comparison constraints, etc.

To evaluate Interval Type Inference, this thesis conducts experiments on selected open source projects. The experimental results show that Interval Type Inference successfully discovers issues including unsafe narrowing cast and use of invalid input.

Acknowledgements

I would like to give my most sincere thanks to my supervisor Professor Werner M. Dietl, for his constant support and guidance. I would like to thank my readers, Professor Arie Gurfinkel and Professor Mahesh Tripunitara, for their time and valuable feedback. I am thankful to all my colleagues in our research group, especially Jenny Xiang, Weitian Xing, Lian Sun, Zhiping Cai and Piyush Jha for their generous help and support to these two years of my graduate study.

Dedication

This is dedicated to my parents who have always wanted the best for me.

Table of Contents

List of Figures ix

List of Tables x

1 Introduction 1

2 Background and Related Work 4

2.1 Pluggable Type Systems 4

2.1.1 Checker Framework 4

2.1.2 Dataflow Framework 5

2.2 Whole-Program Type Inference 6

2.2.1 Checker Framework Inference 6

2.3 Related Work 13

2.3.1 Constant Value Checker 13

2.3.2 Value Range Inference 14

3 Improvements of Interval Type Inference 15

3.1 Improvements to Type Hierarchy 15

3.1.1 Integral Type Hierarchy 16

3.1.2 Non-Integral Type Hierarchy 16

3.1.3 Polymorphic qualifiers 17

3.2 Improvements to Type Rules 18
3.2.1 Conversions between Integral and Non-integral Types 18
3.2.2 Widening and Narrowing 19
3.3 Improvements to Flow-sensitive Refinement 21
3.3.1 Refinement on Comparison 21
3.3.2 A Demonstration of Refinement on Comparison in `if`-condition . . 23
3.3.3 A Demonstration of Refinement on Comparison in Loops 24
3.4 Changes to Encoding 25
3.4.1 Constraint Variables 26
3.4.2 Constraint Encoding from *Value Range Inference* 26
3.4.3 Well-formedness Constraint 27
3.4.4 Least-Upper-Bound Constraint 30
3.4.5 Arithmetic Constraint 30
3.4.6 Comparison Constraint 31
3.4.7 A Demonstration of Encoding 36

4 Implementation and Evaluation 39
4.1 Implementation 39
4.1.1 Tree Annotator 39
4.1.2 Stub Classes 40
4.1.3 Transfer Function 41
4.1.4 Inference Visitor 41
4.1.5 Constraint Encoder 41
4.1.6 Limitations 42
4.2 Evaluation 43
4.2.1 Unsafe Narrowing Cast 44
4.2.2 Using Invalid Input 45
4.2.3 Dead Branches 45

4.2.4 False Positives 46
4.2.5 Performance 50
4.2.6 Comparison with Value Range Inference 50

5 Conclusion and Future Work 53

References 55

APPENDICES 59

A Unsafe Narrowing Conversions in Selected Projects 60

B Uses of Invalid Input in Selected Projects 69

List of Figures

2.1 Diagram of Checker Framework Inference 7
2.2 Type hierarchy of the Nullness type system 10
2.3 An example to illustrate the creation of constraint variables 11
2.4 CFG for the example in Figure 2.3 12

3.1 An example of widening 20
3.2 An example of if statement 23
3.3 CFG for the example in Figure 3.2 24
3.4 An example of while loop 25
3.5 Control flow graph for the example in Figure 3.4 26
3.6 A example to illustrate constraint encoding 36
3.7 CFG for the example in Figure 3.6 37

4.1 Diagram of Interval Type Inference 40
4.2 A false-positive case caused by under-constrained `Integer.parseInt` . . . 46
4.3 A false-positive case caused by under-constrained `Math.min` 47
4.4 A false-positive case caused by unconstrained post-conditions 48
4.5 A false positive case caused by missing relational analysis 49

List of Tables

4.1 The statistics on UNSAT causes 43
4.2 Performance of Interval Type Inference 50
4.3 Comparison of the number of constraint variables 51
4.4 Comparison of the number of constraints 51
4.5 Comparison of performance 52

Chapter 1

Introduction

Interval analysis [1] estimates the possible run-time values of each expression in the source code by computing a lower bound and an upper bound. The interval analysis for integral types is useful in providing facts of the target program that help developers find issues including unsafe narrowing casts, out-of-bound array indices, numerical overflows/underflows, divisions-by-zero, dead branches, etc. [2][3]. Various approaches have been developed to achieve this goal, including static analysis. Static analysis is the type of program analysis that is performed without executing the program, as opposed to dynamic analysis. Static analysis has the advantage of zero run-time overhead, and it gives developers a guarantee of the absence of domain-specific issues [4].

A type system associates a type with a particular property and enforces a set of type rules to the programming language syntax [5]. A type checker under the specific type system guarantees that the associated property holds throughout the program. Therefore, a type system in the interval domain can be defined by associating a type with the interval of an expression's run-time values, and enforcing the type rule: narrowing cast cannot cause data loss. If a program type-checks in such a type system, narrowing casts in the program are guaranteed to be safe.

The Checker Framework [6][7] is a framework that provides enhanced pluggable type systems for Java in a variety of domains. It provides the built-in Constant Value Checker that supports the analysis for integral value ranges.

However, the type checkers provided by the Checker Framework are intra-procedural, i.e. methods are checked independently, which requires specifications on all the subroutines invoked in the method being checked. If a subroutine is not manually annotated, the

Checker Framework makes sound but conservative assumptions that possibly lead to over-approximation and false positives. Therefore, manual annotations are necessary to achieve higher precision, which bring a heavy burden to the development of large-scale projects.

A solution to reduce the human effort is inter-procedural whole-program type inference [8, 9, 10, 11, 12, 13, 14]. It takes account of the entire program rather than just reasoning one method at a time. Without making conservative assumptions, the whole-program type inference takes an unannotated program as input and infers an entire typing that type-checks. If no such typing exists, the reason is either a real type error or a false positive.

Checker Framework Inference[1] is a framework for constraint-based whole-program inference [8][9] built upon the Checker Framework. The input source code is first compiled by the Java compiler, which produces abstract syntax trees (AST hereinafter) [15] and passes the control to Checker Framework Inference. The workflow of Checker Framework Inference is as follows. First, an AST visitor creates a constraint variable for every type use location, including fields, method parameters, method returns, local variables, etc. Second, dataflow analysis is performed to refine the types of all the expressions with new constraint variables. After the dataflow analysis is completed, a type visitor traverses the ASTs and enforces type rules by generating constraints with the constraint variables that are created before. Then, the constraint variables and constraints for the whole program are encoded and solved by a solver. If the constraints are satisfiable, the solver yields a solution, which is decoded to the typing of the input program. Otherwise the solver gives the core of the unsatisfiable constraints to help with the trouble shooting.

Value Range Inference [16] is a whole-program type inference for the domain of integral value range (or interval), which is implemented upon Checker Framework Inference. It proposes the constraint rules regarding well-formedness of interval types, casting, arithmetic operation, comparison, etc. It also provides the SMT encoding of the constraint rules.

This thesis proposes Interval Type Inference, a whole-program inference for the interval abstract domain, which improves Value Range Inference. Compared with Value Range Inference, Interval Type Inference

- simplifies the interval type hierarchy and the representation of a interval type, thereby reduces the size of the SMT encoding;
- redefines the type rules regarding widening and narrowing;
- redefines constraints about flow-sensitive refinement on comparison;

[1] https://github.com/opprop/checker-framework-inference

- redefines the SMT encoding for constraints including well-formedness constraints, comparison constraints, etc.

To evaluate Interval Type Inference, this thesis conducts experiments on selected open source projects. The experimental results show that Interval Type Inference successfully discovers issues including unsafe narrowing cast, use of invalid input, etc.

The thesis is organized as follows. Chapter 2 introduces the background including pluggable type systems, whole-program type inference, and the related work Value Range Inference. Chapter 3 introduces the improvements of Interval Type Inference compared to Value Range Inference. Chapter 4 shows the implementation details and the evaluation of Integral Type Inference. Experiments are performed on certain open source projects, and the results are analyzed and compared with that of Value Range Inference. Finally, Chapter 5 concludes the whole thesis and discusses the future work.

Chapter 2

Background and Related Work

2.1 Pluggable Type Systems

Statically-typed programming languages have built-in type systems that find and prevent basic errors at compile time. For example, the Java compiler checks for errors such as using variables without initialization, incompatible assignment, unreachable code, etc. However the built-in type system is usually weak in precision and does not check type errors in a variety of domains.

With pluggable type systems, developers can customize type systems of their own interest by associating a type with a particular property and defining specific type rules that restrict program behaviors [7, 17, 18, 19, 20, 21]. A syntax-based type checker guarantees that property holds throughout the code and thereby proves the absence of domain-specific issues.

2.1.1 Checker Framework

The Checker Framework enhances Java type system by providing framework that finds and prevents type error in a variety of domains. It also provides a programming interface that allows developers to customize a type checker based on special needs [22][23].

To design a type system, developers need to specify the type qualifiers in the type system and build the type qualifier hierarchy based on their semantics. A type hierarchy is formed of subtyping relations among type qualifiers. There are two kinds of relations between any two qualifiers in a type hierarchy. If no partial order exists between the

two qualifiers, then the two types are incompatible. Otherwise, one type qualifier is the subtype of the other. In the Checker Framework, type qualifiers are defined and used as Java annotations.

A type checker is implemented as an annotation processor. According to the compilation workflow [15], after the Java compiler parses the source code and produces ASTs, it invokes specified annotation processors including the type checker. The type checker thereby runs an AST visitor (called "Type Visitor") to traverse all the constructs in Java source files.

When a Type Visitor visits a certain tree node, it first determines the type qualifier on the tree from multiple possible sources in the following precedence.

- If the tree being visited is defined in the source code, the precedence is: (1) any explicit annotation added by programmers, (2) default annotation.
- If the tree being visited is defined in bytecode (e.g JDK), the precedence is: (1) any explicit annotation added by programmers, (2) annotation specified in the stub files, (3) annotated JDK[1], (4) default annotation.

After the annotation is determined, the annotated type for the tree is generated. The Type Visitor checks the validity of the annotated type based on the type rules of the specific type system.

The Checker Framework contains various built-in type checkers [7]. For example, the Constant Value Checker estimates possible run-time values of primitive variables or expressions. The Fenum Checker provides the same type-safe guarantees for sets of constants as real enumeration types. Programmers can develop type checkers upon the Checker Framework, e.g. a type checker that prevents unsafe end-of-file (EOF hereinafter) value comparisons [24].

2.1.2 Dataflow Framework

With bare Checker Framework, the type checking process is flow-insensitive and easily causes false positive. The Dataflow Framework [25] provides a framework for dataflow analysis that estimates the abstract values[2] of each expression to improve the precision.

[1] Annotated JDK: https://github.com/opprop/jdk

[2] Abstract value: a term from static analysis[1]. In contrast to concrete values in the concrete execution, static analysis evaluates the expression to a value in an abstract domain. In the Checker Framework, the abstract values to be computed are annotated types.

By default, the Checker Framework incorporates the Dataflow Framework as a pre-pass, which firstly transforms the AST of each method into a control flow graph (CFG hereinafter). It maintains a `Store` at each point of the CFG to store the refined types of variables or expressions. The Dataflow Framework walks through all the CFG nodes by running the corresponding transfer functions[3]. Whenever the type of a variable is refined (through assignement, comparison, method post-condition, etc.), the refined type is stored in `Store`. The process proceeds until it encounters the exit of the CFG. If the CFG contains a loop, the Dataflow Framework iteratively evaluates the loop until it reaches a fixed-point (`Store` values unchanged).

2.2 Whole-Program Type Inference

The type checking in this context is also called "modular type checking" [9]. Modular type checking checks each method independently, which requires manual annotations on all the subroutines invoked in the method being checked. If the annotation for a method parameter or method return is default, the Checker Framework makes sound but conservative assumptions that possibly lead to over-approximation and false positives. Therefore, manual annotations are necessary to achieve higher precision, but bring a heavy burden to the development of large-scale projects.

Whole-program type inference reduces the human effort by taking account of the entire program rather than just reasoning one method at a time. Without making assumptions, the whole-program type inference takes unannotated program as input, and outputs an entire typing for the program that type-checks under the type rules. If no such typing exists, the reason is either a real type error or a false positive.

2.2.1 Checker Framework Inference

Built upon the Checker Framework, Checker Framework Inference provides a constraint-based whole-program inference framework. Figure 2.1 shows the workflow of Checker Framework Inference.

The input source code is parsed by the Java compiler, which produces the intermediate ASTs and invokes an annotation processor that starts Checker Framework Inference.

[3]A transfer function represents the effect of a single CFG-node on the dataflow. It computes the next `Store` based on the previous `Store` and the current CFG node.

The syntax-based Tree Annotator adds a constraint variable to every type use location including fields, method parameters, method returns, local variables, type arguments, type variables, etc. Constraint variables created by the Tree Annotator are referred to as **source variables**, defined as follows[4].

- **Source Variables** These kind of constraint variables are created to represent the type uses in any declarations. The inferred result of source variables are inserted into the source code to provide specifications for those declarations.

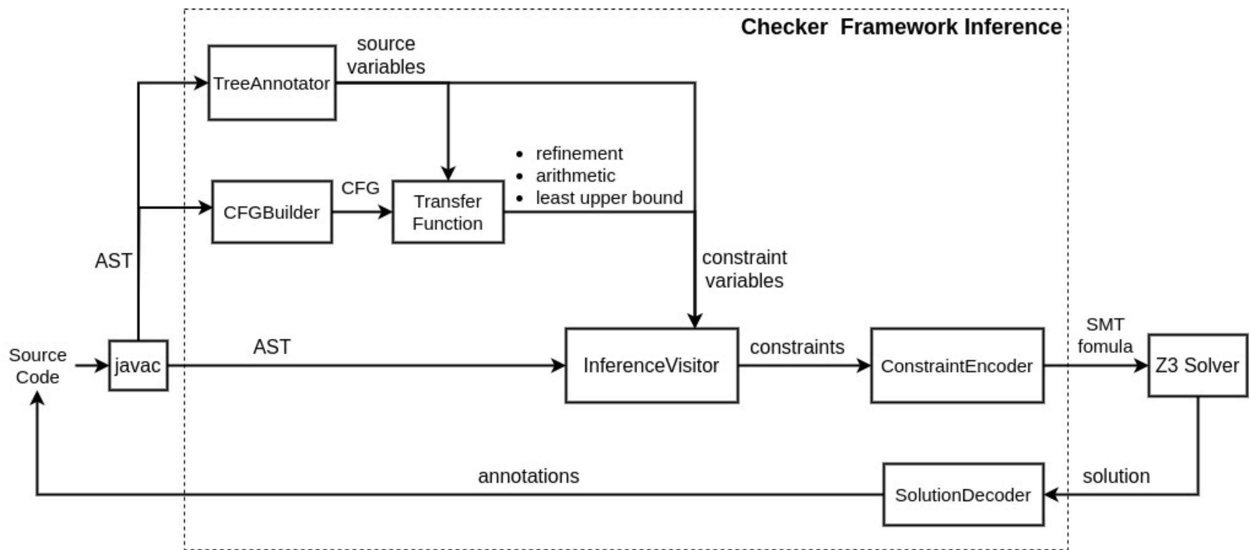

Figure 2.1: Diagram of Checker Framework Inference

The Inference Visitor traverses the AST, gets the annotated type of AST node (which is represented by constraint variable created in the annotating phases), and enforces the type rules by forming constraints with the related constraint variables. As a comparison, in the Checker Framework, when the Nullness Type Visitor visits an assignment tree, and determines the declared type of a left-hand-side (LHS hereinafter) variable is @NonNull, and the type of the right-hand-side (RHS hereinafter) expression is @Nullable, it issues an error. In Checker Framework Inference, the Nullness Inference Visitor use the constraint variable created by the Tree Annotator (denoted by v_{LHS} and v_{RHS}) to generates subtype constraint $v_{RHS} <: v_{LHS}$ (introduced in the following subsection).

