Type Checking and Whole-program Inference for Value Range Analysis

by

Tongtong Xiang

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, 2020

Author's Declaration

Statement of contributions

Tongtong Xiang is the sole author for Chapters 1, 3, 4, and 6. They are written under the supervision of Professor Werner Dietl, and are not yet published.

Chapter 5 of the thesis is part of the publication for the OOPSLA 2020 paper, *Precise Inference of Expressive Units-of-Measurement Types* [46]. It is co-authored with Jeff Yucong Luo and Werner Dietl. Chapter 2, background, and Chapter 7, conclusion and future work, also contain pieces from the paper.

Abstract

Value range analysis is important in many software domains for ensuring the safety and reliability of a program and is a crucial facet in software development. The resulting information can be used in optimizations such as redundancy elimination, dead code elimination, instruction selection, and improve the safety of programs. This thesis explores the use of static analysis with type systems for value range analysis. Properly formalized type systems can provide mathematical guarantees for the correctness of a program at compile time. This thesis presents (1) a novel type system, the Narrowing and Widening Checker, (2) a whole-program type inference, the Value Inference for Integral Values, (3) a units-of-measurement type system, *PUnits*, and (4) an improved algorithm to statically analyze the data-flow of programs.

The Narrowing and Widening Checker is a type system that prevents loss of information during narrowing conversion of primitive integral data types and automatically distinguishes the signedness of variables to eliminate the ambiguity of a widening conversion from type `byte` and `short` to type `int`. This additional type system ensures soundness in programs by restricting operations that violate the defined type rules.

While type checking verifies whether the given type declarations are consistent with their use, type inference automatically finds the properties at each location in the program, and reduces the annotation burden of the developer. The Value Inference for Integral Values is a constraint-based whole-program type inference for integral analysis. It supports the relevant type qualifiers used by the Narrowing and Widening type system, and reduces the annotation burden when using the Narrowing and Widening Checker. Value Inference can infer types in two modes: (1) ensure a valid integral typing exists, and (2) annotate a program with precise and relevant types. Annotation mode allows human inspection and is essential since having a valid typing does not guarantee that the inferred specification expresses design intent.

PUnits is a type system for expressive units of measurement types and a precise, whole-program inference approach for these types. This thesis presents a new type qualifier for this type system to handle cases where the method return and method parameter type are context-sensitive to the method receiver type. This thesis also discusses the related work and the benefits and trade-offs of using *PUnits* versus existing Java unit libraries, and demonstrates how *PUnits* can enable Java developers to reap the performance benefits of using primitive types instead of abstract data types for unit-wise consistent scientific computations in real-world projects.

The Dataflow Framework is a data-flow analysis for Java used to evaluate the values at each program location. Data-flow analysis is considered a terminating, imprecise abstract interpretation of a program and many false-positives are issued by the Narrowing and Widening Checker due to its imprecision. Three improvements to the algorithm in the framework are presented to increase the precision of the analysis: (1) implementing a dead-branch analysis, (2) proposing a path-sensitive analysis, and (3) discussing how loop precision can be improved.

The Narrowing and Widening Checker is evaluated on 22 of the Apache Commons projects with a total of 224k lines of code. Out of these projects, 18 projects failed with 717 errors. The Value Inference for Integral Values is evaluated on these 18 Apache Commons projects. Out of these projects, 5 projects are successfully evaluated to SAT and the Value Inference inferred 10639 annotations. The 13 projects that are evaluated to UNSAT are manually examined and all of them contain a real narrowing error. Manual annotations are added to 5 of these projects to resolve the reported errors. In these 5 projects, the Narrowing and Widening Checker detects 69 real errors and 26 false-positives, with a false-positive rate of 37.7%. The type system performs adequately with a compilation time overhead of 5.188x for the Narrowing and Widening Checker and 24.43x for the Value Inference. These projects are then evaluated with the addition of dead-branch analysis to the framework; the additional evaluation time is negligible. Its performance is suitable for use in a real-world software development environment.

All the presented type systems build on techniques from type qualifier systems and constraint-based type inference. Our implementation and evaluation of these type systems show that these techniques are necessary and are effective in ensuring the correctness of real-world programs.

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 Richard Trefler, for their time and feedback. I am thankful to all my colleges in our research group, especially Jeff Y. Luo, Weitian Xing, and Lian Sun, for their help and bringing joy to these two years of my study. I would also like to thank my boyfriend, for taking care of my well-being and daily matters.

Dedication

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

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ensuring the safety and reliability of a program is a crucial facet in software development. To achieve this, sophisticated and elaborated processes and methods are presented and implemented. This includes static analysis [2] and type systems [11]. Static program analysis is the analysis of a computer program without execution of the program, as opposed to dynamic analysis, which analyzes the program by performing executions [38]. A type system is a logical system that assigns properties and defines type rules between different parts of a computer program, such as variables, expressions, functions, or modules. Properly formalized type systems can provide mathematical guarantees for the correctness of a program at compile time. The type properties in a type system can be checked and inferred. Type checking verifies whether the given type declarations are consistent with their use. Type inference can automatically define the properties at each location in the computer program, and reduce annotation burden on the developer.

Java is one of the most popular programming languages today and is widely used in many places, from Android and web applications to scientific and financial computations. Java's built-in type system can already prevent various type errors. For example, Java will issue a type incompatible error for expression `String x = 0`, since x has type `String` and `0` has type `int`. The Java type system forbids direct assignment of a variable with a type `int` to a variable with a type `String` and vice versa. Nevertheless, Java's type system has its limitations. As a basic example, Java's built-in type system cannot prevent the possible occurrence of a null pointer exception.

The Checker Framework [41, 18] is a pluggable type system that enhances Java's type system and makes it "more powerful and useful". This pluggable type system framework allows additional type systems to be designed and 'plugged-in' to the existing compiler

```
1 void foo(@NonNull Object nn, @Nullable Object nbl) {
2     nn.toString();        // OK
3     nbl.toString();       // Error
4     if (nbl != null) {
5         nbl.toString(); // OK
6     }
7 }
```

```
Error: [dereference.of.nullable] dereference of possibly-null
   reference nbl
        nbl.toString();
           ^
```

Figure 1.1: Example on how the Nullness type system prevents possible null pointer exceptions. An `dereference.of.nullable` error is issued on line 3.

infrastructure to further improve program reliability. These additional type systems ensure soundness in programs by restricting operations in which violate the type rules defined.

For example, the Nullness type system implemented using the Checker Framework can effectively prevent the occurrence of null pointer exceptions during program execution. Fig. 1.1 shows how null pointer exceptions are prevented using the Nullness type system. The parameters *nn* and *nbl* are annotated with `@NonNull` and `@Nullable`, respectively. Those annotations are the additional property provided by the type system. `@NonNull` means that this variable will never be null. Therefore the dereference of *nn* on line 2 is always valid. `@Nullable` means that the variable can be a null value. Therefore, the compiler issues a dereference of nullable error on line 3 as the type rule of the Nullness type system forbids the dereferencing of a `@Nullable` type. Variables with type `@Nullable` should perform a nullness check before dereferencing for program safety.

Other than the Nullness type system, various other type systems are built using the Checker Framework. This thesis focuses on the design and implementation of two novel type systems that are built using the Checker Framework, the Narrowing and Widening Type System, and the Value Inference Type System. Both of these type systems are built on top of the Constant Value type system, and can be viewed as an enhancement to the type system. This thesis will also discuss the improvements and evaluations of *PUnits* [46], a units-of-measurement type system built together with Ph.D. student Jeff Yucong Luo.

The Constant Value type system tries to determine the primitive values, box primitive values, string values, and array lengths of each variable at compilation time. Knowing the value of variables at compiler time can be useful to prevent many common run-time exceptions such as the index out of bound exception [30], division by zero exception, overflow exception, etc. It is also useful to prevent unsafe codings that are not easily detectable, such as enforcing cryptography functions to only accept certain key strings or length[1]. This thesis presents the Narrowing and Widening Checker, a novel type system that builds on top of the Constant Value type system and prevents unsafe narrowing and widening conversions between Java's primitive types [23]. A conversion is considered unsafe when there is a loss or ambiguity of information during narrowing and widening. It is inspired by the EOF Value Checker [14].

Ensuring computations are unit-wise consistent is an important task in software development for scientific, engineering, and business domains. Performing calculations with mismatched dimensions (e.g., length, time, speed, etc.) or units (e.g., meters, millimetres, seconds, meter per second, etc.) can result in catastrophic failures. The Mars Climate Orbiter disintegrated in 1998 on its approach to Mars due to one software component communicating values in Imperial units, while the receiving component expected values in International System (SI) units [7]. This mismatch cost US taxpayers $327.6 million USD. For efficiency reasons, such computations are usually performed with primitive types instead of abstract data types, which results in very weak static guarantees about correct usage and conversion of units. This thesis makes improvements on the Unit type system and demonstrates how this type system can enable Java developers to reap the performance benefits of using primitive types instead of abstract data types for unit-wise consistent scientific computations in real-world projects.

Manually adding type qualifiers to every location of the computer program can be a burden. To reduce annotation burden for the developers, a type inference framework, Checker Framework Inference [20], has been presented. The Checker Framework Inference is a whole-program type inference that uses a constraint-based analysis approach. Constraints are generated over the program locations and are collected for a solver. If a solution exists, then a valid typing exists for the program. Optionally, these solutions can be inserted back into the source code to allow human inspection and provide well-specified applications and libraries.

The whole-program type inference is not implemented for all the type systems since the annotation burden and the necessity for human inspection for some type systems may be low. For some type system where we have localized information, like regex [43], inference is not necessary. The Units type system on the other hand, wants to provide well-specified applications and libraries and thus, has a type inference implemented. The Constant Value

[1]AWS KMS Compliance Checker: https://github.com/awslabs/aws-kms-compliance-checker

type system currently does not have a type inference implemented, but due to its wide uses, having a whole-program type inference for this type system can be useful, especially for numeric variables. This thesis introduces the Integral Value Inference, a type system that implements a whole-program inference for integral values in the program.

The Checker Framework uses the Dataflow Framework, a data-flow analysis for Java [40], to estimate the values during compile time. Data-flow analysis is a static analysis technique that is flow-sensitive and is guaranteed to terminate. However, it is considered an imprecise abstract interpretation of a program as this analysis is path-insensitive – it does not depend on the predicates at conditional branch instructions. This thesis introduces two enhancements to the precision of the analysis done by the Dataflow Framework, a dead-branch analysis and a path-sensitive analysis, and discusses how they can be used to improve loop precision.

The focuses of this thesis is on value range analysis, where the resulting information can be used in optimizations such as redundancy elimination, dead code elimination, instruction selection, and improve the safety of programs. The motivation of this thesis is to enhance the analysis and the pluggable type systems implemented by the Checker Framework and demonstrate their effectiveness in ensuring program correctness for real-world projects. This thesis makes contributions in the following areas: the thesis

- presents the Narrowing and Widening Checker, a type checker that prevents loss of information during narrowing conversion and automatically distinguishes the signedness of variables to eliminate the ambiguity of a widening conversion.

- presents a whole-program inference for integral analysis and reduces the annotation burden when using the Narrowing and Widening Checker.

- introduces the comparison constraint, a constraint for the refinement of variable types after a boolean expression.

- adds receiver-dependent types to the Unit type system to further minimize the need for unit annotations in method bodies and remove unnecessary clutter.

- discusses the related works and the benefits and trade-offs of using the Units Checker versus existing Java unit libraries, while demonstrating how *PUnits* enables developers to reap the performance benefits of using primitive types instead of abstract data types for unit-wise consistent scientific computations in real-world projects.