[4] In contrast, other kinds of constraint variables are created by dataflow analysis and are described in the following section.

Dataflow Analysis in Inference

Dataflow analysis in the inference context is different from that in type checking. In type checking, a transfer function computes the real type of a CFG node with the real abstract values in the previous `Store`. While in inference, a transfer function creates a new constraint variable to refine the type of a CFG node, and relates the new constraint variable with the previous ones through certain constraints.

Different kinds of constraint variables are created depending on the effect of a tree node. In Checker Framework Inference, constraint variables are classified into the following categories.

- **Refinement Variables** These kind of constraint variables are created to represent the refined type of the LHS of an assignment context.
- **Least-Upper-Bound Variables** These kind of constraint variables are created to represent the type of a variable after multiple execution paths are merged.
- **Comparison Variables** These kind of constraint variables are created to represent the refined type of a variable appears in a comparison expression. Comparison variables are usually created in pairs, one for the then-branch and one for the else-branch.
- **Arithmetic Variables** These kind of constraint variables are created to represent the type of an arithmetic or bit-wise operation.

Constraints

Checker Framework Inference provides the following basic constraints to express type rules of a given type system [8].

- **Subtype Constraint** $(v_1 <: v_2)$ constraint variable v_1 is a subtype of v_2. Subtype constraints are usually used to express type rules regarding assignment, pseudo assignment, etc.
- **Equality Constraint** $(v_1 = v_2)$ constraint variable v_1 and v_2 are equal. Equality constraints are usually used in flow-sensitive refinement.
- **Inequality Constraint** $(v_1 \neq v_2)$ constraint variable v_1 and v_2 are different. Inequality constraints are often used to forbid a constraint variable being assigned a certain type qualifier.

- **Preference Constraint** ($v \ \hat{=}\ c$) constraint variable v equals to type qualifier c by a certain weight. Preference constraints are breakable and are used to express programmers' preference to achieve more precise solutions.

Work Modes

Checker Framework Inference provides three work modes for different purposes [9]: type-check mode, inference mode and annotation mode. Type-check mode fulfills modular type checking. Inference mode and annotation mode fulfill the whole-program type inference. The difference between inference mode and annotation mode is described as follows.

Inference mode is intended to ensure the absence of domain-specific errors in the whole program. The solver only checks the satisfiability of the constraints. If satisfiable, the program is guaranteed to be absent of type errors. If UNSAT, the solver reports the core of the conflicting constraints, so that developers can locate the problematic code and resolve the conflict. This process is repeated until the solver yields satisfiable.

Working upon inference mode, annotation mode gets a model from the solver when the constraints are satisfiable, and then annotates the original source code with the selected model. However, for a given set of mandatory constraints, the model may not be unique. Annotation mode is expected to find a precise typing that is consistent with developers' intention. Therefore breakable, weighted preference constraints (soft constraints) are introduced, so that a MaxSAT solver chooses the model with the least penalty for unsatisfied preference constraints [26]. Compared with inference mode, annotation mode improves the precision at cost of performance.

A Demontration of Checker Framework Inference

We demonstrate the process of whole-program type inference through the Nullness Inference built in Checker Framework Inference. The Nullness type hierarchy is as Figure 2.2.

Figure 2.3 gives an example and illustrates the generated constraint variables. The annotation `@VarAnnot(`i`)` $i \in \mathbb{N}$ uniquely identifies the constraint variable for a type use location. In the example, `@VarAnnot(5)` is the declaration bound for class `Demo`; `@VarAnnot(6)` is the declared type for `field`; `@VarAnnot(8)` is the return type of the method `getField`; `@VarAnnot(9)` is the type of method receiver. The new Object instantiation is constant `@NonNull`, therefore no constraint variable is created for it. The

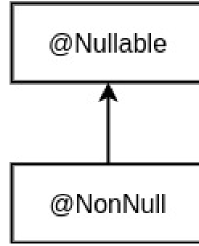

Figure 2.2: Type hierarchy of the Nullness type system

constraint variables mentioned above are created by the purely syntax-based Tree Annotator, which is flow-insensitive.

The constraint variables created in dataflow analysis are not displayed in Figure 2.3. The dataflow analysis process is as follows.

Line 10 For `field == null`, a comparison variable `@VarAnnot(10)` of `field` is created and updates the Store at the beginning of the then-branch, while comparison variable `@VarAnnot(11)` of `field` is created and updates the Store at the beginning of the else-branch.

Meanwhile, one comparison constraint is created for each of the then-branch and else-branch with the comparison variables, as

$$\texttt{@VarAnnot(10)} = \texttt{@Nullable}, \quad \texttt{@VarAnnot(11)} \neq \texttt{@Nullable}$$

Line 11 For the assignment, a refinement variable `@VarAnnot(12)` is created, and the abstract value of `field` in the Store after the assignment is updated to `@VarAnnot(12)`. Meanwhile a refinement constraint is created, which is expressed by an equality constraint between the refinement constraint variable and the RHS types:

$$\texttt{@VarAnnot(12)} = \texttt{@NonNull}$$

if-End At the end of the if-then block when two branches merge, the abstract value of `field` in the then-branch (i.e. `@VarAnnot(12)`) and that in the else-branch (i.e. `@VarAnnot(11)`) is merged. A least-upper-bound variable `@VarAnnot(13)` is created, i.e.

$$\texttt{@VarAnnot(13)} = \texttt{@VarAnnot(12)} \sqcup \texttt{@VarAnnot(11)}$$

which is expressed by two subtype constraints:

$$\texttt{@VarAnnot(12)} <: \texttt{@VarAnnot(13)} \quad \wedge \quad \texttt{@VarAnnot(11)} <: \texttt{@VarAnnot(13)}$$

```
1  class Demo {
2      Object field;
3
4      Object getField() {
5          if (field == null) {
6              field = new Object();
7          }
8          return field;
9      }
10 }
```

```
1  import checkers.inference.qual.VarAnnot;
2
3  @VarAnnot(5)
4  class Demo {
5      @VarAnnot(6)
6      int field;
7
8      @VarAnnot(8)
9      Object getField(@VarAnnot(9) Demo this) {
10         if (field == null) {
11             field = new Object();
12         }
13         return field;
14     }
15 }
```

Figure 2.3: An example to illustrate the creation of constraint variables

After merging, the abstract value of `field` in the Store is updated to `@VarAnnot(13)`

Figure 2.4 shows the Store at each point of the CFG when the dataflow analysis process completes.

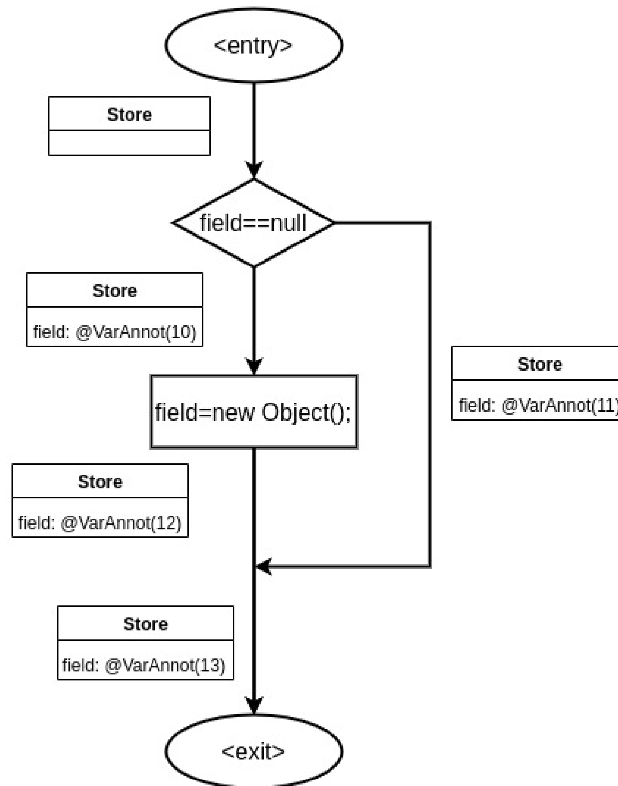

Figure 2.4: CFG for the example in Figure 2.3

After dataflow analysis, the Inference Visitor traverses the AST and creates constraints as follows.

Line 11 The Inference Visitor encounters an assignment. It creates a subtype constraint to enforce the type rule regarding assignment: `@NonNull` <: `@VarAnnot(6)`

Line 13 The Inference Visitor encounters a return tree. It creates a subtype constraint to enforce the type rule regarding pseudo-assignment, `@VarAnnot(13)` <: `@VarAnnot(8)`

Combine all the constraints mentioned above. The deterministic constraint variables are

```
@VarAnnot(10)=@Nullable, @VarAnnot(11)=@NonNull, @VarAnnot(12)=@NonNull
```

However, the solutions for `@VarAnnot(8)` and `@VarAnnot(13)` are not unique. Additional preference constraints are needed to improve the precision. For example, programmers may prefer non-null references and introduce a preference constraint for every constraint variable as follows.

$$\texttt{@VarAnnot}(i) \stackrel{\sim}{=} \texttt{@NonNull}, i \in \mathbb{N}$$

Then the optimal solution for `@VarAnnot(8)` and `@VarAnnot(13)` is

```
@VarAnnot(8)=@NonNull, @VarAnnot(13)=@NonNull
```

2.3 Related Work

2.3.1 Constant Value Checker

Constant Value Checker is a built-in type checker of the Checker Framework, which estimates possible run-time values of expressions. It supports analyses of integer values, float-point values, string values, etc. For integral value analysis, it provides annotations `@IntVal` and `@IntRange`. `@IntVal` specifies the possible concrete values at run-time and takes as argument a set of integers, such as `@IntVal({1, 3, 5})`. While `@IntRange` takes two arguments, a lower bound and an upper bound, such as `@IntRange(from=1, to=5)`. The possible run-time values of an expression are between the bounds (inclusive). Therefore `@IntVal({1, 3, 5})` can be converted to `@IntRange(from=1,to=5)`, even through the even integers in between are impossible values.

Based on the semantic above, `@IntRange` is more efficient in specifying the abstract domain of intervals. A type qualifier `@IntRange(from=`l`,to=`u`)` corresponds to the abstract value of interval $[l, u]$ [1, Section 6.1]. And the subtype relations in the type qualifier hierarchy regarding `@IntRange` corresponds to the subset relations between abstract values of intervals, i.e.

$$\texttt{@IntRange(from=}a\texttt{, to=}b\texttt{)} <: \texttt{@IntRange(from=}c\texttt{, to=}d\texttt{)} \quad \Leftrightarrow \quad [a, b] \sqsubseteq [c, d]$$

The maximal interval supported by Constant Value Checker is $[-2^{63}, 2^{63} - 1]$, the magnitude of `long`.

2.3.2 Value Range Inference

The research work described in this thesis is based on a prototype of whole-program type inference for integral intervals — *Value Range Inference* [16, Chapter 4], which is implemented on Checker Framework Inference.

Value Range Inference reuses the type qualifier hierarchy from Constant Value Checker, but focuses on integral types. The main annotations supported in Value Range Inference are `@UnknownVal`($\top$), `@IntRange`, `@BottomVal`($\bot$), where `@UnknownVal` is equivalent to `@IntRange(from=`-2^{63}`, to=`$2^{63}-1$`)`.

Value Range Inference proposes the constraint rules regarding well-formedness of intervals, casting, arithmetic operation, comparison, etc. It also provides the SMT encoding for the constraints. Value Range Inference introduces soft constraints to support annotation mode, which relies on the Z3 MaxSMT solver [27] to find the optimal solution.

Chapter 3

Improvements of Interval Type Inference

This chapter introduces the improvements of Interval Type Inference compared to Value Range Inference. First, the type hierarchy is simplified by decreasing the number of type qualifiers in it, making the SMT encoding more lightweight, as described in Section 3.1. Second, the type rules regarding integral type operations are refined, as described in Section 3.2. Then, more expressive and more precise flow-sensitive refinement on comparison is introduced in Section 3.3. The encoding of the constraint variables and the constraints are demonstrated in Section 3.4.

3.1 Improvements to Type Hierarchy

Value Range Inference contains three kinds of type qualifiers — `@UnknownVal`($\top$), `@BottomVal`($\bot$) and `@IntRange` (with an upper bound and a lower bound), which require at most five SMT variables to represent a constraint variable: three boolean vairables to select a type qualifier from `@UnknownVal, @BottomVal` and `@IntRange`. If `@IntRange` is selected, then two integer variables are used to represent the lower bound and the upper bound.

To simplify the type hierarchy, Interval Type Inference separates the original type hierarchy into two type hierarchies: the integral type hierarchy and the non-integral type hierarchy. The integral type hierarchy is exclusive for integral types (`byte, short, char, int, long` and the corresponding wrapper classes) and contains exactly one kind of type qualifier — `@IntRange`. The non-integral type hierarchy is exclusive for non-integral types and contains two kinds of type qualifiers — `@UnknownVal`($\top$) and `@BottomVal`($\bot$).

3.1.1 Integral Type Hierarchy

The Interval Type System does not assign separate type qualifiers to $\top$ and $\bot$. The integral type hierarchy contains exactly one kind of type qualifier — `@IntRange`.

- **@IntRange(from=l, to=u)** the interval of the underlying type is $[l, u]$, where $l, u \in [-2^{63}, 2^{63}-1]$. In this context, l denotes the **lower bound** of the interval, and u denotes the **upper bound** of the interval. There are three special cases as follows.
  1. If $l = u$, the underlying expression has constant value.
  2. If $l > u$, the interval type is $\bot$.
  3. If $l = -2^{63}$ and $u = 2^{63} - 1$, the interval type is $\top$.

Therefore, the type hierarchy of the Interval Type System is defined as follows.

$$\forall a, b, c, d \in \mathbb{Z} \cap [-2^{63}, 2^{63}-1], \text{where}[a, b] \subseteq [c, d],$$
$$\text{@IntRange(from=}a, \text{to=}b) <: \text{@IntRange(from=}c, \text{to=}d)$$

3.1.2 Non-Integral Type Hierarchy

In Interval Type Inference, the non-integral type hierarchy is a two-type type hierarchy consisted of `@UnknownVal` ($\top$) and `@BottomVal` ($\bot$). We apply `@UnknownVal` to any non-integral types, with the semantic — "the interval is unknown". `@BottomVal` is only used as the default lower bound of a generic type parameter[1].

Since the Java compiler's built-in type checking forbids direct assignment between non-integral types and integral types, these two type hierarchies are disjoint.

Value Range Inference creates constraint variables and constraints for both integral types and non-integral types uniformly. This causes a waste in the computational resources, since the interval domain is not applicable to non-integral types. Interval Type Inference improves this by creating constraint variables and constraints only for integral types. Each constraint variable is eventually inferred to a type qualifier in the integral type hierarchy. As a result, the simplified interval type hierarchy significantly reduces the size of the constraint variables and constraints, which boosts the performance (see Section 4.2.6).

[1] The Checker Framework allows developers to specify both the upper and the lower bound for a type parameter.

3.1.3 Polymorphic qualifiers

@PolyVal

Value Range Inference supports type qualifier polymorphism for methods, which allows a method having different annotations depending on the concrete method use [16, Section 4.2]. The polymorphic qualifier in Value Range Inference is `@PolyVal`. A use case for `@PolyVal` is boxing of primitive types, which converts a primitive value (e.g. `int`) to an object of the corresponding wrapper class (e.g. `Integer`). The boxing methods are annotated with `@PolyVal` as follows.

```
static @PolyVal Integer valueOf(@PolyVal int arg0);
```

Value Range Inference handles a method invocation of `Byte.valueOf` in the following way. When it encounters a method invocation of `Byte.valueOf`, it replaces all the `@PolyVal` in the method declaration with a constraint variable v_{ret}.

static @v_{ret} Integer valueOf(@v_{ret} int arg0);

Then it gets the constraint variable that represents the argument type (denoted by v_{arg}). According to the subtype relation between the actual argument (RHS) and the formal parameter (LHS), a subtype constraint v_{arg} <: v_{ret} is created. This allows the interval of the method return varying with the interval of the method argument, which can be arbitrary.

However, this method is still under-constrained. According to the semantic of the method `Integer.valueOf`, the method return has the same interval as the method argument. The subtype constraint cannot guarantee that the input and the output have the same interval. In Interval Type Inference, we add additional equality constraint $v_{arg} = v_{ret}$ to get more precise solution, especially for the annotation mode. For example, in the following method,

```
Integer read(InputStream in) throws IOException {
    return in.read();
}
```

`in.read` returns an `int` value in [-1, 255]. Since the method return type is the wrapper class `Integer`, the boxing method `Integer.valueOf` is implicitly invoked, as `Integer.valueOf(in.read())`. Assume the constraint variable for the interval of `Integer.valueOf(in.read())` is V_1, **Value Range Inference** creates a subtype constraint $[-1, 255]$ <: V_1 regarding the polymorphism. In addition, assume the declared type of the method return is V_2, then the subtype constraint V_1 <: V_2 is created according to the subtype

relation between the expression being returned (RHS) and the declared return type (LHS). Therefore, the constraints are

$$[-1, 255] <: V_1, \quad V_1 <: V_2$$

With the two subtype constraints above, we cannot precisely infer the specification of the method as

```
@IntRange(from=-1,to=255) Integer read(InputStream in) throws IOException
```

Additional soft constraints are needed to improve the precision. For example, we can add two soft equality constraints $[-1, 255] = V_1$ and $V_1 = V_2$.

In contrast, **Interval Type Inference** creates the following hard constraints.

$$[-1, 255] = V_1, \quad V_1 <: V_2$$

To get the expected specification of the method, only one soft constraint $V_1 = V_2$ is added. This reduces the number of the constraints and simplifies the encoding.

3.2 Improvements to Type Rules

3.2.1 Conversions between Integral and Non-integral Types

Since integral types and non-integral types have disjoint type hierarchies (Section 3.1), Interval Type Inference defines the type rules regarding conversions between integral types and non-integral types as follows.

1. If an integral expression is cast to a non-integral type, such as
   ```
   Integer x = 0; Object o = (Object) x;
   ```
   the type of the casting is `@UnknownVal`.

2. When a non-integral expression `obj` is cast to an integral type, such as
   ```
   Object o = list.get(0); Integer x = (Integer) o;
   ```
   the type of the casting is `@IntRange(from=`-2^{31}`, to=`$2^{31}-1$`)`.

3.2.2 Widening and Narrowing

Widening or Narrowing conversions (explicit or implicit) of integral types (primitive or boxed type) happen in the following scenarios [28, Section 5.1].

- **Cast Expression** A widening or narrowing conversion occurs explicitly in the form of $(T)e$, where the expression e is converted to the type T.
- **Binary Operation** In Java, the widening conversions related to binary operations between integral expressions are performed in the following steps.
  1. If one of the operands is `long` and the other is not, then the other operand is firstly widened to `long` before the binary operation is performed.
  2. If none of the operands are `long`, and one of the operands is shorter than `int`, that operand is firstly widened to `int` before the binary operation is performed.
- **Assignment** An implicit widening is performed if the RHS type is narrower than the LHS type (e.g., `byte` to `int`), while an implicit narrowing is not supported, i.e., the Java compiler issues an error if the RHS type is wider than the LHS type.
- **Compound Assignment** For compound assignment like `x += 1`, the Java compiler handles it in the following way. If the type of `x` is narrower than `int`, `x` is firstly widened to `int`. Then the arithmetic addition is performed. Finally, the arithmetic result is cast to the original type of `x` and assigned to `x`.
- **Prefix/Postfix Expression** Similar to compound assignment, the increment/decrement of prefix/postfix expressions also follow the "Widen-Compute-Narrow-Assign" pattern if the type of the underlying variable is narrower than `int`.

Narrowing conversions that cause data loss are forbidden. For example, one common misuse of the JDK method `InputStream.read` is the premature conversion from the read `int` to `byte` [24]. `InputStream.read` returns an `int` value in the interval [-1, 255] (`InputStream.read` returns -1 when the input stream reaches EOF). Converting the return value to an unsigned `byte` without EOF test is unsafe. When the value is -1, it is converted to unsigned 8-bit value 255 and handled as a normal byte of data, which may cause unexpected behaviors.

Therefore, the type rule for a narrowing conversion is formalized as follows. Let Γ be the environment before a narrowing conversion. Under Γ an integral expression e has

type $Q\ T_1$ (denoted as $\Gamma \vdash e : Q\ T_1$), where Q is the interval type qualifier and T_1 is the underlying Java type. Converting e to a narrower Java type T_2 (i.e. $T_2 <: T_1$) through $(T_2)e$ is allowed only if the interval Q is contained in the maximal interval of T_2. After the narrowing conversion, the interval type qualifier of the casting $(T_2)e$ remains Q. The type rule is

$$\frac{\Gamma \vdash e : Q\ T_1 \quad T_2 <: T_1 \quad Q <: \text{MaxInterval}(T_2)}{\Gamma \vdash (T_2)\ e : Q\ T_2}$$

Widening conversions between integral types do not cause data loss [28, Section 5.1.2]. Therefore the interval of an expression after a widening conversion does not change. The type rule is formalized as follows. Given the environment $\Gamma \vdash e : Q\ T_1$, where Q is the interval type qualifier and T_1 is the underlying Java type. When e is converted to a wider Java type T_2 (i.e. $T_1 <: T_2$), the interval type qualifier of the casting $(T_2)\ e$ remains Q. The type rule is

$$\frac{\Gamma \vdash e : Q\ T_1 \quad T_1 <: T_2}{\Gamma \vdash (T_2)\ e : Q\ T_2}$$

Both of the narrowing and widening rules mentioned above are expressed by an **equality constraint**. Let the constraint variable for the expression e be V_1, and the constraint variable for the converted type T_2 be V_2. Then the equality constraint is $V_1 = V_2$.

Instead of $V_1 = V_2$, Value Range Inference creates a **subtype constraint** $V_1 <: V_2$. In certain cases, it may yield different results regarding constraint satisfiability. For example in Figure 3.1, the comparison `x > -200` is always `true`, so the else-branch is a dead branch.

```
1 int foo(byte x) {
2     if (x > -200) {
3         return true;
4     }
5     return false;
6 }
```

Figure 3.1: An example of widening

Assume the type of the parameter `x` is V_1. In the comparison, the left operand `x` is first widened to `int` before comparing. Assume the type of the widening conversion is V_2. In Interval Type Inference, the equality constraint $V_1 = V_2$ ensures that the interval of the parameter `x` is equally propagated to the result of the widening conversion, such that $V_2 \sqsubseteq$

$[-128, 127]$. Then the comparison constraint ensures the refined type in the else-branch is $\bot$. Since $\bot$ is forbidden in interval type inference, the constraints are unsatisfiable, and thereby this dead-branch issue is discovered.

However, if a subtype constraint $V_1 <: V_2$ is created, the constraints are satisfiable (a solution is $V_1 = [-128, 127], V_2 = [-2^{31}, 2^{31} - 1]$), which is less precise and misses the dead-branch issue.

3.3 Improvements to Flow-sensitive Refinement

There are two forms of flow-sensitive refinement in the dataflow analysis process of Interval Type Inference: refinement on **assignment** and refinement on **comparison**. Interval Type Inference utilizes the refinement on assignment from Value Range Inference and focuses on improving refinement on comparison.

3.3.1 Refinement on Comparison

Flow-sensitive refinement on comparison is essential to improve precision and reduce false positives. For example, `InputStream.read()` returns a `int` value of the interval [-1, 255]. After checking the value is **not equal** to EOF, the interval is refined to [0, 255]. Then narrowing it to `byte` is type-safe.

When encountering a comparison expression, Value Range Inference introduces a pair of comparison constraint variables to refine each of the variables in the comparison expression, one for the then-branch and one for the else-branch. Comparison constraint variables for the same branch form specific comparison constraints based on the comparison operator [16, Section 4.3.1].

Value Range Inference performs refinement on the following two cases of comparisons.

- One operand of the comparison is a variable, the other operand is constant, such as `x <= 0`.
- Both operands of the comparison are variables, e.g. `x == y`.

Interval Type Inference proposes a new approach of refinement on comparison to support general linear expressions in the form "ax + by $\hat{op}$ c", where a, b, c are constant integers, a $\neq$ 0 or b $\neq$ 0. $\hat{op}$ is one of the six comparison operators $=, \neq, <, \leq, >, \geq$.

To reduce the complexity of the constraint encoding and improve the solving performance, comparisons that contain nonlinear expressions are not refined in Interval Type Inference. Comparisons that contain linear expressions of more than two variables are also not refined, because the size of the encoded SMT formulas grows exponentially to the number of variables in the expression. Moreover, comparison expressions in the form "$c_1x_1 + c_2x_2 + ...c_nx_n \ \hat{op} \ \ c_0$" ($c_i \neq 0, n > 2$) are rarely used in practice according to a case study on Apache commons-csv in this thesis: among 101 comparisons that are linear expressions, only 2 comparisons consist of more than two variables.

In Interval Type Inference, when encountering a comparison expression, it first examines whether the expression can be unified to the form "ax + by $\hat{op}$ c". If so, for each of the then-branch and else-branch, two comparison constraint variables are created to refine x and y respectively, and stored as the initial state in the corresponding branch. The comparison constraint variables of x and y for the then-branch form the comparison constraint indicated by $\hat{op}$. Similarly, the comparison constraint variables for the else-branch form the comparison constraint opposite to $\hat{op}$. In this thesis, we define the comparison constraint for a comparison "ax + by $\hat{op}$ c" with a 6-tuple ($X_{before},\ X_{after},\ Y_{before},\ Y_{after},\ f,\ \hat{op}$), where X_{before}, Y_{before} are the abstract values of x, y before comparison, X_{after}, Y_{after} are the abstract values of x, y after comparison (either in the then-branch or the else-branch), f represents the unified linear function $ax + by - c$, and $\hat{op}$ is the comparison operator. When the comparison expression contains a single variable, as "ax $\hat{op}$ c", the comparison constraint is simplified to a 4-tuple ($X_{before},\ X_{after},\ f,\ \hat{op}$).

Note that the unification is performed in the domain of the SMT encoding (where variables are unbounded) instead of the domain of the program execution (where variables are bounded). Therefore, the change to the form of the comparison expression (e.g. from `x==y+1` to `x-y-1 ==0`) does not raise overflow/underflow issues.

The unification unsoundly omits the possible overflow/underflow in the original condition. For example, in the following case

```
int y = Integer.MAX_VALUE;
int x = Integer.MIN_VALUE;
if (x == y + 1) {   // true, overflow occurs
    ...
}
```

Since the unification is performed in the domain of the SMT encoding (where variables are unbounded), the unified condition `x-y-1==0` is always `false`, which contradicts to the actual program execution. In the future work, additional constraints are added to

both operands of the comparison before the unification to ensure no overflows/underflows occur, so that the unification is sound. For the above case, suppose the intervals of `x` and `y` before the comparison are V_x, V_y respectively, then the following constraints are added before unification.

$$V_x \sqsubseteq [-2^{31}, 2^{31} - 1], \quad V_y + 1 \sqsubseteq [-2^{31}, 2^{31} - 1]$$

These two constraints cause UNSAT, which warns developers the existence of an overflow.

If a comparison expression cannot be unified to the form "ax + by $\hat{op}$ c", the variables in the comparison expression are not refined. The state prior to the comparison is propagated to both of the then-branch and the else-branch for soundness, which is an over-approximation and may cause false positives.

3.3.2 A Demonstration of Refinement on Comparison in `if`-condition

This section demonstrates the flow-sensitive refinement in the context of an if-statement through the example in Figure 3.2. Figure 3.3 shows the corresponding CFG and the constraint variables V_i created at each point of the execution. V_1 is the declared type of the method parameter `x`. The if-condition `x<100` satisfies the requirements in Section 3.3.1, so Interval Type Inference performs refinement by creating a comparison constraint variable V_2 as the initial state of the then-branch, and a comparison constraint variable V_3 as the initial state of the else-branch. Meanwhile, the comparison constraint for the then-branch is created with the 4-tuple (V_1, V_2, '$x - 100$', ' $<$ '); and the comparison constraint for the else-branch is created with the 4-tuple (V_1, V_3, '$x - 100$', ' $\geq$ '). The encoding of comparison constraints are introduced in Section 3.4.6.

```
int foo(int x) {
    if (x < 100) {
        return 100;
    }
    return x;
}
```

Figure 3.2: An example of if statement

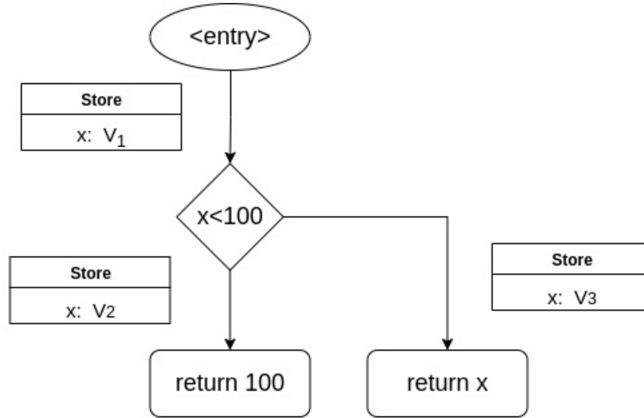

Figure 3.3: CFG for the example in Figure 3.2

3.3.3 A Demonstration of Refinement on Comparison in Loops

The refinement on a loop condition is the same as the refinement on an if-condition. The complexity of the flow refinement in the context of a loop is introduced by the additional backward flow that merges with the flow before the loop condition. Here we demonstrate the refinement for loops through the example in Figure 3.4. Figure 3.5 shows the corresponding CFG and the fixed-point state at each point of the program execution. The left sub-figure is the partial CFG without the backward flow. The right sub-figure is the complete CFG, which shows the effect of the backward flow on the CFG.