- improves the algorithm in the framework to increase the precision of the analysis by implementing a dead-branch analysis, proposing a path-sensitive analysis, and discussing loop precision improvements.

This thesis is organized as follows:

Chap. 2 discusses the background on the Dataflow Framework, Checker Framework, Checker Framework Inference, and introduces the Constant Value type system and the Units type system. Chap. 3 introduces the Narrowing and Widening Checker, which is built on top of the Constant Value type system. Chap. 4 introduces Integral Value Inference, which is influenced by the Constant Value type system. Chap. 5 discusses the improvement to the Units type system and the benefits and trade-offs of using the Units Checker versus existing Java unit libraries. Chap. 6 introduces the improvement to the Dataflow Framework for precision. Finally, Chap. 7 concludes the whole thesis and discusses future work.

Chapter 2

Background and Related Work

2.1 Dataflow Framework for Java

Data-flow analysis is a flow-sensitive analysis that predicts the flow of information through the locations of a program. It is considered a terminating, imprecise abstract interpretation of a program. Even though it is imprecise, having an analysis that is decidable is important, e.g. for compiler optimization. The Dataflow Framework is a data-flow analysis framework for Java programs that estimate the values a variable might contain. The java program is portrayed as a control-flow graph (CFG) in the Dataflow Framework. The Dataflow Framework transforms the abstract syntax tree (AST) generated from an input Java program to its CFG interpretation. Then the analysis is applied to the CFG iteratively until the abstract values reach a fix-point.

The Dataflow Framework contains the following main modules [40]:

- **Node**: represents an individual operation of a program. It is a minimal Java operator or expression.
- **Block**: consists of groups of nodes. There are four types of blocks: regular block, exception block, conditional block, and a special block. Conditional block represents all non-exceptional control-flow splits, or branches, and contains no nodes.
- **Abstract Value**: is the internal representation of data-flow information computed by the specific analysis. It is an abstract representation of the values during an execution.

- **Store**: is a mapping of nodes to their abstract values.
- **Transfer Function**: takes in a transfer input and produces a transfer result. Each node has its transfer function. The transfer input and the transfer result may contain a single store or a pair of `then` and `else` stores. A Boolean-valued expression produces a result that contains both a `then` and an `else` store, other Nodes produces a result with a single store.
- **Analysis**: performs the iterative data-flow analysis over the control flow graph blocks.

Fig. 2.1 is an example program and the corresponding control flow graph generated by the Dataflow Framework of that program. Special blocks, entry and exit, are represented using circles. The regular blocks are represented using rectangles and the conditional blocks are represented using an octagon. The regular blocks contain the sets of nodes that make up the program. The type of node is indicated in [].

If there is one store flowing into a conditional block, then it is duplicated to both successors. If two stores are flowing into a conditional block, the `then` store is propagated to the block's `then` successor and the `else` store is propagated to the block's `else` successor.

Control flow split occurs in if, for, while, and switch statements. A transfer result is produced containing two stores, `then` and `else` stores, after certain boolean-valued expressions. The `then` and an `else` store is always produced by the transfer function after a boolean expression. The `then` store will flow to the `then` branch of the control flow graph and the `else` store will flow to the `else` branch of the control flow graph. After the branching, the two stores are merged back together. Merge means taking the least-upper-bound of the two stores.

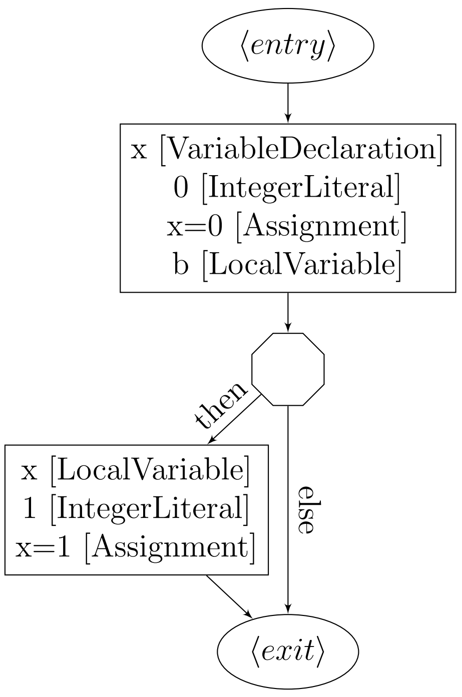
2.2 Checker Framework and Inference

The Checker Framework provides the abstract values, rules, various options, etc. for the analysis done by the Dataflow Framework. To develop a type system using the Checker Framework [1], the developer is required to define the five main implementation components of a type system:

[1]https://checkerframework.org/manual/#creatingachecker

```java
int foo(boolean b) {
      int x = 0;
      if (b) {
            x = 1;
      }
}
```

(a) Example program



(b) CFG of `foo`

Figure 2.1: A simple example with its CFG produced by the Dataflow Framework. Special blocks are represented using circles. Regular blocks are represented using rectangles. Conditional blocks are represented using octagon. The regular block consists set of nodes. The type of node is indicated in [].

- **Type Qualifier**: defines the types in the type system. Usually, a type system contains a $\top$ and a $\bot$ indicating the least upper bound and the greatest lower bound of all the types in that type system.
- **Type Hierarchy**: defines the typing relationships between the type qualifiers.
- **Type Introduction Rules**: specifies the type qualifiers at each node or location in the program. The qualifiers at these locations can be explicitly annotated by the developer, implicitly defined by the type system, or inferred by the type inference.
- **Type Rules**: defines the type system's semantics and yields type errors and warnings if violated.
- **Type Refinement Rules**: defines how type qualifiers are refined during flow-sensitive type refinement when evaluating the program using Dataflow Framework. Usually, the type is defaulted to $\top$ and then refined to a more precise subtype.
- **Compiler Interface**: specifies the type qualifiers and the options supported by the type system.

Developers can provide partially-annotated source code and specifications for library code as input. Type inference alleviates the manual annotation effort for the developer. The Checker Framework Inference provides a constraint-based whole-program inference framework for type qualifiers provided by the Checker Framework [36]. Soundness is enforced by generating and solving typing constraints through syntax-guided constraint generation rules. Checker Framework Inference currently supports the following constraints:

- Subtype constraint $T_a <: T_b$: enforces T_a to be the same type or a subtype of T_b. They are usually used for assignments, parameter passing, result passing, and type variable bound checks.
- Equality Constraint $T_a = T_b$: ensures T_a and T_b are the same. They are usually used to handle method overriding, type refinement, and type argument.
- Inequality Constraint $T_a \neq T_b$: ensures T_a and T_b are not the same.
- Comparable Constraint $T_a <:> T_b$: ensures T_a and T_b are comparable. These constraints are usually used for casts or comparison operations.

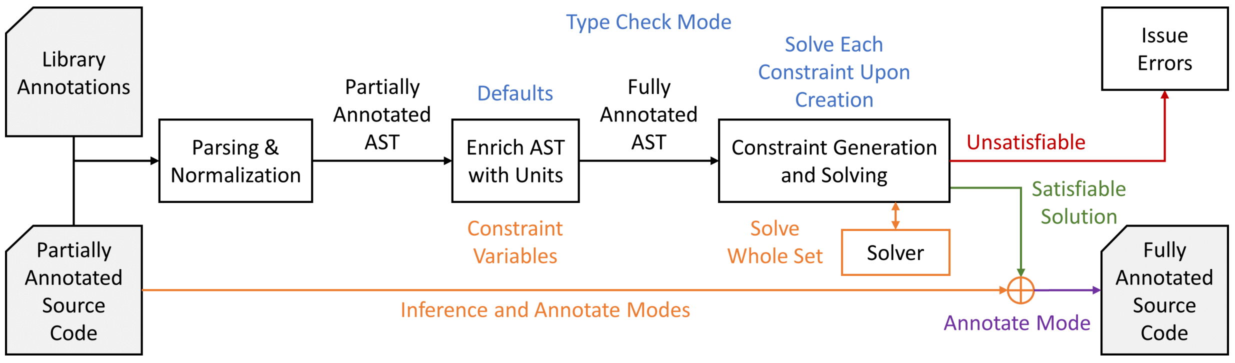


Figure 2.2: Type checking and inference process with three different modes: 1) Modular type checking using defaults and eager constraint solving; 2) Whole-program type inference to ensure valid typing; and 3) Whole-program type inference with additional breakable constraints to annotate code with the most precise types. [46]

- Combination Constraint $T_c = T_a \triangleright T_b$: ensures that the viewpoint adaptation of T_b from the viewpoint expressed by T_a results in T_c. They are usually used for receiver dependent variables.
- Arithmetic Constraint $T_c = T_a \; op \; T_b$: enforces that α will be equal to the result of the arithmetic computation of $T_a \; op \; T_b$, for some defined arithmetic operation op.
- Preference Constraint $T_a \approx T_b$: prefers the solution for T_a be the same as T_b when possible. These constraints are breakable.

A user of Checker Framework Inference can pick between one of three modes: 1) modular type checking, 2) whole-program type inference to ensure valid typing, and 3) whole-program type inference to annotate the program with the most precise typing solution. The modes are respectively given the short names of type check mode, inference mode, and annotate mode. Fig. 2.2 illustrates the overall type checking and type inference process.

2.2.1 Modular Type Checking

Modular type check mode allows methods and classes to be type-checked independently. It gives developers quick feedback on potential errors, with a method-local view of potential problems.

All type systems use defaults for missing type qualifiers. The defaults are chosen to give errors across method and class boundaries, allowing developers to quickly catch potential problems and focus on fixing one method or class at a time. A set of mandatory constraints is generated by syntax-guided constraint generation rules and follows the type system semantics. In type check mode, each constraint is solved upon creation. Unsatisfied constraints cause compilation errors.

Type check mode gives method-local views of potential errors, some of which may be due to a lack of annotations in the code.

2.2.2 Whole-program Type Inference

Whole-program type inference mode ensures a valid typing exists for a program and, optionally, infers the most precise units for a program using annotation mode. Inference mode pinpoints the set of code locations across the program that together could cause a type inconsistent error.

In whole-program type inference, constraint variables are created, which are placeholders for the type qualifiers. The same mandatory constraints are generated as in type check mode. The constraints for the whole program are collected and solved together by a solver. Multiple type-safe solutions are possible in inference mode and, as long as a solution exists, inference succeeds.

Annotation mode creates additional, breakable, preference constraints. It annotates the program with the most precise types possible by relying on the solver to produce the optimal solution guided by hard and soft constraints. The annotation mode inserts the inferred solutions back into the source code, giving the developer a chance to inspect the results. This is intended to help developers create well-specified applications and libraries. Like in inference mode, if any of the mandatory constraints cannot be satisfied, errors are raised for the code locations which generated the unsatisfiable constraints.

If inference succeeds, there exists a valid typing for the program. If a typing exists, inference will succeed. However, type checking with default qualifiers can fail for different reasons than type inference. Errors given in inference mode provide a whole-program view of the reasons why no type-safe solutions can be inferred, and indicate potential errors.