Let V_i ($i \in \{1, 2, 3, 4\}$) denote the abstract value of `x` at each point of the flow refinement (the value in each `Store`), and $V_i = [a_i, b_i]$. The flow-refinement process for the example is as follows.

Line 2 After variable initialization, the interval of `x` is refined to [0, 0].

Line 3 When encountering the loop condition `x < 100`, Interval Type Inference creates comparison constraint variable V_1 to refine the type of `x` in the then-branch, and V_2 to refine the type of `x` in the else-branch.

Meanwhile, the comparison constraint for the then-branch is created with the 4-tuple ($[0, 0]$, V_1, '$x - 100$', '$<$'), and the comparison constraint for the else-branch is created with the 4-tuple ($[0, 0]$, V_2, '$x - 100$', '$\geq$').

Line 4 The refinement for the increment `x = x + 1` contains the following steps: (1) an arithmetic constraint variable (denoted by V_a) is created for the RHS addition.

$V_a = [a_1 + 1, b_1 + 1]$. (2) a refinement constraint variable V_3 is created for x, such that $V_3 = V_a$.

Loop End The backward flow at the end of the loop body is merged with the flow before the loop. Therefore, a least-upper-bound constraint variable V_4 is created to represent the merge between [0, 0] and V_3, i.e. $V_4 = V_3 \sqcup [0, 0]$.

Comparison Update Since the backward flow changes the abstract value of x before the comparison at line 3, the comparison constraints need to be updated by substituting V_4 for [0, 0], such that

1. $V_1 \sqcup V_2 = V_4$
2. The comparison constraints for the then-branch and the else-branch are updated to (V_4, V_1, '$x - 100$', ' $<$ ') and (V_4, V_2, '$x - 100$', ' $\geq$ ') respectively.

```
1 int foo() {
2     int x = 0;
3     while (x < 100) {
4         x = x + 1;
5     }
6     return x;
7 }
```

Figure 3.4: An example of while loop

The encoding introduced in Section 3.4 ensure the following solution.

$$V_1 = [0, 99], V_2 = [100, 100], V_3 = [1, 100], V_4 = [0, 100]$$

3.4 Changes to Encoding

This section introduces the SMT encoding in Interval Type Inference. Section 3.4.1 describes the encoding of constraint variables. Section 3.4.2 summarizes the reused part of encoding from Value Range Inference. Because of the improvements to type hierarchy (Section 3.1) and flow-sensitive refinement (Section 3.3), the encoding of well-formedness constraints, least-upper-bound constraints, arithmetic constraints and comparison constraints are changed and optimized accordingly, as described in Section 3.4.3 to 3.4.6.

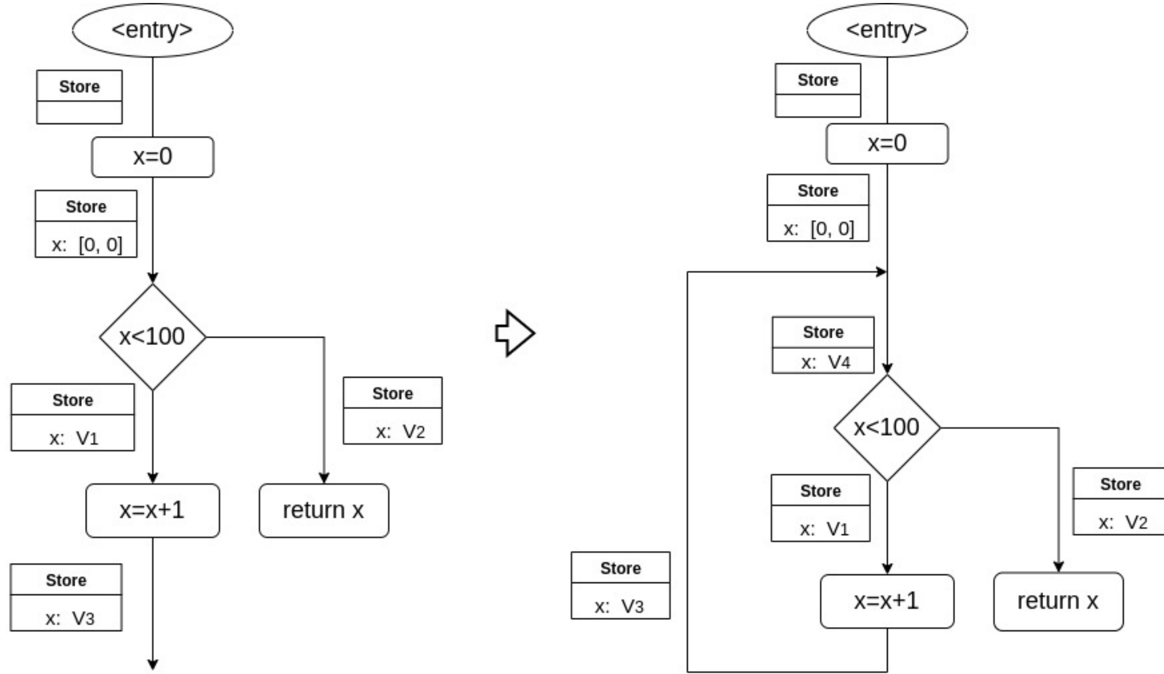

Figure 3.5: CFG for the example in Figure 3.4. The left sub-figure is the partial CFG without the backward flow, while the right sub-figure shows the effect of the backward flow.

3.4.1 Constraint Variables

As mentioned in Section 3.1.1, Interval Type Inference contains exactly one kind of type qualifier — `@IntRange`. To encode a constraint variable that stands for a real type qualifier, only two variables α, β are required to represent the lower bound and the upper bound of the interval, i.e. `@IntRange(from=`α`, to=`β`)`. Since Value Range Inference represents a constraint variable using three boolean variables and two integer variables, our approach decreases the number of SMT variables by up to 60%.

3.4.2 Constraint Encoding from *Value Range Inference*

Interval Type Inference reuses the constraint encoding from Value Range Inference for subtype constraints, equality constraints and refinement constraints [16, Section 4.4]. This section briefly summarizes the encoding of these constraints for the completeness of the thesis.

The following constraints involve multiple constraint variables. Denote the constraint variables

$$V_i := \text{@IntRange}(\text{from} = \alpha_i, \text{to} = \beta_i), i \in \mathbb{Z}^+$$

Subtype Constraint $V_1 <: V_2$

When a constraint variable V_1 is a subtype of V_2, it means the interval represented by V_2 contains the interval represented by V_1, i.e. the encoding is

$$\alpha_1 \geq \alpha_2 \wedge \beta_1 \leq \beta_2$$

Equality Constraint $V_1 = V_2$

When two constraint variables are equal, they have equal lower bounds and equal upper bounds, i.e. the encoding is

$$\alpha_1 = \alpha_2 \wedge \beta_1 = \beta_2$$

Refinement Constraint $V_3[\ V_1\ ] = V_2$

The refinement constraint refers to the constraint created for refinement on assignment. (For the refinement on comparison, a comparison constraint is created. See Section 3.4.6.) For an assignment $x = e$, a refinement constraint $V_3[\ V_1\ ] = V_2$ is created, where V_1 represents the declared type of x, V_2 represents the interval of the RHS expression e, and V_3 is the refinement constraint variable for the assignment. After refinement, the type of x is updated to V_3 and stored until the next refinement on x occurs.

$V_3[\ V_1\ ] = V_2$ is equivalent to a subtype constraint $V_2 <: V_1$ and an equality constraint $V_3 = V_2$. Therefore the encoding is

$$V_3[\ V_1\ ] = V_2 \quad \Leftrightarrow \quad \alpha_2 \geq \alpha_1 \wedge \beta_2 \leq \beta_1 \wedge \alpha_3 = \alpha_2 \wedge \beta_3 = \beta_2$$

3.4.3 Well-formedness Constraint

Well-formedness constraints ensure that a constraint variable represents a valid interval [16]. Since Interval Type Inference improves the encoding of constraint variables, the well-formedness constraints for constraint variables are re-encoded. Given a constraint variable

V := `@IntRange(from=`α`,to=`β`)`, the well-formedness constraint is encoded depending on the kind of V, as follows.

Source Variables

$$\alpha \leq \beta \ \wedge \ \alpha \geq min(T) \wedge \beta \leq max(T)$$

where T stands for an integral Java types. Based on the Java Language Specification [28, Section 4.2],

- If T is `char` (or `Character`), min(T) = 0, max(T) = $2^{16} - 1$;
- If T is `int` (or `Integer`), min(T) = -2^{31}, max(T) = $2^{31} - 1$;
- If T is `long` (or `Long`), min(T) = -2^{63}, max(T) = $2^{63} - 1$;

According to the case study on certain Apache Commons projects (Section 4.2), we found that unsigned `byte` and unsigned `short` are frequently used in applications that involve IO operations. For example, `InputStream.read()` returns an `int` value in [0, 255] when it reads data successfully, or -1 when the input stream reaches EOF. A common use case is to convert the `int` value to unsigned `byte` after EOF test.

Therefore, Interval Type Inference supports unsigned `byte` and unsigned `short` as follows. If the underlying Java type is `byte` or `Byte`, the encoding of the well-formedness constraint is

$$\alpha \leq \beta \ \wedge \ \ (\ (\alpha \geq -128 \wedge \beta \leq 127) \vee (\alpha \geq 0 \wedge \beta \leq 255)\ )$$

If the underlying Java type is `short` or `Short`, the well-formedness constraint is encoded as

$$\alpha \leq \beta \ \wedge \ \ (\ (\alpha \geq -2^{15} \wedge \beta \leq 2^{15} - 1) \vee (\alpha \geq 0 \wedge \beta \leq 2^{16} - 1)\ )$$

Refinement Variables

Since the lower bound and the upper bound of a refinement variable rely on that of the RHS expression, it is redundant to constrain it with the Java integral type bounds. Therefore,

$$\alpha \leq \beta$$

Arithmetic Variables

Considering the possibility of arithmetic overflow/underflow, additional variables are introduced to represent the **computed interval** of an arithmetic operation. Denote $\alpha^{'}$, $\beta^{'}$ as the computed lower bound and upper bound of the arithmetic operation. If $[\alpha^{'}, \beta^{'}] \sqsubseteq [\min(T), \max(T)]$, then the actual bounds $[\alpha, \beta] = [\alpha^{'}, \beta^{'}]$; Otherwise we over-approximate the interval to the maximal interval for soundness, i.e. $[\alpha, \beta] = [\min(T), \max(T)]$. Therefore, the well-formedness encoding is as follows.

$$\alpha \leq \beta$$

$$\wedge$$

$$\begin{aligned}
(\ (\alpha^{'} \geq \min(T) \ \wedge \ \beta^{'} \leq \max(T) \ \wedge \ \alpha = \alpha^{'} \ \wedge \ \beta = \beta^{'}) \vee \\
(\alpha^{'} < \min(T) \wedge \alpha = \min(T) \wedge \beta = \max(T)) \vee \\
(\beta^{'} > \max(T) \wedge \alpha = \min(T) \wedge \beta = \max(T)) \ )
\end{aligned}$$

Least-Upper-Bound Variables

Since the lower bound and the upper bound of a least-upper-bound variable rely on the constraint variables that merge to the least upper bound, it is redundant to constrain it with the Java integral type bounds. Therefore,

$$\alpha \leq \beta$$

Comparison Variables

Since the lower bound and the upper bound of a comparison variable never exceed the interval of the variable it refines, it is redundant to constrain it with the Java integral type bounds. Therefore,

$$\alpha \leq \beta$$

3.4.4 Least-Upper-Bound Constraint

When a constraint variable V_3 is the least upper bound of V_1 and V_2, the interval represented by V_3 is the union of V_1 and V_2. The constraint is denoted as $V_3 = V_1 \sqcup V_2$.

Value Range Inference represents $V_3 = V_1 \sqcup V_2$ with two subtype constraints $V_1 <: V_3$ and $V_2 <: V_3$. This is not precise because it only ensures that V_3 is an upper bound of V_1 and V_2 in the lattice, not the least upper bound. Therefore, it does not give the exact lower bound and upper bound of V_3.

In Interval Type Inference, the lower bound and the upper bound of V_3 are expressed using V_1 and V_2, as

$$V_3 = V_1 \sqcup V_2 \quad \Leftrightarrow \quad \alpha_3 = \min(\alpha_1, \alpha_2) \wedge \beta_3 = \max(\beta_1, \beta_2)$$

where α_3=min(α_1, α_2) is equivalent to

$$(\alpha_1 \leq \alpha_2 \wedge \alpha_3 = \alpha_1) \ \vee \ (\alpha_1 > \alpha_2 \wedge \alpha_3 = \alpha_2)$$

and β_3=max(β_1, β_2) is equivalent to

$$(\beta_1 \leq \beta_2 \wedge \beta_3 = \beta_2) \ \vee \ (\beta_1 > \beta_2 \wedge \beta_3 = \beta_1)$$

3.4.5 Arithmetic Constraint

In contrast to Value Range Inference, Interval Type Inference avoids any non-linear arithmetic operations in the SMT encoding to use the theory of linear integer arithmetic for decidability and higher performance. The non-linear arithmetic operations include multiplication between two constraint variables, division/modulo where the divisor is a constraint variable, bit-shift where the number of positions to shift is a constraint variable, etc.

In Section 3.4.3, we introduced computed lower bound $\alpha_i^{'}$ and upper bound $\beta_i^{'}$ for arithmetic constraint variables, which are used in the encoding of arithmetic constraints. Take addition $V_3 = V_1 + V_2$ as an example, the computed lower bound and upper bound are as follows.

$$V_3 = V_1 + V_2 \quad \Leftrightarrow \quad \alpha_3^{'} = \alpha_1 + \alpha_2 \wedge \beta_3^{'} = \beta_1 + \beta_2$$

If either of the computed bounds exceeds the limit of the underlying Java integral types (which means an overflow/underflow is possible), then the real interval for V_3 is over-approximated to the maximal interval of the underlying type of the arithmetic operation for soundness. Assume the underlying types of V_1, V_2 are `int`, then the encoding is

$$
\begin{gathered}
\alpha_3^{'} \leq \beta_3^{'} \\
\wedge \\
(\ (\alpha_3^{'} < -2^{31} \wedge \alpha_3 = -2^{31} \wedge \beta_3 = 2^{31} - 1) \vee \\
(\beta_3^{'} > 2^{31} - 1 \wedge \alpha_3 = -2^{31} \wedge \beta_3 = 2^{31} - 1) \vee \\
(\alpha_3^{'} \geq -2^{31} \ \wedge \ \beta_3^{'} \leq 2^{31} - 1 \wedge \alpha_3 = \alpha_3^{'} \wedge \beta_3 = \beta_3^{'})\ )
\end{gathered}
$$

3.4.6 Comparison Constraint

According to Section 3.3.1, the comparison constraint for a comparison in the form "ax + by $\hat{op}$ c" is defined by a 6-tuple

$$(X_{before},\ X_{after},\ Y_{before},\ Y_{after},\ f,\ \hat{op})$$

where X_{before}, Y_{before} are the abstract values of x, y before comparison, X_{after}, Y_{after} are the abstract values of x, y after comparison, f represents the unified linear function $ax + by - c$, and $\hat{op}$ is the comparison operator. When the comparison expression contains a single variable, as "ax $\hat{op}$ c", then the comparison constraint is simplified to a 4-tuple $(X_{before},\ X_{after},\ f,\ \hat{op})$. In this section we present the encoding for the more general 2-variable expressions, and it is easy to simplify it to 1-variable expressions.

A comparison constraint is encoded by expressing the bounds of X_{after}, Y_{after} with the bounds of X_{before}, Y_{before}, based on the particular linear function f and the relation $\hat{op}$, i.e.

$$\boldsymbol{\alpha_{after}} = f_L(\boldsymbol{\alpha_{before}}, \boldsymbol{\beta_{before}})$$

$$\boldsymbol{\beta_{after}} = f_U(\boldsymbol{\alpha_{before}}, \boldsymbol{\beta_{before}})$$

where $\boldsymbol{\alpha_{after}} = (\alpha_{X_{after}}, \alpha_{Y_{after}})$ is a 2-dimension vector consisted of the lower bounds of $X_{after},\ Y_{after}$. $\boldsymbol{\beta_{after}} = (\beta_{X_{after}}, \beta_{Y_{after}})$ is a 2-dimension vector consisted of the upper bounds of X_{after}, Y_{after}. The similar for $\boldsymbol{\alpha_{before}}$ and $\boldsymbol{\beta_{before}}$. Functions f_L, f_U vary with the linear function f and comparison operator $\hat{op}$.

To determine f_L, f_U, we perform case analysis on the lower bound and upper bound of X_{after}, Y_{after} separately. We take the lower bound of X_{after} for example, and simplify the notation by letting

$$X_{before} = [x_1, x_2], Y_{before} = [y_1, y_2], X_{after} = [x_3, x_4], Y_{after} = [y_3, y_4]$$

1. The lower bound before comparison is the lower bound after comparison (i.e. $x_3 = x_1$), iff $\exists y \in [y_1, y_2]$ such that the comparison condition is true (i.e. $f(x_1, y) \;\; \hat{op} \;\; 0$).
2. Otherwise, the lower bound before comparison cannot be the lower bound after comparison. Then the lower bound after comparison is a value between $x_1 + 1$ and x_2 (i.e. $x_3 > x_1$). In this case, x_3 is reasoned from the following constraints.
   (a) $\exists y \in [y_1, y_2]$ such that the pair of $(x_3,\ y)$ satisfies the comparison condition, and
   (b) $\nexists y \in [y_1, y_2]$ such that the pair of $(x_3 - 1,\ y)$ satisfies the comparison condition.

The detailed encoding for comparison constraints vary with the comparison operators. Here we demonstrate the encoding details of “equal-to” and “less-than”.

“Equal-To” Comparison

Given a general two-variable “equal-to” comparison $c_1x + c_2y = k$ (c_1, c_2, k are constant integers, $c_1 \neq 0, c_2 \neq 0$). We first unify the expression to the standard form $c_1x + c_2y = k$ where c_1 and c_2 are positive integers. This minimizes the size of the encoding without considering the polarity of the coefficients.

1. If $c_1 < 0$, convert the expression to $-c_1x - c_2y = -k$, and update $c_1 \leftarrow -c_1, c_2 \leftarrow -c_2, k \leftarrow -k$. Otherwise go to step 2.
2. For the intermediate result $c_1x + c_2y = k$ (where $c_1 > 0$) from step 1, if $c_2 < 0$, introduce an auxiliary variable z. Let $z = -y$ and update $c_2 \leftarrow -c_2$. Otherwise $z = y$. Done.

The unified comparison expression to be encoded is

$$c_1x + c_2z - k = 0 \ (c_1, c_2 \in \mathbb{Z}^+, k \in \mathbb{Z})$$

Denote $f(x, z) = c_1x + c_2z - k$ in the following text for brevity.

Assume before comparison, the environment $\Gamma \vdash x : X_{be}$, $\Gamma \vdash z : Z_{be}$. In the then-branch after refinement, $\Gamma \vdash x : X_{af}$, $\Gamma \vdash z : Z_{af}$. Here we introduce the approach to represent the after-comparison state X_{af}, Z_{af} using the before-comparison state X_{be}, Z_{be}. Let

$$X_{be} = [x_1, x_2],\ Z_{be} = [z_1, z_2],\ X_{af} = [x_3, x_4],\ Z_{af} = [z_3, z_4]$$

For the **lower bound** of X_{af}, we perform the following case analysis:

1. $x_3 = x_1$ if $\exists z \in [z_1, z_2]$, such that $f(x_1, z) = 0$, i.e. $[z_1, z_2]$ contains a zero-crossing for the equation $f(x_1, z) = 0$. Therefore,

$$f(x_1, z_1) \leq 0 \ \wedge \ f(x_1, z_2) \geq 0 \quad \Rightarrow \quad x_3 = x_1$$

2. Otherwise, $\nexists z \in [z_1, z_2]$, such that $f(x_1, z) = 0$. Either the maximum value $f(x_1, z_2) < 0$ or the minimum value $f(x_1, z_1) > 0$ (note that in the latter case the $f(x, y)$ is always positive), i.e. $f(x_1, z_2) < 0$. Then $x_1 < x_3 \leq x_2$, and x_3 satisfies the following constraints:

   (a) $\exists z \in [z_1, z_2]$, such that $f(x_3, z) = 0$. $\Leftrightarrow$

$$f(x_3, z_1) \leq 0 \ \wedge \ f(x_3, z_2) \geq 0$$

   (b) $\nexists z \in [z_1, z_2]$, such that $f(x_3 - 1, z) = 0$. $\Leftrightarrow$ $\forall z \in [z_1, z_2], f(x_3 - 1, z) < 0$ $\Leftrightarrow$

$$f(x_3 - 1, z_2) < 0$$

Combining the two cases, the encoding for the lower bound of X_{af} (i.e. x_3) is as follows.

$$\begin{gathered}(\ f(x_1, z_1) \leq 0 \ \wedge \ f(x_1, z_2) \geq 0 \wedge x_3 = x_1 \ ) \\ \vee \\ (\ f(x_1, z_2) < 0 \ \wedge \ f(x_3, z_1) \leq 0 \ \wedge \ f(x_3, z_2) \geq 0 \ \wedge \ f(x_3 - 1, z_2) < 0 \ )\end{gathered}$$

For the **upper bound** of X_{af}, we perform the following case analysis:

1. $x_4 = x_2$ if $\exists z \in [z_1, z_2]$, such that $f(x_2, z) = 0$, i.e. $[z_1, z_2]$ contains a zero-crossing for the equation $f(x_2, z) = 0$. Therefore,

$$f(x_2, z_1) \leq 0 \ \wedge \ f(x_2, z_2) \geq 0 \quad \Rightarrow \quad x_4 = x_2$$

2. Otherwise, $\nexists z \in [z_1, z_2]$, such that $f(x_2, z) = 0$. Either the maximum value $f(x_2, z_2) < 0$ or the minimum value $f(x_2, z_1) > 0$ (note that in the former case the $f(x, y)$ is always negative), i.e. $f(x_2, z_1) > 0$. Then $x_1 \le x_4 < x_2$, and x_4 satisfies the following constraints:

   (a) $\exists z \in [z_1, z_2]$, such that $f(x_4, z) = 0$, i.e.

   $$f(x_4, z_1) \le 0 \ \wedge \ f(x_4, z_2) \ge 0$$

   (b) $\nexists z \in [z_1, z_2]$, such that $f(x_4 + 1, z) = 0$. $\Leftrightarrow$ $\forall z \in [z_1, z_2], f(x_4 + 1, z) > 0$ $\Leftrightarrow$

   $$f(x_4 + 1, z_1) > 0$$

Combining the two cases, the encoding for the upper bound of X_{af} (i.e. x_4) is as follows.

$$\begin{gathered}(\ f(x_2, z_1) \le 0 \ \wedge \ f(x_2, z_2) \ge 0 \wedge \ x_4 = x_2 \ ) \\ \vee \\ (\ f(x_2, z_1) > 0 \ \wedge \ f(x_4, z_1) \le 0 \ \wedge \ f(x_4, z_2) \ge 0 \ \wedge \ f(x_4 + 1, z_1) > 0 \ )\end{gathered}$$

Symmetrically, Z_{af} is encoded in the same way as above.

"Less-Than" Comparison

Given a general two-variable "less-than" comparison $c_1 x + c_2 y < k$ (c_1, c_2, k are constant integers, $c_1 \neq 0$ and $c_2 \neq 0$). We first unify the expression to the standard form $c_1 x + c_2 y < k$ or $c_1 x + c_2 y > k$ where c_1 and c_2 are positive integers. The steps are as follows.

1. If $c_1 > 0$, go to step 2; otherwise convert the expression to $-c_1 x - c_2 y > -k$, and update $c_1 \leftarrow -c_1, c_2 \leftarrow -c_2, k \leftarrow -k$. Then go to step 3.

2. For the current intermediate result $c_1 x + c_2 y < k$ (where $c_1 > 0$), if $c_2 < 0$, introduce an auxiliary variable z. Let $z = -y$, and update $c_2 \leftarrow -c_2$. Otherwise $z = y$. Done.

3. For the current intermediate result $c_1 x + c_2 y > k$ (where $c_1 > 0$), if $c_2 < 0$, introduce an auxiliary variable z. Let $z = -y$, and update $c_2 \leftarrow -c_2$. Otherwise $z = y$. Done. Since the original "less-than" comparison is converted to a "greater-than" comparison, **the encoding of "greater-than" comparison is applied** and is omitted here.

The unified comparison expression to be encoded is

$$c_1x + c_2z - k < 0 \ (c_1, c_2 \in \mathbb{Z}^+, k \in \mathbb{Z})$$

Denote $f(x, z) = c_1x + c_2z - k$ in the following text for brevity.

Assume before comparison the environment $\Gamma \vdash x : X_{be}$, $\Gamma \vdash z : Z_{be}$. In the then-branch after refinement, $\Gamma \vdash x : X_{af}$, $\Gamma \vdash z : Z_{af}$. Here we introduce the approach to represent the after-comparison state X_{af}, Z_{af} using the before-comparison state X_{be}, Z_{be}. Let

$$X_{be} = [x_1, x_2],\ Z_{be} = [z_1, z_2],\ X_{af} = [x_3, x_4],\ Z_{af} = [z_3, z_4]$$

For the lower bound of X_{af}, $x_3 = x_1$.

For the upper bound of X_{af}, we perform the following case analysis:

1. If $\exists z \in [z_1, z_2]$, such that $f(x_2, z) < 0$, then $x_4 = x_2$, i.e.

$$f(x_2, z_1) < 0 \ \Rightarrow \ x_4 = x_2$$

2. Otherwise, $\nexists z \in [z_1, z_2]$, such that $f(x_2, z) < 0$, i.e. $f(x_2, z_1) \geq 0$. Then $x_1 \leq x_4 < x_2$, and x_4 satisfies the following constraints:

   (a) $\exists z \in [z_1, z_2]$, such that $f(x_4, z) < 0$, i.e.

$$f(x_4, z_1) < 0$$

   (b) $\nexists z \in [z_1, z_2]$, such that $f(x_4 + 1, z) < 0$. $\Leftrightarrow$ $\forall z \in [z_1, z_2], f(x_4 + 1, z) \geq 0$ $\Leftrightarrow$

$$f(x_4 + 1, z_1) \geq 0$$

Combining the two cases, the encoding for the upper bound of X_{af} is as follows.

$$\begin{gathered} (\ f(x_2, z_1) < 0 \ \wedge \ x_4 = x_2 \ ) \\ \vee \\ (\ f(x_2, z_1) \geq 0 \ \wedge \ f(x_4, z_1) < 0 \ \wedge \ f(x_4 + 1, z_1) \geq 0 \ ) \end{gathered}$$

Symmetrically, Z_{af} is encoded in the same way as above.

3.4.7 A Demonstration of Encoding

This section demonstrates the encoding of the example in Figure 3.6. `InputStream.read` is a library method which returns an `int` value in [-1, 255]. Therefore at line 2, the interval of the RHS expression `in.read()` is a constant interval [-1, 255]. At line 3, `data` is compared with -1 (i.e. EOF) before casting to ensure the interval of `data` is within the magnitude of unsigned byte, [0, 255]. Therefore, the narrowing cast is safe.

In the context of Interval Type Inference, the following constraint variables are created. An invisible constraint variable `@VarAnnot(6)` is the declared type of the local variable `data`. An invisible constraint variable `@VarAnnot(8)` is the refinement constraint variable when `data` is assigned the return value of `in.read()` at line 2. `@VarAnnot(10)` and `@VarAnnot(11)` (not shown in the annotated code) are the comparison constraint variables for the then-branch and else-branch separately after the comparison at line 3. `@VarAnnot(12)` is the interval type of the narrowing cast to `byte` at line 4. Another invisible constraint variable `@VarAnnot(13)` is the least-upper-bound constraint variable created for `data` when the two branches merge. `@VarAnnot(14)` is the interval type of the method return. For brevity, we use V_i ($i \in \mathbb{N}$) to refer to `@VarAnnot(`i`)` in the rest of this section, and let $V_i := [a_i, b_i]$. Figure 3.7 shows the result of dataflow analysis.

```
1  byte foo(InputStream in) throws IOException {
2      int data = in.read();   // RHS: [-1,255]
3      if (data > -1) {
4          return (byte) data;
5      }
6      throw new EOFException();
7  }
8
9
10 @VarAnnot(14) byte foo(InputStream in) throws IOException {
11     int data = in.read();
12     if (data > -1) {
13         return (@VarAnnot(12) byte) data;
14     }
15     throw new EOFException();
16 }
```

Figure 3.6: A example to illustrate constraint encoding. Lines 1-7 are the raw code. Lines 10-16 are the annotated code with the constraint variables.

The constraints created through the dataflow analysis and the Type Visitor are as follows.