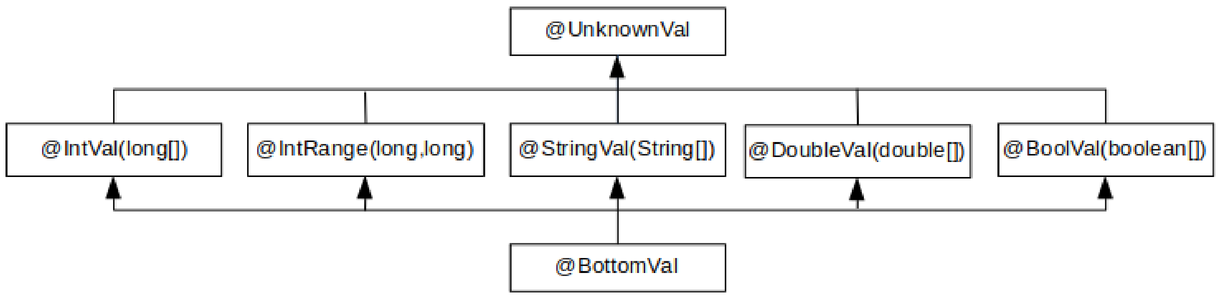
Prior type systems that build a type inference using the Checker Framework Inference includes the Ontology type system [14], Immutability type system [44], and the Units type system (Chap. 5). The Ontology type system is used to reason about a coarse abstraction for a given program based on ontic concepts. The immutability type system is used to control mutation and avoid unintended side-effects.

2.3 Constant Value Type System

The Constant Value type system is used to approximate the possible value of an expression at compiler time. Fig. 2.3 shows the type hierarchy of this type system containing the main type qualifiers and the subtyping relationships among them. The main type qualifiers for this system are `@BoolVal`, `@IntVal`, `@IntRange`, `@DoubleVal`, and `@StringVal`. `@IntVal` and `@IntRange` indicate the possible values for a `byte`, `short`, `char`, `int` or `long` type and their wrappers. `@DoubleVal` indicate the possible values for a `byte`, `short`, `char`, `int`, or `long` type and their wrappers. The `@IntVal` and `@DoubleVal` type annotations take a set of possible values as argument. This set is limited to 10 entries. At run time, the expression will evaluate to one of the values in the set. If an integral variable can be more than 10 values, then the `@IntVal` annotation changes to `@IntRange`. `@IntRange` takes a lower bound and an upper bound. The smallest value in the range is the minimum value of long, -9223372036854775808L, and the largest value in the range is the maximum value of long, 9223372036854775807L. At run time, the expression will evaluate to a value between the bounds inclusively. `@BoolVal` indicates the possible values for `boolean` and its wrappers, and `@StringVal` indicates the possible values for `String` Objects.

For two annotations of the same type, subtypes have a smaller set of possible values. At the very top of the hierarchy is `@UnknownVal` ($\top$), which means the variable could be any value. At the very bottom of the hierarchy is `@BottomVal` ($\perp$), which means the expression could only be evaluated to null. These qualifiers are internal and should never be written by a programmer.

Many related works statically predict the possible values of variables at run-time, such as including a value range analysis to compilers [26]. Frama-C is a modular static analysis framework for the C language and it has a value analysis plug-gin [10]. To reduce the number of false-positives, it discards the alarms that seem most likely to be false positives, allowing false-negatives. In the Constant Value type system, all errors are presented ensuring soundness. It has the same limitations as the Narrowing and Widening Checker described in Sec. 3.5.3, where loops and external functions call lack precision. It does not contain a whole-program inference. An entry point must be specified when analyzing a program with multiple functions. The global variable will differ based on the specified entry point. With whole-program inference, Value Inference for Integral Values sees all operations that affect the value of the global variable, and estimate the bounds of that value accordingly. It does not have type inference capabilities.



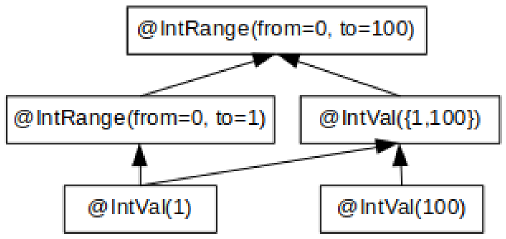


Figure 2.3: Main qualifier hierarchy of the Constant Value Checker. The type qualifiers are pointing to its supertype. For example, `@IntVal(1,100)` is an subtype of `@IntRange(0,100)`, and a supertype of `@IntVal(1)` and `@IntVal(100)`, but is neither a subtype nor a supertype of `@IntRange(0,1)`.

2.4 *PUnits* - The Units Type System

PUnits [46] allows developers to specify units of measurement systems by defining a set of base units, and to specify the units of their program elements via type qualifiers. Each type qualifier specifies a single unit. For example, `@m int` is a qualified type specifying an integer with the unit of meters. The qualifier `@Dimensionless` specifies the unit of truly dimensionless quantities, such as number literals, e, and π, and usually does not need to be written. The qualifiers are organized as a flat lattice over the set of scientific units, with top $\top$ and bottom $\perp$ type qualifiers. $\top$ expresses unit-wise agnostic types. To support object-oriented languages, $\perp$ is used for null types, and $\top$ and $\perp$ are respectively used as the upper and lower bounds of generic type parameters. We use the term *unit* to denote both the scientific unit of a program element as well as the type qualifier used to annotate the program element.

PUnits internally represents all units through a normalized representation to efficiently reason about program correctness. A *base unit* is the unit of measurement from which all other units from the same dimension can be derived. Each base unit is represented by a symbol. For SI, the base units are $\{\mathtt{m}, \mathtt{s}, \mathtt{g}^{2}, \mathtt{A}, \mathtt{cd}, \mathtt{K}, \mathtt{mol}\}$.

PUnits represents scientific units as a single prefix and a product of one or more base units, each raised to an integer power: $p\ \overline{u^z}$ (where $\overline{X}$ denotes a sequence of elements $X_1, \ldots, X_k$). Each base unit appears exactly once.

The representation can support any unit system. In the Java implementation, p is encoded as a base-10 prefix with an integer exponent to support SI-like units: $p = 10^z$. Thus, we need to declare additional base units for units that are not a base 10 multiple (e.g., inch vs. cm). Fig. 2.4 shows some examples of how units are represented in *PUnits*. The prefix p could be encoded as a floating-point value. However, the precision of analysis will be subjected to floating-point rounding errors, and safe floating-point comparisons must be used. The set of base units is customizable, allowing for base units such as currencies, abstract quantities, lengths, and pixels. There are special cases where the units alone cannot decide whether a computation is permitted. For example, a radian and a steradian, respectively defined as m/m and m^2/m^2, would both be normalized to dimensionless. In *PUnits*, radian and steradian can be declared as two different base units. Type systems that only allow SI units cannot support such derived units. Some derived units have several different representations, such as energy, which can be kinetic, thermal, or electrical, etc. These representations can be supported in *PUnits* by declaring

[2]Although `kg` is defined as the SI base unit for mass, *PUnits* uses `g` since the metric prefix is captured and encoded in the prefix component of the normalized representation.

Unit	$\texttt{10}^z$	$\texttt{g}^z$	$\texttt{m}^z$	$\texttt{s}^z$	...
$\texttt{m}^2$	0	0	2	0	0
`kg`	3	1	0	0	0
`N`	3	1	1	-2	0
`kN`	6	1	1	-2	0
`Dimensionless`	0	0	0	0	0

Figure 2.4: Example unit representations. Squared-meter is represented as $\texttt{m}^2$ and kilogram is represented as 10^3 `g` with `g` as the base unit. All SI prefixes of a unit are multiplied together into one prefix. Newton is represented by 10^3 $\texttt{g m s}^{-2}$ and a kilo-newton is normalized and represented by 10^6 $\texttt{g m s}^{-2}$. `@Dimensionless` is represented by 10^0.

distinct base units. The representation is compact and allows for efficient calculation of the resulting units of various arithmetic operations. Two units are equal if and only if every component in their normalized representations are pairwise equal.

PUnits can also be instantiated for dimensional analysis [9] by using a set of base dimensions (e.g. `@Length`) instead of base units (e.g. `@m`), together with the fixed prefix $p = 1$. Dimensional analysis can be useful if the developer does not care about the specific units, such as verifying if two units are convertible. Unit-wise analysis offers more precision than dimensional analysis and can detect more errors.

Chapter 3

Numeric Narrowing and Widening Conversion Checker

3.1 Introduction

Java allows narrowing and widening primitive conversions on seven of the eight primitive data types [23], except `boolean`. These data types are, from the smallest to the largest size, `byte`, `short`, `char`, `int`, `long`, `float` and `double`. The five integral types are `byte`, `short`, `char`, `int`, and `long`; `double` and `float` are floating-point types. This section focuses only on the narrowing and widening of integral primitives types and their wrappers. Table 3.1 shows the size of the five integral types. Widening is the process of converting a smaller size data type to a larger size type, and it is done implicitly. Narrowing is the process of converting a larger size data type to a smaller size type, and requires explicit casting. All integral data types in Java are signed except for `char`; `char` is a 16-bit Unicode character rather than signed integer, but can be represented as an integer.

A narrowing conversion from an integral type to another integral type T discards all but the n lowest order bits, where n is the number of bits used to represent type T. This may cause a loss of information about the overall magnitude of a numeric value. Fig. 3.1 is an example where the program has an incorrect behaviour due to loss of information during a narrowing conversion discussed in the paper on EOF Value Checker [13]. Reading from an `IntputSream` in Java returns an integer from 0 to 255 to represent an unsigned byte or -1 if the end-of-file (EOF) has been reached. The value returned by `read` should be compared to -1 before converting it to a byte. In this example, the conversion is performed before the comparison. The while loop could exit prematurely when `read` returns a 255.

```
InputStream in = new FileInputStream("file");
byte data;
while ((data = (byte) in.read()) != -1) {     // Error
    print(data);
}

Reader in = new FileReader("file");
char data;
while ((data = (char) in.read()) != -1) {     // Error
    print(data);
}
```

Figure 3.1: Two program failures caused by loss of information due to incorrect usage of narrowing conversion from `int` to `byte` and `char`. The first while loop could exit prematurely if `read` returns a 255. The second while loop is stuck in an infinite loop as `char` cannot be −1.

Similarly, reading form an `Reader` returns an integer from 0 to 65535 to represent a `char` or −1 if EOF has been reached. The conversion is performed before comparing it with −1 and the method is stuck in an infinite loop as `char` will never be −1.

A widening conversion from a signed integer to another integral type sign-extends the twos-complement representation of the integer value. Even though a widening conversion from an integral type to another integral type does not lose information, problems would occur when developers are trying to work with unsigned value. Unsigned bytes are usually desired for low-level programming. Since Java does not handle unsigned bytes, the most common way to represent an unsigned byte in Java is using a signed `int`. A bitwise-and (&) with 0xFF is required to correctly convert the signed byte to an `int` to represent the unsigned byte. Consider the program in Fig. 3.2, variable `byteValue` can be widened to a signed value or an unsigned value both in `int`. However, Java has no way of knowing which conversion is desired. Since variable `byteValue` is initialized as an unsigned byte 0xFF, after the widening conversion, the value should stay as 0xFF for consistency. Shorts have similar problems. Even though unsigned short can be represented using `char`, they are sometimes represented using signed int to differentiated the use of `char` for a character value.

Despite these problems, both of these conversions will never result in a run-time exception. Incorrect widening and narrowing behaviours may trigger other run-time exceptions

```
int value = 0xFF;
byte byteValue = (byte) value;          // value = 255
int signedByte = byteValue;             // value = -1
int unsignedByte = byteValue & 0xFF;    // value = 255
```

Figure 3.2: Two different widening of `byte` to `int` as int is sometimes used to represent unsigned bytes in Java. The second widening is the desired behaviour as it is consistent with the initialization of `byteValue`.

and produce incorrect programs, and the developer may have a difficult time pin-pointing the exact source of the problem. This chapter introduces the Narrowing and Widening Checker, a type system that prevents loss of information during narrowing conversion of primitive integral data types and automatically distinguishes the signedness of variables to eliminate the ambiguity of a widening conversion from type `byte` and `short` to type `int` in Java during program compilation. This checker is built on top of the Constant Value type system, and is inspired by generalizing the problem resolved by the EOF Value Checker [13], where it focuses on preventing unsafe EOF comparison such as the examples in Fig. 3.4.