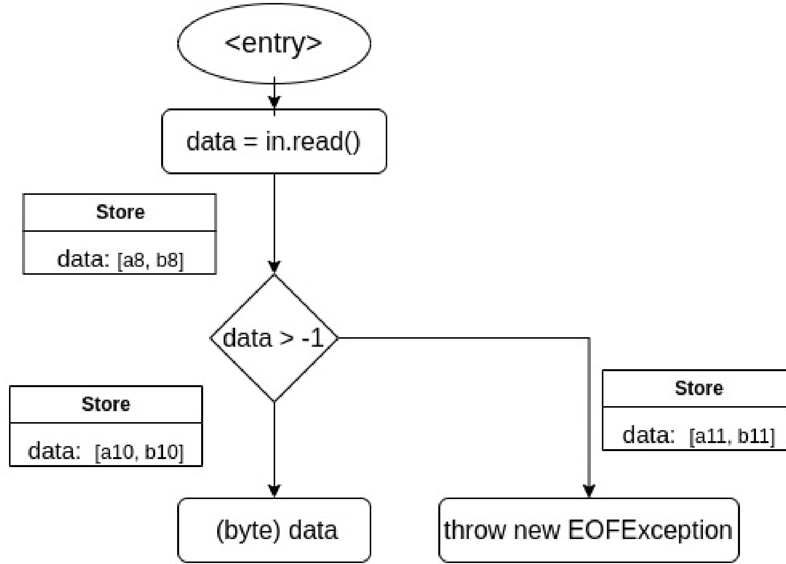

Figure 3.7: CFG for the example in Figure 3.6.

Line 2 : A refinement constraint $V_8[V_6] = [-1, 255]$ is created for the assignment. The refinement constraint is converted to a subtype constraint $[-1, 255]$ <: V_6, and an equality constraint $V_8 = [-1, 255]$. Therefore $a_8 = -1, b_8 = 255$.

Line 3 : A comparison constraint of 4-tuple $(V_8, V_{10}, \text{'x + 1'}, \text{'} > \text{'})$ is created for the refinement on the then-branch. The comparison constraint is encoded by representing the bounds of V_{10} with the bounds of V_8, as follows.

For the upper bound of V_{10}, $b_{10} = b_8 = 255$. For the lower bound of V_{10}, a_{10}, we perform the following case analysis.

1. If the lower bound of V_8 satisfies the greater-than condition, i.e. $a_8 > -1$, then a_8 is the lower bound of V_{10}, i.e. $a_{10} = a_8$. However, this case is **not applicable** because it is already derived from the previous constraints that $a_8 = -1$.
2. Otherwise, the lower bound of V_8 does not satisfy the greater-than condition. Then the lower bound of V_{10}, a_{10} is a value between $a_8 + 1$ and b_8. Therefore the following constraints hold: $a_{10} > -1 \ \wedge a_{10} - 1 \leq -1. \ \Rightarrow \ a_{10} = 0$.

Similarly, a comparison constraint of 4-tuple $(V_8, V_{11}, \text{'x + 1'}, \text{'} \leq \text{'})$ is created for the refinement on the else-branch. The comparison constraint is encoded by representing the bounds of V_{11} with the bounds of V_8, as follows.

For the lower bound of V_{11}, $a_{11} = a_8 = -1$. For the upper bound of V_{11}, b_{11}, we perform the following case analysis.

1. If the upper bound of V_8 satisfies the less-than-equal condition, i.e. $b_8 \leq -1$, then b_8 is the upper bound of V_{11}, i.e. $b_{11} = b_8$. However, this case is **not applicable** because it is already derived from the above constraints that $b_8 = 255$.
2. Otherwise, the upper bound of V_8 does not satisfy the less-than-equal condition. Then the upper bound of V_{11}, b_{11} is a value between a_8 and $b_8 - 1$. Therefore the following constraints hold: $b_{11} \leq -1 \ \wedge b_{11} + 1 > -1. \ \Rightarrow \ b_{11} = -1$.

Line 4 : An equality constraint $V_{12} = V_{10}$ is created to enforce the type rule of narrowing conversion. i.e. $a_{12} = a_{10} = 0, \ b_{12} = b_{10} = 255$.

Meanwhile, the type of the narrowing cast is a subtype of the method's return type, so a subtype constraint $V_{12} <: V_{14}$ is created and encoded as $a_{14} \leq a_{12} \ \wedge \ b_{14} \geq b_{12}$. Combine the well-formedness constraint of V_{14} (whose underlying type is `byte`) $\Rightarrow$ $a_{14} = 0, b_{14} = 255$.

Therefore, the solution for the example is

$$V_8 = [-1, 255], \ V_{10} = [0, 255], \ V_{11} = [-1, -1], \ V_{12} = [0, 255], \ V_{14} = [0, 255]$$

Chapter 4

Implementation and Evaluation

In this chapter, Section 4.1 introduces the implementation details of Interval Type Inference upon Checker Framework Inference. Section 4.2 introduces the experiments performed to evaluate Interval Type Inference and a comparison between Value Range Inference and Interval Type Inference. A docker image[1] is provided to reproduce the experiments, and the implementation of Interval Type Inference is available on Github[2].

4.1 Implementation

Interval Type Inference is implemented upon Checker Framework Inference. It utilizes the basic components shown in Figure 2.1 and extends the functionality of certain components (colored in Figure 4.1). Section 4.1.1 to 4.1.5 introduce the extended components in detail. Section 4.1.6 summarizes the limitations in the implementation.

4.1.1 Tree Annotator

The Tree Annotator in Interval Type Inference extends the base type from the following aspects.

1. If the tree is not integral, add the constant annotation `@UnknownVal` to it.

[1]https://hub.docker.com/repository/docker/wongdi/artifact
[2]https://github.com/d367wang/value-inference/tree/artifact

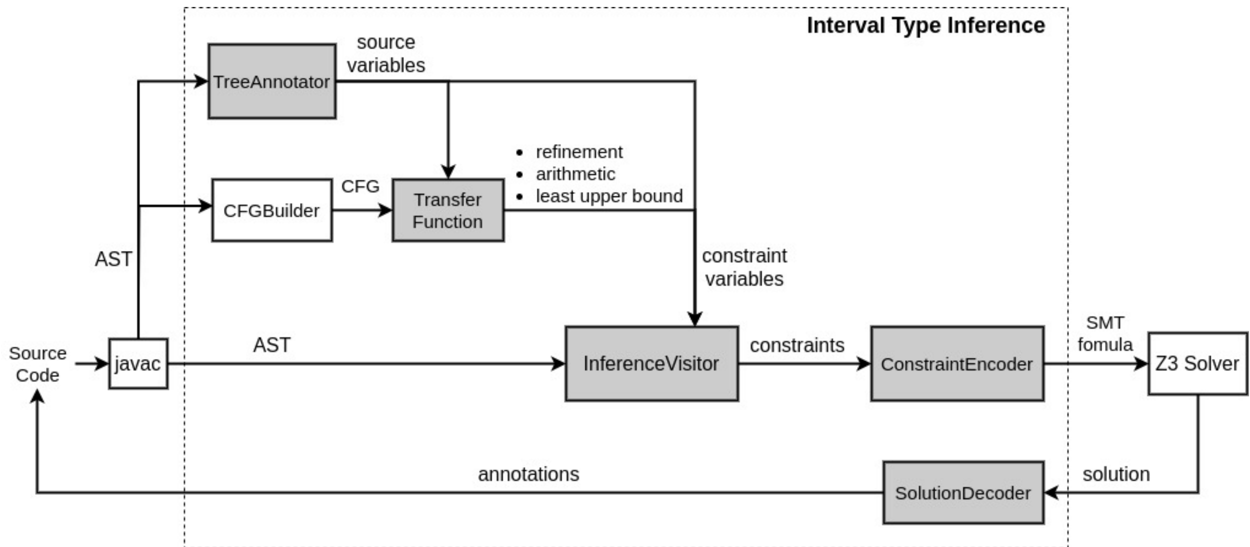

Figure 4.1: Diagram of Interval Type Inference. It is implemented upon Checker Framework Inference (Figure 2.1) by extending the components in gray color.

2. If the tree has constant integral value c (integral literal or **final static** variable), add the constant annotation `@IntRange(from=`c`, to=`c`)` to it.

3. Otherwise, call the base Tree Annotator to create a source variable.

4.1.2 Stub Classes

The Tree Annotator in Section 4.1.1 adds annotations only to source code. For fields and methods from bytecode, the annotations are specified through Java stubs.

A Java stub file allows omitting the method bodies and only listing the annotated signatures of library (e.g. JDK) methods [29, Section 34.5]. Checker Framework Inference supports stub classes as follows. When the Tree Annotator encounters a field access or method invocation which is specified in a stub file, it adds the annotations specified in the stub-version field/method declaration. If no such stub file exists, then the default annotation is applied.

The stub classes annotated in Interval Type Inference are mainly I/O libraries such as `java.io` and `java.nio`. The manual annotations are added based on the specifications in JDK documentation.

4.1.3 Transfer Function

Transfer Functions for interval are extended in the following aspects.

Transfer Function for Comparison

The transfer function first checks if the comparison operands are integral. If not, no comparison constraint is created.

We implement a lightweight tree visitor that extracts and unifies the comparison expression. When the expression is a linear expression of at most 2 variables, the transfer function for the comparison tree creates a comparison constraint variable for each variable in the expression in both of the then-branch and the else-branch. Meanwhile, a comparison constraint is created for each of the then-branch and the else-branch separately (see Section 3.3). Then the output `Store` of the transfer function is updated with the comparison constraint variable. Otherwise the transfer function passes the input `Store` to the successive nodes in the CFG, meaning no refinement is performed for any variable in the expression.

Transfer Function for Assignment

The transfer function first checks if the underlying type of the assignment expression is integral. If not, no refinement constraint is created.

4.1.4 Inference Visitor

The Inference Visitor in Interval Type Inference extends the base type by implementing the type rules regarding widening and narrowing in Section 3.2.

4.1.5 Constraint Encoder

The Constraint Encoder transforms the constraint variables and constraints to SMT-LIB2 format as the input of the Z3 solver, according to the encoding described in Section 3.4. The Constraint Encoder utilizes Z3 Java API to generate SMT-LIB2 file and then the Z3 solver is invoked.

4.1.6 Limitations

Interval Type Inference currently does not support the full features of Java language, including:

1. It does not perform flow-sensitive refinement when a comparison expression contains:
   (a) method invocations;
   (b) conditional expressions, e.g. `if (Constants.CR != (i > 0 ? buf[i - 1] : lastChar));`
   (c) postfix increment/decrement operations, e.g. `while (count++ < 10)`.
2. It does not correctly handle a comparison within a loop body, e.g.

```
int x = 0;
while (obj != null) {
    if (x > 10) {
        break;
    }
    x += 1;
}
```

   The dataflow analysis in Interval Type Inference only evaluates the first iteration of a loop. After the backward flow is merged with the flow before the loop condition `obj != null`, the dataflow analysis quits the loop and proceeds. Therefore, the update of `x` caused by the flow merge is not propagated to the comparison `x > 10`.

   A solution to this issue is to adapt the dataflow analysis to evaluate two iterations of a loop:

   (a) In the first iteration, walk through all the CFG nodes in the loop by running the corresponding transfer functions. In the case above, two comparison constraints are created at the comparison, as (V_0, V_{then}, "$x - 10$", " $>$ ") and (V_0, V_{else}, "$x - 10$", " $\leq$ "), where V_0 is the initial state of `x`, V_{then} is the abstract value in the then-branch, and V_{else} is the abstract value in the else-branch.
   (b) In the second iteration, only check each comparison within the loop body. If the abstract value of any variable in the comparison expression is changed after the flow merge at the begining of the loop condition, update the corresponding comparison constraints. In the above case, `x += 1` updates the abstract value

of `x` at the end of the first iteration, which is then merged with the flow before the loop condition. The merge causes the creation of a LUB constraint variable V_{LUB}. The LUB constraint variable is the fixed-point state of `x` at the point before `x > 10`. Therefore, the comparison constraint is updated to $(V_{LUB}, V_{then}, \text{"}x - 10\text{"}, \text{"} > \text{"})$ and $(V_{LUB}, V_{else}, \text{"}x - 10\text{"}, \text{"} \leq \text{"})$.

4.2 Evaluation

The experiments are conducted on a 64-bit Ubuntu 20.04 platform with an eight-core CPU and 16GB RAM. We run Interval Type Inference on 5 Apache Commons projects: commons-bcel, commons-crypto, commons-csv, commons-io and commons-text, all of which are chosen for uses of narrowing conversions. Interval Type Inference runs in inference mode (Section 2.2.1) in the following steps.

1. Run Interval Type Inference on the target project. If the result is SAT, collect the statistics, and the experiment on the target project is completed.
2. Otherwise, get the UNSAT core constraints and locate the conflicting constraints. Analyze the reason of the conflict, resolve it and repeat step 1.

All the 5 projects yield UNSAT in the first run. By analyzing the UNSAT core constraints, we find multiple reasons that cause UNSAT: **unsafe narrowing casts, uses of invalid input, dead branches, false positives**. Table 4.1 shows the statistics on the UNSAT causes.

Project	Unsafe Narrowing	Using Invalid Input	Dead Branch	False Positive
commons-bcel	11	1	0	5
commons-crypto	1	0	0	4
commons-csv	0	0	0	6
commons-io	1	2	1	7
commons-text	2	0	1	5

Table 4.1: The statistics on UNSAT causes

4.2.1 Unsafe Narrowing Cast

Unsafe narrowing conversions between integral types are the most common issues in the experiments. For example in commons-io, method `SwappedDataInputStream.readByte` is defined as

```
@Override
public byte readByte() throws IOException, EOFException {
    return (byte)in.read();
}
```

The interval type of `in.read()` is `@IntRange(from=-1,to=255)`. Casting it to byte without EOF test is unsafe, because both -1 and 255 have the same byte representation. According to the specification of `readByte`, an `EOFException` is thrown when the input stream reaches EOF. Therefore, a possible fix is

```
@Override
public byte readByte() throws IOException, EOFException {
    int data = in.read();
    if (data > -1) {
        return (byte)in.read();
    }
    throw new EOFException();
}
```

Another issue is found in commons-bcel. Method `Signature.matchIdent` contains the following code fragment.

```
1 final StringBuilder buf2 = new StringBuilder();
2 ch = in.read();
3 do {
4     buf2.append((char) ch);
5     ch = in.read();
6 } while ((ch != -1) && (Character.isJavaIdentifierPart((char) ch)
7         || (ch == '/')));
8 buf.append(buf2.toString().replace('/', '.'));
```

At line 2, `ch` is assigned the result of `in.read()`. Then at line 3, `ch` is directly converted to `char` and appended to the StringBuilder `buf2` without EOF test. If the value is -1, the character corresponding to 65535 is appended to `buf2`. A possible fix to this issue is to use `while` loop instead of **do...while** loop, as follows.

```
final StringBuilder buf2 = new StringBuilder();
ch = in.read();
```

```
while ((ch != -1) && (Character.isJavaIdentifierPart((char) ch)
        || (ch == '/'))){
    buf2.append((char) ch);
    ch = in.read();
}
buf.append(buf2.toString().replace('/', '.'));
```

For other issues of unsafe narrowing conversions, please refer to Appendix A.

4.2.2 Using Invalid Input

In commons-io, class `NullInputStream` extends the JDK class `InputStream` and overrides method `int read(final byte[] bytes, final int offset, final int length)`. The overriding method directly uses the input length as the output without sanity check, which may return arbitray negative integer values and violate the method specification: the return interval is [-1, $2^{31} - 1$], where -1 means the input stream reaches EOF.

A fix to this issue is to explicitly annotate the parameter `length` with `@IntRange(from=0, to=2147483647)`, which guarantees that only non-negative integer is passed for `length`. An alternative is to validate the input `length` before using it.

For other issues of uses of invalid input, please refer to Appendix B.