The Narrowing and Widening Checker is evaluated on 23 of the Apache Commons project with 224k lines of code. Out of these projects, 18 projects failed with 28 widening warnings, 354 narrowing errors, 217 type incompatible errors, and 118 override errors. The output is manually examined and 5 of these projects are evaluated in detail. Manual annotations are added to those 5 projects to resolve the issued errors. In these 5 projects, the Narrowing and Widening Checker detects 69 real errors and 26 false-positives, with a false-positive rate of 37.7%.

3.2 Type Hierarchy and Rules

The Narrowing and Widening Checker is built on top of the Constant Value type system. The main type annotations for the Constant Value Checker are `@BoolVal`, `@IntVal`, `@IntRange`, `@DoubleVal`, and `@StringVal`. Out of these, the only ones that the Narrowing and Widening Checker uses are `@IntVal` and `@IntRange`, as they are used for integral primitives and their wrappers. The type introductory rules and method-local flow-sensitive type refinement minimize the need for annotations in method bodies and remove unnecessary clutter. Narrowing and Widening Checker follows all the type rules of the Constant

Data Type	Range	Default Annotation
`byte`	8-bit signed integer	`@IntRange(-128,127)`
`short`	16-bit signed integer	`@IntRange(-32768,32767)`
`char`	16-bit Unicode character	`@IntRange(0,65535)`
`int`	32-bit signed integer	`@IntRange(-2147483648,2147483647)`
`long`	64-bit signed integer	$\top$

Table 3.1: Default annotations for different data types on field instances, method parameters and return. Long is defaulted to $\top$ as it contains all possible ranges.

Value Checker with additional rules added.

Sec. 3.2.1 introduces the default qualifier for different data types in various locations. Sec. 3.2.2 presents the additional type rules defined by the type system. Sec. 3.2.3 presents the process of method-local flow-sensitive type refinement and the constraints generated to ensure soundness.

3.2.1 Type Introductory Rules

The Constant Value Checker has defined a default qualifier for every variable, bound, cast, and method. It uses the CLIMB-to-top rule, which states that the $\top$ qualifier in the hierarchy is used as the default qualifier. For the Narrowing and Widening Checker, the default qualifier for some instances, methods, and casts use a different qualifier than $\top$.

Table 3.1 defines the default annotations of each data type on field instances, method returns, and method parameters locations. Other than `char`, all integral data types in Java default to signed ranges, which follows Java's specification. If the programmer wants to declare the type qualifier of the field variable, the method parameter, or the method return to an unsigned type, then an explicit annotation is required. `@IntRange(0,255)` is used for unsigned byte, and `@IntRange(0,65535)` is used for unsigned short.

For primitive narrowing conversion type, the type qualifier for `byte` can be a subtype of `@IntRange(-128,127)` or `@IntRange(0,255)`, and for `short` can be a subtype of `@IntRange(-32768,32767)` or `@IntRange(0,65535)`, depending on the integral range of the expression to be narrowed. This is to allow the narrowing of a larger size type to both the signed range and the unsigned range of bytes and shorts. Unsigned `int` can be represented using `long`. It is not supported for this type system as it is not commonly used, but can be easily included. Unsigned `long` cannot be represented using a primitive data type in Java and thus is not supported. All other data types follows Table 3.1.

$$\frac{\Gamma \vdash e : Q\ c \qquad d <: c \qquad Q <: P}{\Gamma \vdash (P\ d)\ e : Q\ d}$$

Figure 3.3: Type rule for narrowing conversion. Γ is the environment. Q is the type qualifier of expression e. P is the type qualifier of the cast. Narrowing a variable from type c to d is only allowed if Q is a subtype of type P, meaning no information is lost.

To allow the data-flow-sensitive refinement to process for local variables, variables default to the minimum and maximum bounds. `@IntRange(-128,255)` is the default qualifier for `byte` local variables and `@IntRange(-32768,65535)` is the default qualifier for `short` local variables. The other default qualifier for local variables follows Table 3.1.

3.2.2 Type Rules

The Narrowing and Widening Checker includes all the types rules of the Constant Value Checker. Additional type rules are added to prevent the loss of information during narrowing and ensure consistency during the widening of `byte` and `short` types.

Narrowing Type Rule

This Checker restricts the standard type rules for narrowing conversions. Narrowing a value that has an annotation with a range that is larger or smaller than the annotation range to be converted to is forbidden. As shown in Fig. 3.3, only expressions with annotation that is a subtype of the narrowing type can be converted. An environment Γ maps variables to their types. Q is the type qualifier of expression e with data type d and P is the type qualifier of the primitive narrowing conversion with data type c. Narrowing a variable from type c to d is only allowed if Q is a subtype of type P. The integral type narrowing rules ensure that the annotation of the expression to be converted is compared against the narrowing type annotation before being converted.

This type narrowing rule effectively prevents loss of information by preventing narrowing conversions that may result in a loss of information. An unsafe cast error is issued for any possible narrowing that could result in lost information. Fig. 3.4 is an example of the narrowing error issued by the Narrowing and Widening Checker for the example shown in Fig. 3.1. The correct implementation is shown in Figure 3.7. Note that the cast to byte is allowed after the comparison against -1, which refined the annotation on of the `inbuff` variable from `@IntRange(-1, 255)` to `@IntRange(0, 255)`, and allowing the cast in the loop body. No explicit annotation is required in the source code.

```
error: [cast.unsafe] "@IntRange(from=-1, to=255) int" may not be
   casted to the type "@IntRange(from=0, to=255) byte"
        while ((data = (byte) in.read()) != -1) {
                       ^
error: [cast.unsafe] "@IntRange(from=-1, to=65535) int" may not be
    casted to the type "@IntRange(from=0, to=65535) char"
        while ((data = (char) in.read()) != -1) {
                       ^
```

Figure 3.4: Cast unsafe error issued by the checker for the program in Fig. 3.1.

$$\frac{\Gamma \vdash e : Q\ \mathit{byte} \qquad Q <: \mathit{IntRange}(0, 255)}{\Gamma \vdash \texttt{0xFF}\ \&\ e : Q\ \mathit{int}}$$

$$\frac{\Gamma \vdash e : Q\ \mathit{byte} \qquad Q <: \mathit{IntRange}(-128, 127)}{\Gamma \vdash\ \ e : Q\ \mathit{int}}$$

$$\frac{\Gamma \vdash e : Q\ \mathit{short} \qquad Q <: \mathit{IntRange}(0, 65535)}{\Gamma \vdash \texttt{0xFFFF}\ \&\ e : Q\ \mathit{int}}$$

$$\frac{\Gamma \vdash e : Q\ \mathit{short} \qquad Q <: \mathit{IntRange}(-32768, 32767)}{\Gamma \vdash\ \ e : Q\ \mathit{int}}$$

Figure 3.5: Type rules for implicit widening from a byte or a short to an int. Widening of expression e is dependent on qualifier Q.

Widening Type Rule

This checker restricts the standard type rule for widening from a byte or a short to an int. Assignment of an unsigned byte or short variable to an int variable is now forbidden. However, assignment of a signed byte or short to an int variable is still permitted. As shown in Fig. 3.5, the unsigned byte must be processed with a bitwise-and with 0xFF to allow the widening assignment from a `byte` to an `int`. The unsigned short must be processed with a bitwise-and with 0xFFFF to allow the widening assignment from a `short` to an `int`.

This type widening rule effectively prevents inaccurate widening from a `byte` or a `short` to an integer. An unsafe widening warning will be issued if any widening of an unsigned byte to int is performed without masking the bits. Fig. 3.6 is an example of the widening error issued by the Narrowing and Widening Checker for the example shown in Fig. 3.2 on line 2. The correct conversion is shown on line 3. Note that the type qualifier of the variable

```
error: [widening.unsafe] widening of the unsigned value @IntVal
   (255) byte is unsafe.
        int signedByte = byteValue;
                         ^
```

Figure 3.6: Widening unsafe error issued by the checker for the program in Fig. 3.2.

```
1 InputStream in = new FileInputStream("file");
2 @1 int inbuff;
3 @2 byte data;
4 inbuff@3 = in.read();         // @3 <: @1, @3 = @IntRange(-1,255)
5 if ((@4) inbuff@3 != 1) { // @4 <: @1, @4 = @IntRange(0,255)
6     data@6 = (@5 byte) inbuff@4; // @6 <: @2, @4 <: @5, @6 = @4
7 }
```

Figure 3.7: Method-local flow-sensitive type refinement process. Placeholders `@1` to `@7` are used to mark the locations for which types need to be determined based on the constraints generated at each line. `@4` is the refined value of `inbuff` after the comparison expression.

`byteValue` contains the unsigned value of `byteValue` instead of the signed value stored in memory. This is how the checker distinguishes an unsigned value from a signed value. If the type qualifier contains more values within the range of an unsigned range rather than a signed range, then Narrowing and Widening Checker determines that variable be unsigned. Otherwise, it determines the variable to be signed.

3.2.3 Method-local Flow-sensitive Refinement

This Checker uses Constant Value Checker's data-flow-sensitive type refinement to minimize the annotation effort on the local variables. An annotation on a local variable can be refined to its subtype if the annotation of the expression assigned to the variable is a subtype based on the type hierarchy in Figure 2.3.

The example in Fig. 3.7 highlights the types and generated constraints. @1 and @2 are the declared qualifiers for variable `inbuff` and `data`. During type checking, the types of local variables default to $\top$. On line 4, a refinement type @3 is introduced for the type of `inbuff` after the assignment. With this introduction, an equality constraint and subtype constraints are created and verified. The range of `inbuff` is refined from $\top$ to `@IntRange(-1, 255)`. On line 5, with the if statement, two more refinement types for `inbuff` are created, one for the `then` branch and one for the `else` branch. For clarity, the refinement in the `else` branch is not shown. @4 is refined to `@IntRange(0, 255)` from the boolean expression `inbuff == -1`. The refined type of `inbuff` is safely narrowed to `byte` on line 6 according to rule stated in Fig. 3.3.

Note how the type of `inbuff` and `data` is changed through re-assignments. Reusing of local variables to store values with various integral ranges is permitted. However, it forbids the incorrect use of local variables.

For each re-assignment of a local variable, a new refinement type is introduced. Each refinement type must be a subtype of the declared type of the variable and a corresponding subtype constraint is generated. Additionally, an equality constraint is generated between the expression type and the refinement type, to ensure the refinement type has the most specific type from the expression.

3.3 Type System Features

Similar to the Constant Value type system, the Narrowing and Widening Checker also supports polymorphism, used for method and constructor calls that take on various interval types. The Narrowing and Widening Checker also supports post-condition qualifiers, for evaluating the state of the input parameter after a boolean method call. These features increase the expressiveness of the type system.

3.3.1 Polymorphism

The Constant Value type system supports parametric polymorphism of values using the special `@PolyValue` type qualifier to parameterize methods and constructors. The Narrowing and Widening Checker also supports parametric polymorphism of values for methods via the special type qualifier `@PolyValue`.

At every method and constructor call site, the Narrowing and Widening Checker computes an interval range for the variable which is used to instantiate `@PolyValue` for the called method. The value is computed as the least-upper-bound of the method receiver and/or argument types. For method or constructors with only one polymorphic parameter, this is the same thing as equating the return.

Java's boxed primitives classes are treated the same as their primitive counterparts, as they ultimately hold and represent a numeric value. Polymorphism can be used in ensuring

```
@PolyValue Byte(@PolyValue byte val) { /*...*/ }
@PolyValue int byteValue(@PolyValue Byte this) { /*...*/ }
Byte good = new Byte(1);
byte good_val = good.byteValue();
@IntVal(2) byte bad_val = good.byteValue();
```

Figure 3.8: Example of a polymorphic class constructor and method. Both variables `good` and `good_val` are refined to be `@IntVal(1)`, which are consistent. An error is issued on line 5 as `bad_val` is expected to be `@IntVal(2)` but receives `@IntVal(1)` instead.

the consistency between the primitives and the wrappers. Fig. 3.8 shows an example of a wrapper class `Byte` having a polymorphic constructor and a polymorphic method. The example ensures that the instantiate of the wrapper and the primitive values return from the wrapper are consistent. After instantiating, variable `good` have type `@IntVal(1)` since the argument to the constructor is `@IntVal(1)`. Variable `good_val` is also `@IntVal(1)` because of the type of the receiver. The intervals of Java integral types are preserved during auto-boxing and auto-unboxing through a set of similarly annotated JDK methods that are provided to developers as library annotations.

3.3.2 Post-Condition Qualifiers

The Narrowing and Widening Checker contains two method post-condition qualifiers, `@EnsuresIntRangeIf` and `@EnsuresIntValIf`. A method post-condition is introduced on method annotations and ensures that a certain expression is a valid value or range after the method returns. Fig. 3.9 demonstrates how the post-condition qualifiers are used. These qualifiers ensure that if method `isEndOfFile` returns false, then the qualifier of expression v is `@IntRange(0,255)`. `@IntRange(-1,255)` is the pre-condition on the method parameter `v`. If the method returns true, then the qualifier of expression v is `@IntVal(-1)`. Without these qualifiers, the refinement would not correctly determine the range of c after the if statement.

The expression must be final, or is not modified within the method call; otherwise, soundness cannot be guaranteed. For example, assume that the method parameter `v` in Fig. 3.9 is modified by method `isEndOfFile`. Since Java always passes parameter variables by value, modification to variable `v` inside method `isEndOfFile` does not change the value of the passed-in argument variable `c`. Thus, claiming that the argument is `@IntVal(-1)` if

```
@EnsuresIntRangeIf(expr="#1", result=false, from=0, to=255)
@EnsuresIntValIf(expr="#1", result=true, value={-1})
boolean isEndOfFile(@IntRange(from=-1, to=255) int v) {
    return v == -1;
}

InputStream in = new FileInputStream("file");
int c = in.read();
if (!isEndOfFile(c)) {
    byte val = (byte) c; // OK
}
```

Figure 3.9: An example showing how `EnsuresIntRangeIf` and `EnsuresIntRangeIf` are used.

evaluated to true or `@IntRange(0,255)` if evaluated to false will not be sound. An error is issued by the Narrowing and Widening Checker if the parameter is modified.

The current post-condition qualifiers only support methods that return a boolean value. For future work, more expressive forms of post-condition qualifiers can be explored and implemented.

3.4 Implementation

The Narrowing and Widening Checker is implemented as a pluggable type system using the Checker Framework. Excluding empty lines and comments, 741 lines of Java code is used to implement this checker in total. It is an extension of the Constant Value type system and follows all the standard rules and data type refinement in the Constant Value type system with two new type rules in Sec. 3.2.2 implemented.

We add annotations to methods in libraries that handle stream inputs. These includes 26 `@IntRange(-1, 255)` annotations for the `read` methods in the `InputStream` classes and its subclasses; 9 `@IntRange(-1, 65535)` annotations for the `read` methods in the `Reader` classes and its subclasses; one `@IntVal(-1)` annotation for the `EOF` field in the `IOUtils` class in; 61 `@@IntRange(-1, 2147483647)` to other overloading `read` methods in both `InputStream` and `Reader` since the method returns the number of bytes or characters read or -1 indicating none.

We add annotations to methods in libraries that handle stream outputs. These includes 24 `@IntRange(-128, 255)` annotations for the parameters of the `write` methods in the `InputStream` classes and its subclasses. These annotations are provided as the value passed into this method is narrowed to an 8-bit value. 9 `@IntRange(0, 65535)` annotations for the parameters of the `write` methods in the `Reader` classes and its subclasses. These annotations are provided as the value passed into this method is narrowed to a 16-bit value.

76 methods in the primitive wrapper classes are polymorphic with `@PolyValue`. These include the constructors and the `getValue` methods. 8 methods in Java Math library are polymorphic with `@PolyValue`. These are the `min` and `max` methods calls.

It is easy to provide additional annotations for other APIs. More annotations are added to the library during the experiments.

3.5 Experiments

The Narrowing and Widening Checker is evaluated on 22 open source project. The Checker issued errors on 18 of these projects. Table 3.2 gives a summary of the size of the projects, the number of errors issued by the Checker, and the compilation time of running the Narrowing and Widening Checker vs. OpenJDK compilation. These errors are manually evaluated and are grouped into three categories: errors due to insufficient annotation, real errors, or false positives.

3.5.1 Errors due to Insufficient Annotations

Most of the errors issued are due to missing annotations in the program source code or insufficient library annotations. They can be resolved by adding explicit annotations to the source code and binary-code libraries. Type inference will remove the manual annotation effort on the source code for these types of errors. The details of how type inference is performed will be introduced in Chap. 4.

The defaults defined in Sec. 3.2.1 are an overestimate of all the possible values a variable will contain. Unless the developer explicitly specifies the type qualifier of a variable, defaults are applied. For example in Fig. 3.10, a `cast.unsafe` error will be issued without the explicit annotation on the parameter since `x` will be defaulted to `@IntRange(from=-2147483648, to=2147483647)` and the narrowing conversion on line 2 violates the type rule defined in Fig. 3.3.

			Errors Issued				Time	
Project	Files	SLOC	Narrow	Widen	Type	Override	OpenJDK	Checker
common-bcel	378	30475	30	0	13	2	8.418	70.34
commons-beanutils	111	11644	0	0	0	4	7.739	34.214
commons-bsf	59	6005	21	0	37	8	7.811	28.746
commons-cli	23	2798	0	0	0	0	5.328	15.868
commons-codec	63	7977	58	0	13	6	5.967	34.501
commons-compress	195	22543	65	1	13	21	7.385	56.128
commons-crypto	42	2954	8	0	2	2	5.385	16.118
common-csv	11	1605	10	0	3	2	4.799	13.696
commons-daemon	10	842	0	0	0	0	4.754	11.529
commons-dbcp	58	11596	0	0	0	0	6.135	28.004
commons-email	23	2815	1	0	0	0	4.927	14.996
commons-exec	32	1787	1	0	2	0	5.699	13.439
commons-fileupload	39	2364	3	0	4	4	5.925	15.015
commons-imaging	343	31368	65	13	31	4	7.932	83.123
commons-io	118	10017	29	14	28	41	6.263	27.233
commons-jxpath	171	18773	6	0	22	0	10.842	41.238
commons-logging	14	2613	0	0	0	11	5.798	15.007
common-net	211	20259	35	0	38	11	7.091	42.051
commons-ognl	155	13139	18	0	1	0	9.115	48.778
commons-scxml	100	9075	0	0	0	0	6.040	30.065
commons-text	64	5886	3	0	2	2	5.874	30.219
commons-validator	64	7460	1	0	8	0	6.064	25.133
Total	2285	223995	354	28	217	118	134.265	696.551

Table 3.2: The SLOC column counts the number of non-comment, non-blank lines of code. The `Narrow` column counts the total number of `cast.unsafe` errors issued. The `Widen` column counts the total number of `widening.unsafe` warnings issued. The `Type` column counts the number of `assignment.type.incompatible`, `argument.type.incompatible`, and `return.type.incompatible` issued. The `Override` column counts the number of `override.param.invalid` and `override.return.invalid` errors issued. All values under `Time` are in seconds. It compares the time taken with OpenJDK compilation vs. the type checking time with the checker for checking the entire project.

```
1 byte bound(@IntRange(0,255) int x) {
2     return (byte) x;
3 }
4 void call(int y){
5     y = Data.getUnsignedByte();
6     bound(y);
7 }
```

Figure 3.10: An example of errors issued due to insufficient annotations in source code. A `cast.unsafe` error will be issued on line 2 without the explicit annotation on parameter `x`. A `argument.type.incompatible` error will be issued on line 6 without the explicit annotation on for binary library method `getUnsignedByte()`.

The Narrowing and Widening Checker requires annotations for binary-only dependencies. Applications have large dependencies on binary-only libraries, like the JDK. In Fig. 3.10, an annotation is required for the return type of function `getUnsignedByte` for line 6 to be a valid operation as variable `y` passed into the function `bound` is required to be a subtype of the parameter declaration. An `argument.type.incompatible` will be issued on line 6 without the annotation. Manual annotation efforts for libraries can be re-used to type check in other projects utilizing the same APIs.

All the override errors issued by the Checker are due to missing annotations methods signatures in the source code. When an instance method m_1, declared in class c_1, overrides another instance method m_2, declared in a class c_2 where c_1 is a subclass of c_2, the Liskov Substitution Principle specifies that the method return of m_1 should be a covariant of m_2, and the method parameter of m_1 should be a contravariant of m_2. This means that the type qualifier of the method return in m_1 must be a subtype of m_2, and the method parameter in m_1 must be a supertype of m_2.

The Value inference that is introduced in Chap. 4 is run on the 18 projects that failed type check. Out of these 18 projects, 5 successfully inferred. This means 13 projects issued errors that may contain real errors. Out of these 13 projects, 5 are picked and manually annotated to clean up errors issued due to insufficient annotations. These 5 projects are picked because their annotation efforts is slightly lower compared to other projects and they show interesting results. Table 3.3 gives an overview of the numbers of manual annotations added and the numbers of errors issued.

In total, 153 annotations are manually added for 5 projects and 69 real errors were detected with 26 false positives. The high annotation effort on project `commons-io` was

	Manual Annotation			**Errors Issued**			
Project	`IntVal`	`IntRange`	`EnsureIf`	Narrow	Widen	Type	FP
commons-bcel	1	19	2	19	4	3	12
commons-crypto	0	11	0	5	0	0	3
commons-csv	14	8	4	2	0	2	1
commons-io	8	83	0	10	14	7	8
commons-text	0	3	0	3	0	0	2
Total	23	124	6	39	18	12	26

Table 3.3: The `EnsuresIf` column indicates the number of `@EnsuresIntValIf` and `@EnsuresIntRangeIf` manual annotations added to the project. `IntVal` indicates the number of `@IntVal` and `IntRange` indicates the number of `@IntRange`. The `narrow` column counts the total number of `cast.unsafe` error issued. The `widen` column counts the total number of `widening.unsafe` issued. The `type` column counts the number of `type.incompatible` issued. Column `FP` indicates the number of false positives within the issued errors.

due to the number of methods that overwrites the methods in the `java.io` library.

3.5.2 Real Errors

In this section, the real errors issued by the Narrowing and Widening Checker for each project are described in detail.

commons-bcel[1]: The Byte Code Engineering Library (BCEL) is intended to analyze, create, and manipulate Java binary files. Four narrowing errors are from converting the return value from `read` in the `InputStream` library to `char` before checking against the EOF -1. It is also suspicious that the value return is converted to `char` instead of `byte`. Maybe the desired library to use is `Reader`. There is a type error resulting from passing in an `int` value ranges from 0 to 65535 obtained from the `read` method in the `Reader` library to function `write` from the `OutputStream` library which is expecting an integer within the `byte` range. It probably should use the `write` method from the `Writer` library instead. There is one narrowing error where EOF from `read` in the `Reader` libary is not checked before converting it to a `char`.