4.2.3 Dead Branches

For the current encoding of Interval Type Inference, the existence of a dead branch causes the constraints to be UNSAT. This is because the well-formedness constraints of comparison variables enforce the intervals after comparison are valid (not $\perp$) in both then-branch and else-branch. Therefore, if a dead branch exists, the intervals in the dead branch are $\perp$, so that the well-formedness constraints in the dead branch cannot be satisfied.

In commons-io, method `ByteArrayOutputStream.write` contains if-condition

$$(\texttt{off} < 0)||...||(\texttt{len} < 0)||...||((\texttt{off + len}) < 0)$$

According to short-circuiting, after the refinement on the first two comparison expressions `off < 0` and `len < 0`, each of `off` and `len` is refined with a comparison constraint variable whose lower bound is greater than or equal to 0. Therefore, the last comparison `off + len < 0` is always false and is redundant. The UNSAT core can be resolved by removing the redundant condition.

4.2.4 False Positives

The false-postive cases found in the experiments are caused by various reasons as follows.

Under-Constrained `Integer.parseInt`

The JDK method `Integer.parseInt` parses a given string to an integer. In some cases, the length of the input string is constant and determines the lower bound and upper bound of the result integer. A string of length 2 is capable to represent integers in [-15, 255] (from "-F" to "FF"). Figure 4.2 is a code fragment in commons-bcel. The enclosing method of the code fragment is invoked by another method, which converts the method return to `byte`.

After the refinement for `i` at line 1 and the refinement for `j` at line 3, the parsing result at line 9 is an integer in [0, 255] (from "00" to "FF"), and is safe to be cast and used as a byte. However, Interval Type Inference makes conservative assumptions that the interval of `Integer.parseInt` is $[-2^{31}, 2^{31}-1]$ and falsely issues an error.

```
1  if (((i >= '0') && (i <= '9')) || ((i >= 'a') && (i <= 'f'))) {
2      final int j = in.read();
3      if (j < 0) {
4          return -1;
5      }
6      final char[] tmp = {
7          (char) i, (char) j
8      };
9      final int s = Integer.parseInt(new String(tmp), 16);
10     return s;
11 }
```

Figure 4.2: A false-positive case caused by under-constrained `Integer.parseInt`

We can resolve these kind of false positives by introducing additional constraint variables associated with string lengths and additional constraints to express the input-output relation of `Integer.parseInt`.

Under-Constrained `Math.min/Math.max`

The `min/max` methods have implied relations between input and output. Figure 4.3 is a use case of `Math.min` in commons-io.

```
1 long remain = toSkip;
2 while (remain > 0) {
3     skipByteBuffer.position(0);
4     skipByteBuffer.limit((int) Math.min(remain, 2048));
5     final int n = input.read(skipByteBuffer);
6     if (n == EOF) {
7         break;
8     }
9     remain -= n;
10 }
```

Figure 4.3: A false-positive case caused by under-constrained `Math.min`

At line 4, the actual interval of `Math.min(remain, 2048)` is [1, 2048]. Therefore, narrowing it to `int` is safe. However, Interval Type Inference over-estimates the interval of `Math.min` to $[-2^{63}, 2^{63} - 1]$ and issues an unsafe narrowing cast, which is a false positive.

We can resolve these kind of false positives by introducing additional constraints to express the input-output relations of `Math.min` / `Math.max`.

Unconstrained Post-conditions

A false positive caused by unconstrained post-conditions is shown in Figure 4.4, which comes from commons-bcel. Not aware of the post-conditions of `Instruction.isValidByte` and `Instruction.isValidShort`, Interval Type Inference falsely issues errors at line 5 and line 7 for "unsafe narrowing conversions".

To avoid these kind of false positives, specific post-condition qualifiers should be supported. A post-condition qualifier for method declarations is proposed in [16, Chapter 3] as follows.

@EnsuresIntRangeIf(result=b, expression=e, from=l, to=u) indicates a conditional method post-condition: if the annotated method returns boolean value b, then the interval of the given expression e is $[l, u]$.

By adding the post-condition qualifier to the method `Instruction.isValidByte` at line 16, it guarantees that when the method returns true, the input is in the range [-128, 127], so that in the then branch at line 5, casting `value` to `byte` is safe. Similarly, by adding the post-condition qualifier to `Instruction.isValidShort` at line 21, casting `value` to `short` is safe at line 7.

```
1  public PUSH(final ConstantPoolGen cp, final int value) {
2      if ((value >= -1) && (value <= 5)) {
3          instruction = InstructionConst.getInstruction(Const.ICONST_0 +
               value);
4      } else if (Instruction.isValidByte(value)) {
5          instruction = new BIPUSH((byte) value);
6      } else if (Instruction.isValidShort(value)) {
7          instruction = new SIPUSH((short) value);
8      } else {
9          instruction = new LDC(cp.addInteger(value));
10     }
11 }
12
13 public abstract class Instruction implements Cloneable {
14     ...
15
16     @EnsuresRangeIf(result=true, expression="#1", from=-128, to=127)
17     public static boolean isValidByte(final int value) {
18         return value >= Byte.MIN_VALUE && value <= Byte.MAX_VALUE;
19     }
20
21     @EnsuresRangeIf(result=true, expression="#1", from=-32768, to=32767)
22     public static boolean isValidShort(final int value) {
23         return value >= Short.MIN_VALUE && value <= Short.MAX_VALUE;
24     }
25
26     ...
27 }
```

Figure 4.4: A false-positive case caused by unconstrained post-conditions. By adding post-condition qualifiers to the method `isValidByte` and `isValidShort`, the false positives at line 5 and line 7 can be avoided.

Interval Type Inference can support the above post-condition qualifier by introducing new constraint variables and constraints when a method annotated with the qualifier is invoked.

Relation Insensitivity

Built upon Checker Framework Inference, Interval Type Inference does not keep track of relations between variables. Figure 4.5 shows a false-positive case in commons-text. After the if-statement at line 10, `pos < StrBuilder.this.size()`. Then at line 14, the RHS is always positive, so at line 18, the return value `len` is always positive, which satisfies the method specification that the return value is in $[-1, 2^{31} - 1]$.

However, in Interval Type Inference, the intervals of both `pos` and `StringBuilder.this.size()` at line 14 are $[0, 2^{31}-1]$. Therefore the interval of the subtraction is $[-2^{31}+1, 2^{31}-1]$, so that at line 18, the interval of the return is $[-2^{31}+1, 2^{31}-1]$, which violates the method specification.

```
1  @Overrides
2  public int read(final char[] b, final int off, int len) {
3      if (off < 0 || len < 0 || off > b.length
4              || (off + len) > b.length || (off + len) < 0) {
5          throw new IndexOutOfBoundsException();
6      }
7      if (len == 0) {
8          return 0;
9      }
10     if (pos >= StrBuilder.this.size()) {
11         return -1;
12     }
13     if (pos + len > size()) {
14         len = StrBuilder.this.size() - pos;
15     }
16     StrBuilder.this.getChars(pos, pos + len, b, off);
17     pos += len;
18     return len;
19 }
```

Figure 4.5: A false positive case caused by missing relational analysis

A solution for such false positives is to use deductive verification tools to prove the properties regarding relations between variables, which introduces a significant specification

and verification overhead. A method to achieve a balance between scalability and precision is to combing type systems with deductive program verification approaches [30].

4.2.5 Performance

Table 4.2 shows the performance of Interval Type Inference. We measure the time consumption of different stages of the inference process. The SMT-serialization time $t_{serialize}$ represents the time to generate the SMT formulas from the constraints. The SMT-solving time t_{solve} represents the time to check the satisfiability of the SMT formulas and get a model. The total time t_{total} is the entire running time of Interval Type Inference on the target program, which is the summation of $t_{serialize}$, t_{solve} and the time consumed by other procedures including code compilation, dataflow analysis, AST traversal and constraint generation. The result shows that procedures including code compilation, dataflow analysis, AST traversal and constraint generation consume the largest proportion of the entire running time.

		Interval Type Inference		
Project	kLOC	$t_{serialize}$	t_{solve}	t_{total}
commons-bcel	30.4	2.07	15.09	161.78
commons-crypto	2.9	0.3	0.3	12.10
commons-csv	1.6	0.26	0.45	10.68
commons-io	10.0	0.99	3.77	36.09
commons-text	5.9	1.15	4.80	43.11

Table 4.2: Performance of Interval Type Inference. kLOC is the number of thousands of lines of non-comment code, counted by the tool `cloc`[4]. $t_{serialize}$ is the SMT-serialization time, t_{solve} is the SMT-solving time, and t_{total} is the total running time. All the time consumption are in seconds.

4.2.6 Comparison with Value Range Inference

The comparison between Value Range Inference and Interval Type Inference is performed from two aspects: constraint size and performance.

[4] http://cloc.sourceforge.net/

Comparison of Constraint Size

Table 4.3 and Table 4.4 compare the numbers of constraint and constraint variables between Value Range Inference and Interval Type Inference. Interval Type Inference significantly reduces the numbers of the constraint variables and constraints.

	Value Range Inference						Interval Type Inference					
Project	v_S	v_R	v_A	v_L	v_C	total	v_S	v_R	v_A	v_L	v_C	total
commons-bcel	18551	4523	908	3257	3084	30323	4958	1697	718	1150	2168	10691
commons-crypto	1834	404	113	304	304	2959	712	167	95	111	198	1283
commons-csv	891	247	67	357	306	1868	235	129	64	164	266	858
commons-io	5943	1286	556	1726	1594	11105	1702	645	451	621	964	4383
commons-text	3909	1084	624	1499	1422	5838	1557	803	585	772	1050	4787

Table 4.3: A comparison of the number of constraint variables. v_S, v_R, v_A, v_L, v_C represent the numbers of source variables, refinement variables, arithmetic variables, least-upper-bound variables and comparison variables respectively.

	Value Range Inference				Interval Type Inference			
Project	C_{sub}	C_{eq}	C_{ar}	C_{comp}	C_{sub}	C_{eq}	C_{ar}	C_{comp}
commons-bcel	43132	5241	909	1758	8151	1912	718	1296
commons-crypto	2951	608	113	254	993	196	95	156
commons-csv	2205	372	67	184	618	149	64	148
commons-io	11859	2228	556	1272	3281	828	451	672
commons-text	9911	1461	624	1068	3496	982	585	712

Table 4.4: A comparison of the number of constraints. C_{sub}, C_{eq}, C_{ar}, C_{comp} represent the numbers of subtype constraints, equality constraints, arithmetic constraints and comparison constraints respectively.

Comparison of Performance

We measure the running time of Value Range Inference and Interval Type Inference on commons-csv, as shown in Figure 4.5.

The following conclusions are drawn from the comparison.

1. $t_{serialize}$ depends on the number of the constraints in the entire program. By simplifying the interval type hierarchy, the number of constraints decreases significantly.

		Value Range Inference			Interval Type Inference		
Project	kLOC	$t_{serialize}$	t_{solve}	t_{total}	$t_{serialize}$	t_{solve}	t_{total}
commons-csv	1.6	0.84	2.73	13.66	0.26	0.46	10.68

Table 4.5: Value Range Inference v.s. Interval Type Inference on performance.

2. t_{solve} also decreases significantly, for multiple reasons including
   (a) The reduction in the number of constraints, thereby the size of SMT encoding.
   (b) Refinement is disabled for non-linear arithmetic operations.
3. The decrease in t_{total} is approximately equal to the decrease in $t_{serialize} + t_{solve}$. This means that the time consumed by dataflow analysis, AST traversal and constraint generation barely changes.

Chapter 5

Conclusion and Future Work

Interval analysis is useful in providing facts of the target program that help developers detect issues including unsafe narrowing casts, out-of-bound indices, numerical overflows/underflows, divisions-by-zero, dead branches, etc.

In this thesis, we present Interval Type Inference which improves Value Range Inference. Improvements are made from the aspects of soundness and efficiency. First, we reduce the size of the SMT encoding by simplifying the interval type hierarchy and the SMT representation of interval types. Second, we redefine type rules regarding widening/-narrowing conversions and constraints rules regarding flow-sensitive refinement, especially in the context of loops. Then we propose the SMT encoding for the constraint variables and the constraints. Finally, we perform experiments on selected open source projects and analyze the issues that are discovered by Interval Type Inference.

However the work in the domain is not finished and can be further improved in the following ways.

This thesis only focuses on using Interval Type Inference to find unsafe narrowing casts in a program. Issues such as invalid array index, overflow/underflow, division-by-0 remain to be explored. For each domain, specific type rules need to be defined.

In the encoding phase, all the constraints are encoded together into one SMT file. In the solving phase, the SMT solver exits immediately once it finds a conflict in the constraints. Therefore, Interval Type Inference can only find one issue by one run, even when there exist multiple issues in the source code that cause UNSAT. To improve the efficiency, the following approaches can be explored.

- When the solver finds an UNSAT core, remove the UNSAT core from the SMT formulas and rerun the solver iteratively, until the solver yields SAT.
- Separate constraints to multiple SMT files by certain strategies, so that parallelization is possible.

Interval Type Inference encodes constraints with the linear integer arithmetic (LIA) theory. Since computer programs are based on bounded integral data types instead of unbounded mathematical integers, the bit-vector theory is capable of modeling the program behaviors more precisely than LIA [31], especially for overflows and underflows. Therefore, the encoding of constraints to bit-vector formulas can be explored for higher precision.

According to the experiments in Section 4.2, false positives are divided into three categories based on the causes: under-constrained library methods, unconstrained post-conditions and insensitivity of relations between variables. For the first two categories, specific constraint variables and constraints are required to express the method semantics. To resolve the last category of false positives, one possible solution is to combine deductive program verification approaches [30].

This thesis only evaluates Interval Type Inference in inference mode (see Section 2.2.1). While Value Range Inference defines a set of soft constraints to support annotation mode, Interval Type Inference can incorporate such soft constraints to work in annotation mode after the satisfiability of hard constraints is proved. Further experiments are needed to evaluate the effect of the soft constraints on precision and performance.

References

[1] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Nov*, 2021.

[2] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities.* 2008.

[3] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[5] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[6] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 201–212, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 2011.

[8] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 333–357, Lancaster, UK, July 2011.

[9] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[10] Ana Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[11] Zhuo Chen. Pluggable properties for program understanding: Ontic type checking and inference. Master's thesis, University of Waterloo, 2018. http://hdl.handle.net/10012/13181.

[12] Jianchu Li. A general pluggable type inference framework and its use for data-flow analysis. Master's thesis, University of Waterloo, 2017. http://hdl.handle.net/10012/11771.

[13] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA 2012, Object-Oriented Programming Systems, Languages, and Applications*, pages 879–896, Tucson, AZ, USA, October 2012.

[14] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2012.

[15] David Erni and Adrian Kuhn. The hacker's guide to javac. 2008.

[16] Tongtong Xiang. Type checking and whole-program inference for value range analysis. Master's thesis, University of Waterloo, 2020. http://hdl.handle.net/10012/16445.

[17] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation (PLDI)*, June 2011.

[18] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for controlling UI object access. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2013.

[19] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Computer and Communications Security (CCS)*, November 2014.

[20] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679, 2015.