Four of the widening warnings issued are due to the widening of an unsigned short to int. The two variables being converted are named `start_pc` (program counter) and

[1]https://commons.apache.org/proper/commons-bcel/

`line_number`, and are converted from an `int` that ranges from 0 to 65535 to `short`. The `toString` function prints these values without the bitwise-and operation with `0xffff`. These two values can be printed as negative values without the correct widening conversion.

Nine of the narrowing errors issued are from narrowing the result of arithmetic computations where the smallest resulting ranges estimated by the checker from the computations is larger than the target range. Seven of the narrowing and type errors are from over-approximated parameters and return values. From inspection, there is no clear indication of the bounds on these values, so they are labelled as real errors. One narrowing error is from converting a `double` to an `int`.

commons-crypto[2]: Commons Crypto is a cryptographic library. The checker picked up one ordering of operation error `(byte) counter & 0xff` where the `int counter` is narrowed to `byte` before the bitwise-and operation. There should be parenthesis around `counter & 0xff`. The other four narrowing errors are from arithmetic computations where the result range is not a subtype of the expected range.

commons-csv[3]: Commons CSV reads and writes files in variations of the Comma Separated Value (CSV) format. Function `getLastChar` in the project returns the last character that was read as an integer between 0 to 65535, or -1 if the last read is a EOF, or -2 if no characters were read. This method is annotated with `@IntRange(-2, 65535)`. There are two attempts made narrowing the value return by this function to `char` before checking it against -1 and -2, and one attempt of passing this value to a function expecting `@IntRange(-1, 65535)`, a subtype of `@IntRange(-2, 65535)`, as its parameter.

The other type error issued also results from passing into a function that is not its supertype. Function `isWhitespace` has its parameter annotated with `@IntRange(0, 65535)` since the parameter is narrowed to a `char` in the method. A value obtained from method `read` from the `InputStream` library which ranges from -1 to 65535 is passed into this method before checking it against -1.

commons-io[4]: Commons IO is a library of utilities to assist with developing IO functionality. The 14 widening warnings issued are from the widening of an unsigned byte to int. In this expression `write( (byte)( ( value >> 0 ) & 0xff ) )`, `value >> 0 ) & 0xff` returns a range from 0 to 255 after the bitwise-and operation. Method `write` from the `OutputStream` takes `int` as its parameter type. Since the expression is already within the byte range and function `write` ignores the 24 high-order bits of the parameter, narrowing the expression to `byte` is unnecessary.

[2]https://commons.apache.org/proper/commons-crypto/

[3]https://commons.apache.org/proper/commons-csv/

[4]https://commons.apache.org/proper/commons-io/

Two issued errors are related to method `read` in the `InputStream` library. There is an attempt of narrowing the return value from method `read` in the `InputStream` library before checking the return value of `read` to `byte` before checking for -1. This error is also detected by the EOF Value Checker. Function `readUnsignedByte` returns an int between 0 to 255. The return value of `read` is used as its return value before checking for -1.

Six of the narrowing and type errors are from arithmetic computations where the result range is not a subtype of the expected range. Eight of the narrowing and type errors are from over-approximated parameters. From inspection, there is no clear indication of the bounds of these parameters, so they are labelled as real errors.

commons-text[5]: Commons Text is a library focused on algorithms working on strings. Three of the narrowing errors issued are related to the process of converting an integer representation in string in data type `int`. The resulting `int` is then narrowed to `char`. From inspection, there are no clear indications where the integer representation is within the `char` rang, so they are labelled as real errors. Note that these operations could also produce false positives as the current type system does not handle string to int manipulation, and vice versa.

3.5.3 False Positives

In this section, the different types of false positives issued are described in detail. These show the limitations of the Checker Framework and the Narrowing and Widening Checker.

Loops: The Narrowing and Widening Checker does not have a way to precisely estimate the number of times a loop will be executed, such as using a loop invariant. How the Dataflow Framework handles loop evaluations are described in detail in Sec. 6. Lines 1 to 6 in Fig. 3.11 give an example of a `case.unsafe` false positive due to the lack of loop precision. Since this type system could not estimate the number of times the loop is executed, variable i is estimated to be between 0 and 2147483647 by the checker, and a `cast.unsafe` error is raised on line 5. 11 of the false positives in `commons-bcel` results from such limitations.

External Checks: Some checks are evaluated at an external method invocation rather than inside the local method. The `@EnsuresIf` post-condition qualifiers can only evaluate simple method calls where the input is `final` and the output is a `boolean`. The Narrowing and Widening Checker cannot correlate the return value of a more complicated external method invocation check with the value being checked. Lines 8 to 13 in Fig. 3.11 are an

[5]https://commons.apache.org/proper/commons-text/

example of a false positive due to such limitations. Variable n is estimated to be less than 0 even though it passed the check on line 8. The false positives from project `commons-csv`, `commons-text`, and one from `common-crypto` are all resulted from such limitations.

Method Return Precision: The `min` and `max` functions from Java's `Math` library are annotated with `@PolyValue`. Therefore, the return value of method `Math.min` and `Math.max` are the least-upper-bound of the two parameters rather than the min range and the max range of the two. Lines 15 to 20 issue a false positive due to imprecision of the value returned by method `Math.min`. Since `remaining > 0` and `SIZE = 2048`, the return value should be between 0 to 2048. Two of the false positives from project `commons-cryto`, and seven from `common-io` results from imprecision when evaluating the `Math.min` method. Similarly, when `Integer.parseInt` is called to convert a string representation of an integer to `int`, the full range of `int` is always returned. Lines 22 to 25 show an example where the checker fails in determining the range returned to be within `byte` since the length of the string is two. One of the false positives from project `commons-bcel`, and one from `common-io` results from imprecision when evaluating this method.

3.5.4 Performance Overhead

This case study is performed on a cloud instance with a four-core CPU and 16GB RAM, running 64-bit Ubuntu 20.04.

Table 3.2 reports the execution times of the Narrowing and Widening Checker modular type-checking for 22 projects. The projects take 134.265 seconds in total to compile using the OpenJDK 11 compiler, and 696.551 seconds in total using the checker to type check the projects. The overhead is approximately 5.188x and is consistent with other type systems developed using the Checker Framework[18, 13].

The performance of the checker in type check mode enables it to be used in edit-compile-unit-test development workflows. As type check mode is modular, faster performance is expected when checking one source file at a time. The Narrowing and Widening Checker performs adequately in type-check mode. Its performance is suitable for use in a real-world software development environment. Improvements to the Checker Framework will improve the performance of this type system and other type systems developed using the frameworks.

```java
// loop false positive
int MAX = 65535;
char[] char_map;
for (int i = 0; i < MAX; i++) {
    char_map[i] = (char) i;
}

// external check false positive
Long n;
check(n >= 0, "Negative thrown exception");
if (n <= Integer.MAX_VALUE) {
    int pos = (int) n;
}

// Math.min false positive
long remaining;
long SIZE = 2048;
if (remaining >= 0) {
    int size = (int) Math.min(SIZE, remaining);
}

// Integer.parseInt false positive
String url;
int octet = Integer.parseInt(url.substring(1, 3), 16);
byte b = (byte) octet;
```

Figure 3.11: Four false positive examples.

3.6 Related Work

The Narrowing and Widening type system is built on top of the Constant Value type system. It is inspired by the EOF Value Checker [13], and solves a more generalized problem. The EOF Value Checker focuses only on preventing unsafe EOF comparison. The Narrowing and Widening type system can detect the EOF error found in the EOF Value Checker as well as additional unsafe narrowing and widening conversions that were missed by the EOF Value Checker. EOF Value Checker is also missing type inference capabilities. Type inference for the Narrowing and Widening Checker is introduced in Chap. 4.

The Signedness Checker is another type system build using the Checker Framework. It supports two type qualifiers `@Signed` and `@Unsigned` to indicate the signedness of an integral variable. The Signedness Checker cannot automatically determine the signedness of a variable and does not know the actual range of the variable. It is used to prohibit meaningless operations with unsigned variables. The Narrowing and Widening Checker prohibits ambiguous widening conversions and can automatically signedness of variables. The two type systems can be combined to enforce stricter type rules for uses of unsigned integral types in Java.

Within our knowledge, the Narrowing and the Widening type system is the first type system that specifies additional rules for a narrowing and widening conversion for Java's integral data types.

Chapter 4

Whole-Program Type Inference for Integral Range Analysis

4.1 Introduction

In Sec. 3.5, 599 errors and warnings were issued on 18 unannotated projects. Some of the errors are due to missing annotations in the program source code and insufficient library annotations. Fig. 4.1 shows a similar example to Fig. 3.10 and shows the constraints generated at each line. Soundness is enforced by generating and solving typing constraints through syntax-guided constraint generation rules. With the code not annotated, the Narrowing and Widening Checker will issue a `cast.unsafe` error on line 2 due to the defaulting rules on `int`. To resolve the error on line 2, we can annotate the parameter `x` to either a value within `@IntRange(0, 255)` or `@IntRange(-128, 127)`. Then, we type check the code and the Narrowing and Widening Checker produces an `argument.type.incompatible` error on line 6. We can annotate parameter `y` to the same as parameter `x` to resolve this error. Then line 5 will fail with an `assignment.type.incompatible` error, and requires an annotation on the return type of function `getBytes`. This process is a burden and for larger projects with many interchanging calls between method, the annotation effort could be quite high. Whole-program inference infers type qualifiers on program variables and expressions, greatly reducing manual annotation burden in source code for developers.

In whole-program type inference, constraint variables are created, which are placeholders for concrete types. Whole-program type inference mode ensures a valid typing exists for a program by solving a set of generated constraints. The constraints generated are the same as the ones generated during the type checking process. Multiple type-safe solutions are possible in inference and, as long as a solution exists, inference succeeds. If inference fails, it pinpoints the set of code locations across the program that together could cause a type-inconsistent error.

This chapter introduces the Value Inference for Integral Values, a whole-program inference for integral analysis. It supports the main type qualifiers used by the Narrowing and Widening Checker, and is used to reduce the annotation efforts needed for the Narrowing and Widening Checker. The type system hierarchy of the Value Inference is similar to the Constant Value type system, but focuses only on inferring integral types for integral variables. The constraints generated the MaxSMT encoding can be easily adopted by the Constant Value Type System, which is lacking type inference capability.

The constraints generated for the whole program are collected and solved together by a MaxSMT solver in Value Inference. Value Inference offers two modes to be used by the developer, an inference mode, and an annotation mode. The inference mode is intended to give the program a quick check for the soundness of the whole program without any additional annotation effort. Annotation mode is whole-program type inference with additional, breakable, preference constraints. It provides the program with more precise and relevant annotations. We rely on the MaxSMT solver to produce the optimal solution guided by hard and soft constraints provided to the solver. The annotation mode inserts the inferred solutions back into the source code, giving the developer a chance to inspect the results. This is intended to help developers create well-specified applications and libraries.

Going back to the example in Fig. 4.1, the constraints written out in the comments are collected and solved together to determined its satisfiability. The program is, therefore, type safe. One possible solution in inference mode assigns `@IntRange(0,255)` as the value for all constraint variables. There are multiple solutions, as long as the constraints are satisfied, even `@IntRange(0,0)` for all constraint variables can be a possible solution, and `@3` can be anything within the range of type `int`. With annotation mode, additional constraints are added to prefer type `byte` to resolve to `@IntRange(-128,127)` and `int` to `@IntRange(-2147483648, 2147483647)` rather than some arbitrary value, which is more correct.

As a simple unsatisfiable example, we can annotate the return value of `getByte` on line 7 with `@IntRange(-1, 255)`. This propagates to the assignment on line 5, and then to the argument on line 6. On line 2, the constraints require `@2` to be a subtype of `@IntRange(0,255)`, and `@1` to be a subtype of `@2`. Since `@4` need to be a subtype of `@1` but must be a supertype of `@IntRange(0,255)`, `@1` cannot be a subtype of `@IntRange(0,255)`. The constraints introduced in lines 2, 5, and 6 are together unsatisfiable.

If inference succeeds, there exists a valid typing for the program. If a typing exists,

```
@3 byte bound(@1 int x) {
    return (@2 byte) x;          // @2 <: @3, @1 <: @2
}
void call(@3 int y){
    y@4 = getByte();             // @4 <: @3, @4 = @5
    bound(y@4);                  // @4 <: @1
}
@5 int getByte() { /* ... */ }
```

Figure 4.1: Example with constraint generated at each line. `@1` to `@5` are constraint variable slots. One valid solution is to have all slots

inference will succeed. However, type checking with default qualifiers can fail for different reasons than type inference. Errors given in inference mode provide a whole-program view of the reasons why no type-safe solutions can be inferred, and indicate potential errors.

The Value Inference for Integral Values is evaluated on the 18 Apache Commons projects in Table 3.2 where an error is issued by the Narrowing and Widening Checker. Out of these projects, 5 projects successfully evaluated to SAT and the Value Inference inferred 10639 annotations. The projects evaluated to UNSAT are manually examined and all of them contain a real narrowing error.

4.2 Value Inference Type System

The Value Inference is inspired by the Constant Value type system. The Constant Value type system does not support whole-program inference. It only infers method local variables using flow-refinement process described in Sec. 3.2.3. Value Inference for Integral Values provides a whole-program type inference for integral types qualifiers in Constant Value Type System.

Fig. 4.2 shows the type hierarchy of the Value Inference type system for whole-program type inference. The Value Inference Type System can be viewed as a simplified version of the Constant Value Type System where only the values of numeric integral data types are evaluated during compilation time. It supports all the type qualifiers needed for the Narrowing and Widening Checker, except for the post-condition qualifier. The main type annotations for the Value Inference are `@IntRange` and `@IntVal` indicating the possible

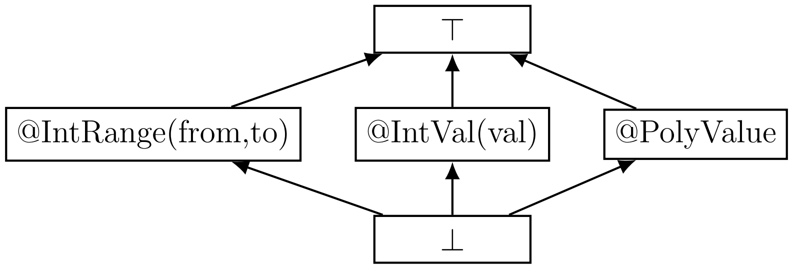


Figure 4.2: Type hierarchy of the Value Inference Type System. `@IntVal` and `@IntRange` are applicable to the five integral primitive data types and their wrappers `@PolyValue` is used for polymorphism.

values for all numeric integral data types. `@IntRange` takes a lower-bound and an upper-bound. The smallest value in the range is -2^{64} and the largest value in the range is $2^{64}-1$. At run time, the expression will evaluate to a value between the bounds inclusively. Unlike the type qualifier in the Constant Value type system, the `@IntVal` qualifier only takes a single possible value as argument for simplicity. Expanding the `@IntVal` type qualifier to support more than one value complicates the MaxSMT encoding.

Value Inference for Integral Values utilizes polymorphic method signatures in inference and annotate modes, and generates the corresponding constraints at each call site. Fresh type variables are generated at each call site and constrained to be the least-upper-bound of the arguments. Within a method declaration, the parameters of polymorphic methods are checked by substituting $\top$ for `@PolyValue`, and the return is checked by substituting $\perp$ for `@PolyValue`, since it can be instantiated with any integral range.

Value Inference for Integral Values does not infer additional `@PolyValue` annotations during whole program inference, but uses existing polymorphism correctly. It is not a common case to require a method to be annotated with `@PolyValue` as an inference result to satisfy constraints. We, therefore, did not need to design a strategy to infer `@PolyValue` method signatures and kept the inference complexity small.

4.3 Constraints

A constraint variable α is a placeholder for a concrete value. A fresh constraint variable is introduced for each location that is missing a type qualifier and for the resulting qualifier of each arithmetic operation and comparison operation. Inference assigns one concrete value to each constraint variable.

The Value Inference for Integral Values generates five kinds of constraints σ over types:

- Well-formedness constraint $wf(\alpha)$: enforces that any satisfying solution for constraint variable α is uniquely interpretable as a value.
- Subtype constraint $T_a <: T_b$: enforces T_a to be the same type or a subtype of T_b. e.g. converting between primitives using cast is a subtype constraint.
- Equality constraint $T_a = T_b$: ensures T_a and T_b are the same.
- Arithmetic constraint $\alpha = T_a\ op\ T_b$: enforces that α will be equal to the result of the arithmetic computation of $T_a\ op\ T_b$, for some defined arithmetic operation op. Value Inference for Integral Values abstracts over a concrete set of arithmetic operations. op constraints build upon the idea of Viewpoint Adaptation [19, 20] and can be thought of as computing the result type of the arithmetic operator op.
- Comparison constraint $\alpha \lhd T_a\ comp\ T_b$: enforces α as a refined type of T_a for which the comparison $\alpha\ comp\ T_b$ can be evaluated to true. The six comparison operators are equal to (==), not equal to ($\neq$), greater than ($>$), less than ($<$), greater or equal to ($\geq$), and less or equal to ($\leq$). *comp* constraints are also built upon the idea of Viewpoint Adaptation and can be thought of as computing all possible values of T_a where the expression can be evaluated to true.

The Value Inference for Integral Values generates six kinds of constraint variables:

- Constant α_{cons}: A constant constraint variable contains valid type qualifiers and does not need to be inferred, but helps with inference. A constant constraint variable is generated for literals, explicit annotations in the source code, and type qualifiers from binary libraries.
- Variable α_{var}: Variable constraint variables are created for field and local variable declarations, type casts, method parameter and return, type parameters, and class declaration and constructors. A variable constraint variable also stores the underlying type (e.g. `byte` or `Byte`) of the declared variable.
- Refinement α_{refin}: Refinement constraint variables are created during the variable assignment. Consider an expression `x = expr`, the refinement constraint variable is the refined value of variable `x`. For every refinement variable generated, a subtype constraint $\alpha_{refin} <: T_x$ and an equality constraint $\alpha_{refin} = T_{expr}$ are generated with it.

- Least-upper-bound α_{lub}: Least-upper-bound constraint variables are created when we want to find the least-upper-bound of two values. They are created when two branch merges or when there are multiple returns in a method. Consider finding the least-upper-bound of two types T_a and T_b, the least-upper-bound constraint variable contains the least-upper-bound of these two types. For every least-upper-bound variable generated, two subtype constraint $T_a <: \alpha_{lub}$ and $T_b <: \alpha_{lub}$ are generated with it.

- Arithmetic Variable α_{op}: This constraint variable is created when there is a binary arithmetic expression. Consider an arithmetic expression `x + y`, the arithmetic constraint variable contains the results of the operation. For every arithmetic variable generated, a arithmetic constraint $\alpha_{op} = T_x + T_y$ is generated with it.

- Comparison Variable α_{comp}: This constraint variable is generated for every variable in a comparison expression. They are used to refine the variables used after a comparison expression. Consider a comparison expression `x < y`, the comparison constraint variable contains the refined result of variable `x`. For every comparison variable generated, a comparison constraint $\alpha \lhd T_x < T_y$ is generated with it.

All the constraint variables must satisfy the well-formedness constraint $wf(\alpha)$.

4.3.1 Comparison Expression Refinement

This section explains the constraint variables and constraints generated by the Value Inference when encountering a comparison expression during inference. The method-local flow-sensitive type refinement type checking process is explained in Sec. 3.2.3. The support for comparison constraint and comparison constraint variables is added to the Checker Framework Inference for Value Inference. Unlike numeric value types, condition expressions usually would not change the type of the variable after evaluation for most type systems, so a simple comparable constraint (Sec. 2.2) is usually sufficient for ensuring program soundness. Value Inference is the first type inference build using the Checker Framework Inference with comparison constraints and comparison constraint variables.

When encountering a comparison expression, two refinement types will be introduced for each of the variables in the comparison expression, one for the `then` branch, and one for the `else` branch. During inference, the refinement types are represented with comparison constraint variables. These constraint variables will propagate to their respective branches. Each comparison constraint variables generates one comparison constraint. For a comparison expression where both operands are variables, four comparison constraint variables and comparison constraints are generated.

For expression $T_a == T_b$, constraints $\alpha_{a\_th} \lhd T_a == T_b$ and $\alpha_{b\_th} \lhd T_b == T_a$ propagates to the `then` branch, and constraints $\alpha_{a\_el} \lhd T_a \neq T_b$ and $\alpha_{b\_el} \lhd T_b \neq T_a$ propagates to the `else` branch. For expression $T_a \neq T_b$, the constraints are the same with `then` and `else` reversed.

For expression $T_a < T_b$, constraint $\alpha_{a\_th} \lhd T_a < T_b$ and $\alpha_{b\_th} \lhd T_b > T_a$ propagates to the `then` branch, and constraints $\alpha_{a\_el} \lhd T_a >= T_b$ and $\alpha_{b\_el} \lhd T_b \leq T_a$ propagates to the `else` branch. The constraints are the same for expression $T_a \geq T_b$ but with `then` and `else` reversed.

For expression $T_a \leq T_b$, constraint $\alpha_{a\_th} \lhd T_a \leq T_b$ and $\alpha_{b\_th} \lhd T_b \geq T_a$ propagates to the `then` branch, and constraints $\alpha_{a\_el} \lhd T_a > T_b$ and $\alpha_{b\_el} \lhd T_b < T_a$ propagates to the `else` branch. The constraints are the same for expression $T_a > T_b$ but with `then` and `else` reversed.

The example in Fig. 4.3 highlights the types and constraints generated when encountering a conditional expression. `@1` and `@2` are the declared qualifiers for variables `x` and `y`. At the comparison expression `x < y`, two refinement types `@3` and `@4` are introduced for the type of `x` and two refinement types `@5` and `@6` are introduced for the type of `y`. With this introduction, four subtype constraint `@3` <: `@1`, `@4` <: `@1`, `@5` <: `@2`, and `@6` <: `@2` and four comparison constraints `@3` $\lhd$ `@1` $<$ `@2`, `@4` $\lhd$ `@1` $\geq$ `@2`, `@5` $\lhd$ `@2` $>$ `@1`, and `@6` $\lhd$ `@2` $\leq$ `@1` are generated. The value of `@3` and `@5` will propagate to the `then` branch while the value of `@4` and `@6` will propagate to the `else` branch. The return type `@7` in this example is bounded by refined type of `x` and `y` are `@3` and `@6` respectively instead of the declared type `@1` and `@2`.

4.4 Encoding of Constraints for Solvers

Value Inference for Integral Values abstracts away the low-level constraint encoding from the high-level constraints through a solver interface. The solver abstraction allows inference to experiment with different solving techniques and solver systems.