[21] Daniel Scott Brotherston, Werner Dietl, and Ondrej Lhoták. Granullar: Gradual nullable types for Java. *Proceedings of the 26th International Conference on Compiler Construction*, 2017.

[22] Weitian Xing, Yuanhui Cheng, and Werner Dietl. Ensuring correct cryptographic algorithm and provider usage at compile time. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, pages 43—50, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, page 20–26, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Charles Zhuo Chen and Werner Dietl. Don't miss the end: Preventing unsafe end-of-file comparisons. In *NASA Formal Methods*, pages 87–94, Cham, 2018. Springer International Publishing.

[25] Checker Framework Organization. A dataflow framework for Java. https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf, 2021.

[26] Nikolaj Bjørner and Anh-Dung Phan. νZ - maximal satisfaction with Z3. In *International Symposium on Symbolic Computation in Software Science(SCSS)*, 2014.

[27] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[28] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[29] Checker Framework Organization. The checker framework manual: Custom pluggable types for Java. https://checkerframework.org/manual/, 2021.

[30] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[31] Sanu Subramanian, Murphy Berzish, Yunhui Zheng, Omer Tripp, and Vijay Ganesh. A solver for a theory of strings and bit-vectors, 2016.

APPENDICES

Appendix A

Unsafe Narrowing Conversions in Selected Projects

commons-bcel[1]

1. In src/main/java/org/apache/bcel/classfile/LineNumber.java

```
1     public LineNumber(final int start_pc, final int line_number
          ) {
2         this.start_pc = (short) start_pc;
3         this.line_number = (short)line_number;
4     }
```

   This constructor is called by another constructor

```
      LineNumber(final DataInput file) throws IOException {
          this(file.readUnsignedShort(), file.readUnsignedShort()
              );
      }
```

   `file.readUnsignedShort()` is unsigned short, therefore the input is in [0, 65535]

   The first constructor is also called in src/main/java/org/apache/bcel/generic/LineNumberGen.java

```
      public LineNumber getLineNumber() {
          return new LineNumber(ih.getPosition(), src_line);
      }
```

[1]https://github.com/opprop-benchmarks/commons-bcel

`ih.getPosition()` is possible to be -1. Therefore, the input `start_pc` has a possible interval of [-1, 65535], and it is unsafe to directly convert `start_pc` to `short`.

2. In src/main/java/org/apache/bcel/classfile/Signature.java :

```
1  private static void matchIdent( final MyByteArrayInputStream in
       , final StringBuilder buf ) {
2      ...
3      final StringBuilder buf2 = new StringBuilder();
4      ch = in.read();
5      do {
6         buf2.append((char) ch);
7         ch = in.read();
8      } while ((ch != -1) && (Character.isJavaIdentifierPart((
          char) ch) || (ch == '/')));
9      buf.append(buf2.toString().replace('/', '.'));
10     if (ch != -1) {
11        in.unread();
12     }
13 }
```

In the first iteration of the do-while loop, at line 6, `ch` (refined by the result of `in.read()` is directly converted to `char`, which is unsafe.

3. In src/main/java/org/apache/bcel/classfile/Signature.java :

```
1  private static void matchGJIdent( final MyByteArrayInputStream
       in, final StringBuilder buf ) {
2      ...
3      ch = in.read();
4      if (identStart(ch)) {
5          in.unread();
6          matchGJIdent(in, buf);
7      } else if (ch == ')') {
8          in.unread();
9          return;
10     } else if (ch != ';') {
11         throw new RuntimeException("Illegal signature: " + in.
              getData() + " read " + (char) ch);
12     }
13     ...
14 }
```

In the throw-statement at line 11, `ch` is converted to `char` without EOF test.

4. In src/main/java/org/apache/bcel/classfile/Utility.java :

```
1 @Override
2 public int read( final char[] cbuf, final int off, final int
      len ) throws IOException {
3     for (int i = 0; i < len; i++) {
4         cbuf[off + i] = (char) read();
5     }
6     return len;
7 }
```

The return of `read()` is directly converted to `char`, while `read()` is possibly equal to -1, as its definition is

```
1 @Override
2 public int read() throws IOException {
3     final int b = in.read();
4     if (b != ESCAPE_CHAR) {
5         return b;
6     }
7     final int i = in.read();
8     if (i < 0) {
9         return -1;
10    }
11    if (((i >= '0') && (i <= '9')) || ((i >= 'a') && (i <= 'f'))
          ) { // Normal escape
12        final int j = in.read();
13        if (j < 0) {
14            return -1;
15        }
16        final char[] tmp = {
17                (char) i, (char) j
18        };
19        final int s = Integer.parseInt(new String(tmp), 16);
20        return s;
21    }
22    return MAP_CHAR[i];
23 }
```

Line 9 and line 14 return -1.

5. In src/main/java/org/apache/bcel/generic/LOOKUPSWITCH.java :

```
1      public LOOKUPSWITCH(final int[] match, final
           InstructionHandle[] targets, final InstructionHandle
           defaultTarget) {
```

```
2            super(org.apache.bcel.Const.LOOKUPSWITCH, match,
                targets, defaultTarget);
3            /* alignment remainder assumed 0 here, until dump time.
                 */
4            final short _length = (short) (9 + getMatch_length() *
                8);
5            super.setLength(_length);
6            setFixed_length(_length);
7        }
```

At line 4, getMatch_length() is the length of the input array match. (short) (9 + getMatch_length() * 8) is not safe.

In another method

```
1        @Override
2        protected void initFromFile( final ByteSequence bytes,
             final boolean wide ) throws IOException {
3            super.initFromFile(bytes, wide); // reads padding
4            final int _match_length = bytes.readInt();
5            setMatch_length(_match_length);
6            final short _fixed_length = (short) (9 + _match_length
                * 8);
7            setFixed_length(_fixed_length);
8            final short _length = (short) (_match_length + super.
                getPadding());
9            super.setLength(_length);
10           super.setMatches(new int[_match_length]);
11           super.setIndices(new int[_match_length]);
12           super.setTargets(new InstructionHandle[_match_length]);
13           for (int i = 0; i < _match_length; i++) {
14               super.setMatch(i, bytes.readInt());
15               super.setIndices(i, bytes.readInt());
16           }
17       }
```

At line 4, bytes.readInt() may return arbitrary integer, which is assigned to _match_length. Therefore, (short)(9 + _match_length * 8) at line 6 and (short) (_match_length + super.getPadding()) at line 8 are not safe.

6. In src/main/java/org/apache/bcel/generic/Select.java :

```
1  @Override
2      protected int updatePosition( final int offset, final int
           max_offset ) {
```

```
3           setPosition(getPosition() + offset); // Additional
                offset caused by preceding SWITCHs, GOTOs, etc.
4           final short old_length = (short) super.getLength();
5           /* Alignment on 4-byte-boundary, + 1, because of tag
                byte.*/
6           padding = (4 - ((getPosition() + 1) % 4)) % 4;
7           super.setLength((short) (fixed_length + padding)); //
                Update length
8           return super.getLength() - old_length;
9       }
```

At line 7, `fixed_length` may be arbitrary integer. `(short)(fixed_length+padding)` is not safe.

7. In src/main/java/org/apache/bcel/generic/TABLESWITCH.java :

```
1       public TABLESWITCH(final int[] match, final
            InstructionHandle[] targets, final InstructionHandle
            defaultTarget) {
2           super(org.apache.bcel.Const.TABLESWITCH, match, targets
                , defaultTarget);
3           /* Alignment remainder assumed 0 here, until dump time
                */
4           final short _length = (short) (13 + getMatch_length() *
                 4);
5           super.setLength(_length);
6           setFixed_length(_length);
7       }
```

At line 4, `getMatch_length()` is the length of the input array `match`. `(short) (13 + getMatch_length() * 4)` is not safe.

In another method

```
1   @Override
2       protected void initFromFile( final ByteSequence bytes,
            final boolean wide ) throws IOException {
3           super.initFromFile(bytes, wide);
4           final int low = bytes.readInt();
5           final int high = bytes.readInt();
6           final int _match_length = high - low + 1;
7           setMatch_length(_match_length);
8           final short _fixed_length = (short) (13 + _match_length
                 * 4);
9           setFixed_length(_fixed_length);
```

```
10          super.setLength((short) (_fixed_length + super.
                getPadding()));
11          super.setMatches(new int[_match_length]);
12          super.setIndices(new int[_match_length]);
13          super.setTargets(new InstructionHandle[_match_length]);
14          for (int i = 0; i < _match_length; i++) {
15              super.setMatch(i, low + i);
16              super.setIndices(i, bytes.readInt());
17          }
18      }
```

At line 6, `_match_length` may be assigned an arbitrary integer. Therefore, `(short)(13+_match_length*4)` at line 8 and `(short)(_fixed_length+super.getPadding())` at line 10 are not safe.

commons-crypto[2]

1. In src/main/java/org/apache/commons/crypto/stream/CtrCryptoInputStream.java:

```
1  public void seek(long position) throws IOException {
2      Utils.checkArgument(position >= 0, "Cannot seek to negative
           offset.");
3      checkStream();
4      if (position >= getStreamPosition() && position <=
           getStreamOffset()) {
5          int forward = (int) (position - getStreamPosition());
6          if (forward > 0) {
7              outBuffer.position(outBuffer.position() + forward);
8          }
9      } else {
10         input.seek(position);
11         resetStreamOffset(position);
12     }
13 }
```

The input `position` can be arbitrary long integer. Therefore, at line 5, `(int)(position - getStreamPosition())` is not safe.

commons-io[3]

1. In src/main/java/org/apache/commons/io/input/SwappedDataInputStream.java :

[2]https://github.com/opprop-benchmarks/commons-crypto
[3]https://github.com/opprop-benchmarks/commons-io

```
1 @Override
2 public byte readByte()
3     throws IOException, EOFException
4 {
5     return (byte)in.read();
6 }
```

`in.read()` returns a value in [-1, 255]. It is unsafe to cast it to byte without EOF test.

commons-text[4]

1. In src/main/java/org/apache/commons/text/translate/UnicodeUnescaper.java :

```
1  @Override
2  public int translate(final CharSequence input, final int index,
        final Writer out) throws IOException {
3      if (input.charAt(index) == '\\' && index + 1 < input.length
           () && input.charAt(index + 1) == 'u') {
4          // consume optional additional 'u' chars
5          int i = 2;
6          while (index + i < input.length() && input.charAt(index
               + i) == 'u') {
7              i++;
8          }
9
10         if (index + i < input.length() && input.charAt(index + i
               ) == '+') {
11             i++;
12         }
13
14         if (index + i + 4 <= input.length()) {
15             // Get 4 hex digits
16             final CharSequence unicode = input.subSequence(index
                   + i, index + i + 4);
17
18             try {
19                 final int value = Integer.parseInt(unicode.
                       toString(), 16);
20                 out.write((char) value);
21             } catch (final NumberFormatException nfe) {
```

[4] https://github.com/opprop-benchmarks/commons-text

```
22                  throw new IllegalArgumentException("Unable to
                        parse unicode value: " + unicode, nfe);
23              }
24              return i + 4;
25          }
26          throw new IllegalArgumentException("Less than 4 hex
                digits in unicode value: '"
27                  + input.subSequence(index, input.length())
28                  + "' due to end of CharSequence");
29      }
30      return 0;
31  }
```

In the try block, at line 19 a string of 4 characters are parsed to an integer `value`, and then `value` is converted to `char`. When the input is parsed to a negative integer (e.g. `input` is `\\u-FFF`), this is unsafe.

2. In src/main/java/org/apache/commons/text/AlphabetConverter.java :

```
1  private static String codePointToString(final int i) {
2      if (Character.charCount(i) == 1) {
3          return String.valueOf((char) i);
4      }
5      return new String(Character.toChars(i));
6  }
```

The JDK method `Character.charCount` does not validate the specified character to be a valid Unicode code point, and it returns 1 when `i < 0`. So when the method `codePointToString` is passed an negative integer, the if-condition is true. Then `i` is converted to `char` and passed to `String.valueOf`. This method is called in the following method.

```
1  public static AlphabetConverter createConverterFromMap(
2          final Map<Integer, String> originalToEncoded) {
3      final Map<Integer, String> unmodifiableOriginalToEncoded =
4              Collections.unmodifiableMap(originalToEncoded);
5      final Map<String, String> encodedToOriginal = new
           LinkedHashMap<>();
6
7      int encodedLetterLength = 1;
8
9      for (final Entry<Integer, String> e
10             : unmodifiableOriginalToEncoded.entrySet()) {
11         final String originalAsString = codePointToString(e.
               getKey());
```

```
12          encodedToOriginal.put(e.getValue(), originalAsString);
13
14          if (e.getValue().length() > encodedLetterLength) {
15              encodedLetterLength = e.getValue().length();
16          }
17      }
18
19      return new AlphabetConverter(unmodifiableOriginalToEncoded,
20              encodedToOriginal,
21              encodedLetterLength);
22  }
```

An input that may cause unexpected behaviors is a HashMap that contains a K-V pair (-1, “xyz”).

Appendix B

Uses of Invalid Input in Selected Projects

commons-bcel

1. In src/main/java/org/apache/bcel/classfile/Utility.java :

```
1  public static byte[] decode(final String s, final boolean
       uncompress) throws IOException {
2      byte[] bytes;
3      try (JavaReader jr = new JavaReader(new CharArrayReader(s.
           toCharArray()));
4              ByteArrayOutputStream bos = new
                   ByteArrayOutputStream()) {
5          int ch;
6          while ((ch = jr.read()) >= 0) {
7              bos.write(ch);
8          }
9          bytes = bos.toByteArray();
10     }
11     ...
12 }
```

`jr.read()` returns a value in [-1, 65535]. After the comparison `(ch = jr.read()) >= 0`, in the then-branch, `ch` is refined to [0, 65535]. `bos.write()` takes an argument in [0, 255]. Passing `ch` as the argument violates the specification and may cause data loss.

commons-io

1. In src/main/java/org/apache/commons/io/input/NullInputStream.java :

```
1  public int read(final byte[] bytes, final int offset, final int
       length) throws IOException {
2      if (eof) {
3          throw new IOException("Read after end of file");
4      }
5      if (position == size) {
6          return doEndOfFile();
7      }
8      position += length;
9      int returnLength = length;
10     if (position > size) {
11         returnLength = length - (int)(position - size);
12         position = size;
13     }
14     processBytes(bytes, offset, returnLength);
15     return returnLength;
16 }
```

Input `length` can be a negative integer. At line 8, `length` is added to `position` without validating.

2. In src/main/java/org/apache/commons/io/output/ProxyOutputStream.java :

```
1  @Override
2  public void write(final byte[] bts, final int st, final int end
      ) throws IOException {
3       try {
4           beforeWrite(end);
5           out.write(bts, st, end);
6           afterWrite(end);
7       } catch (final IOException e) {
8           handleIOException(e);
9       }
10 }
```

The argument `end` of method invocation `beforeWrite(end)` can be a negative integer, while `beforeWrite(end)` is defined in the subclass `CountingOutputStream` as

```
1  @Override
2  protected synchronized void beforeWrite(final int n) {
```

```
3     count += n;
4 }
```

`count` represents the number of bytes that are being written and should not add a negative integer.