Value Inference for Integral Values encodes constraint variables and constraints as a Maximum Satisfiability Modulo Theory (MaxSMT) problem using the linear integer arithmetic and boolean theories. A MaxSMT problem is similar to a Maximum Boolean Satisfiability (MaxSAT) problem, but incorporates additional theories. A MaxSMT problem

```
@7 int condition(@1 int x , @2 int y) {
    if ((@3, @4) x < (@5, @6) y) {
        // @3 <: @1, @5 <: @2, @3 ◁ @1 < @2, @5 ◁ @2 > @1
        return x;    // @3 <: @7
    } else {
        // @4 <: @1, @6 <: @2, @4 ◁ @1 ≥ @2, @6 ◁ @2 ≤ @1
        return y;    // @6 <: @7
    }
}
```

Figure 4.3: Method-local flow-sensitive type refinement process with comparison expressions. Placeholders `@1` to `@7` are used to mark the locations for which types need to be determined based on the constraints generated at each line.

consists of hard constraints and soft or breakable constraints with weights. A MaxSMT solver will generate a solution that satisfies all the hard constraints, while maximizing the weight of satisfying soft constraints. Encoding the constraints Σ as a MaxSMT problem allows Value Inference for Integral Values to take advantage of the optimizations that go into existing SMT solvers. Value Inference for Integral Values uses Z3 as its SMT solver [17]. The complexity of MaxSAT solvers is appropriate for the inference of expressive type systems [28]. Our use of a MaxSMT solver is appropriate because the Value Inference for Integral Values constraints additionally require integer theories for inferring interval ranges.

Each constraint variable is encoded as three boolean variables β^{top}, β^{bot}, β^{int} which respectively represents $\top$, $\bot$, and `IntRange`; two integer variables ω^{fr} and ω^{to} which respectively represents the from and to of the `IntRange` qualifier. Each constraint presented in Sec. 4.3 is encoded as a predicate expressed over the boolean and integer variables.

The encoding of the hard constraints along with the detailed encoding for Java's comparison operators are:

- *Well-formed Constraint* ($wf(\alpha)$): ensures each constraint variable represents a unique type. The encoding is dependent on the data type. At $\top$, the upper-bound and lower-bound value need to be set to as the maximum ($long_{max} = 9223372036854775807$) and the minimum ($long_{min} = -9223372036854775808$) value as it is the supertype.

$$\begin{aligned} wf(\alpha_t) \quad := \;& (\neg\beta^{top} \wedge \beta^{bot} \wedge \neg\beta^{int}) \vee (\neg\beta^{top} \wedge \neg\beta^{bot} \wedge \beta^{int} \wedge \omega^{fr} \leq \omega^{to} \wedge\ p(t)) \\ & \vee (\beta^{top} \wedge \omega^{fr} = long_{min} \wedge \omega^{to} = long_{max} \wedge \neg\beta^{bot} \wedge \neg\beta^{int}) \end{aligned}$$

Predicate $p(t)$ ensures that the constraint variable α_t stays within the allowable range of data type t where t is an integral primitive type or their boxed object. Unsigned int is not encoded since they are not commonly used, but can be easily included.

$$\begin{aligned}
p(byte) &:= (\omega^{fr} \geq -128 \land \omega^{to} \leq 127) \lor (\omega^{fr} \geq 0 \land \omega^{to} \leq 255)\\
p(short) &:= (\omega^{fr} \geq -32768 \land \omega^{to} \leq 32767) \lor (\omega^{fr} \geq 0 \land \omega^{to} \leq 65535)\\
p(char) &:= \omega^{fr} \geq 0 \land \omega^{to} \leq 65535\\
p(int) &:= \omega^{fr} \geq -2147483648 \land \omega^{to} \leq 2147483647\\
p(long) &:= true
\end{aligned}$$

- *Subtype Constraint* ($T_a <: T_b$): given the type lattice, it is enough to check whether the subtype is the bottom type, the supertype is the top type, or if a is within the range of b.

  $$T_a <: T_b \ := \ \beta_b^{top} \lor \beta_a^{bot} \lor (\beta_a^{int} \land \beta_b^{int} \land \omega_a^{fr} \geq \omega_b^{fr} \land \omega_a^{to} \leq \omega_b^{to})$$

- *Equality Constraint* ($T_a = T_b$): two types are equal if their encoding are equal.

  $$T_a = T_b \ := \ \beta_a^{top} = \beta_b^{top} \land \beta_a^{bot} = \beta_b^{bot} \land \beta_a^{int} = \beta_b^{int} \land \omega_a^{fr} = \omega_b^{fr} \land \omega_a^{to} = \omega_b^{to}$$

- *Arithmetic Constraint* ($\alpha_c = T_a \ op \ T_b$): if either argument is $\top$, the result has to be $\top$. If one argument is $\bot$, while the other argument is not $\top$, the result is $\bot$. If both of the arguments are `IntRange`, the resulting ranges are calculated with op.

  $$\begin{aligned}
  \alpha_c = T_a \ op \ T_b \ := \ & ((\beta_a^{top} \lor \beta_b^{top}) \land \beta_c^{top}) \lor (\beta_a^{bot} \land \neg\beta_b^{top} \land \beta_c^{bot}) \lor (\beta_b^{bot} \land \neg\beta_a^{top} \land \beta_c^{bot})\\
  & \lor (\beta_a^{int} \land \beta_b^{int} \land \beta_c^{int} \land p(c) \land \alpha_c^{\omega} = T_a^{\omega} \ op \ T_b^{\omega}))\\
  & \lor (\beta_a^{int} \land \beta_b^{int} \land \beta_c^{int} \land \neg p(c) \land bound(c)
  \end{aligned}$$

  Predicate $bound(c)$ sets the min and max range of α_c based on the data type of c. In Java, all operands in an arithmetic operation implicitly widen to type `int`, or `long` if one of the operands is a `long`. Therefore, α_c can only be a type `int` or type `long`. If the resulting range is larger than the allowable range, then it is set to the lower and upper bound of an `int` or `long`.

  $$\begin{aligned}
  bound(int) &:= \omega^{fr} = int_{min} \land \omega^{to} = int_{max}\\
  bound(long) &:= \omega^{fr} = long_{min} \land \omega^{to} = long_{max}
  \end{aligned}$$

  - *Addition* (+): the resulting variable is the smallest possible range that includes all values resulting from adding the two bounds together.

    $$\alpha_c^{\omega} = T_a^{\omega} + T_b^{\omega} \ := \ \omega_c^{fr} = \omega_a^{fr} + \omega_b^{fr} \land \omega_c^{to} = \omega_a^{to} + \omega_b^{to}$$

– *Subtraction* ($-$): subtracting the upper-bound of T_b^ω from the lower-bound of T_a^ω gives the minimum value, and subtracting the lower-bound of T_b^ω from the upper-bound of T_a^ω gives the maximum value.
  $\alpha_c^\omega = T_a^\omega - T_b^\omega \ := \ \omega_c^{fr} = \omega_a^{fr} - \omega_b^{to} \land \omega_c^{to} = \omega_a^{to} - \omega_b^{fr}$
– *Multiplication* ($\times$): the upper-bound of the resulting variable is the largest value out of the four possible combinations ($\omega_a^{fr} \times \omega_b^{fr}$, $\omega_a^{fr} \times \omega_b^{to}$, $\omega_a^{to} \times \omega_b^{fr}$, $\omega_a^{to} \times \omega_b^{to}$) when the bounds are multiplied to each other. The lower-bound of the resulting variable is the smallest value out of the four combinations.
  $$\begin{aligned}\alpha_c^\omega = T_a^\omega \times T_b^\omega \ :=& \\ &(\omega_a^{fr} \times \omega_b^{fr} \leq (\omega_a^{fr} \times \omega_b^{to} \land \omega_a^{to} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{fr} = \omega_a^{fr} \times \omega_b^{fr}) \\ \land\ &(\omega_a^{fr} \times \omega_b^{to} \leq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{fr} = \omega_a^{fr} \times \omega_b^{to}) \\ \land\ &(\omega_a^{fr} \times \omega_b^{to} \leq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{fr} = \omega_a^{fr} \times \omega_b^{to}) \\ \land\ &(\omega_a^{to} \times \omega_b^{fr} \leq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{fr} \times \omega_b^{to} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{fr} = \omega_a^{to} \times \omega_b^{fr}) \\ \land\ &(\omega_a^{fr} \times \omega_b^{fr} \geq (\omega_a^{fr} \times \omega_b^{to} \land \omega_a^{to} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{to} = \omega_a^{fr} \times \omega_b^{fr}) \\ \land\ &(\omega_a^{fr} \times \omega_b^{to} \geq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{fr} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{to} = \omega_a^{fr} \times \omega_b^{to}) \\ \land\ &(\omega_a^{to} \times \omega_b^{fr} \geq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{fr} \times \omega_b^{to} \land \omega_a^{to} \times \omega_b^{to}) \implies \omega_c^{to} = \omega_a^{to} \times \omega_b^{fr}) \\ \land\ &(\omega_a^{to} \times \omega_b^{to} \geq (\omega_a^{fr} \times \omega_b^{fr} \land \omega_a^{fr} \times \omega_b^{to} \land \omega_a^{to} \times \omega_b^{fr}) \implies \omega_c^{to} = \omega_a^{to} \times \omega_b^{to})\end{aligned}$$

– *Division* ($\div$): the problem of finding the smallest upper-bound and the largest lower-bound for divisions can be solved by separating it into 9 different cases: (1) both variable a and b contains only positive values, (2) only negative values, (3) left contains only positives and right operand contains only negatives, (4) left operand contains only negatives and right operand contains only positives, (5) left contains both and right contains only positives, (6) left contains both and right contains only negatives, (7) left contains only positives and right contains both, (8) left contains only negatives and right contains both, (9) both operands contains both signedness.
  $$\begin{aligned}\alpha_c^\omega = T_a^\omega \div T_b^\omega \ :=\ &(\omega_b^{fr} = 0 \implies \omega_c^{fr} = \omega_a^{fr} \land \omega_c^{to} = \omega_a^{to}) \\ \land\ &(\omega_b^{to} = 0 \implies \omega_c^{fr} = -\omega_a^{to} \land \omega_c^{to} = -\omega_a^{fr}) \\ \land\ &(\omega_a^{fr} > 0 \land \omega_b^{fr} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \div \omega_b^{to} \land \omega_c^{to} = \omega_a^{to} \div \omega_b^{fr}) \\ \land\ &(\omega_a^{fr} > 0 \land \omega_b^{to} < 0 \implies \omega_c^{fr} = \omega_a^{to} \div \omega_b^{to} \land \omega_c^{to} = \omega_a^{fr} \div \omega_b^{fr}) \\ \land\ &(\omega_a^{fr} > 0 \land \omega_b^{fr} < 0 \land \omega_b^{to} > 0 \implies \omega_c^{fr} = -\omega_a^{to} \land \omega_c^{to} = \omega_a^{to}) \\ \land\ &(\omega_a^{to} < 0 \land \omega_b^{fr} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \div \omega_b^{fr} \land \omega_c^{to} = \omega_a^{to} \div \omega_b^{to}) \\ \land\ &(\omega_a^{to} < 0 \land \omega_b^{to} < 0 \implies \omega_c^{fr} = \omega_a^{to} \div \omega_b^{fr} \land \omega_c^{to} = \omega_a^{fr} \div \omega_b^{to}) \\ \land\ &(\omega_a^{to} < 0 \land \omega_b^{fr} < 0 \land \omega_b^{to} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \land \omega_c^{to} = -\omega_a^{fr}) \\ \land\ &(\omega_a^{fr} \leq 0 \land \omega_a^{to} \geq 0 \land \omega_b^{fr} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \div \omega_b^{fr} \land \omega_c^{to} = \omega_a^{to} \div \omega_b^{fr})\end{aligned}$$

$$\begin{aligned}
&\wedge\ (\omega_a^{fr} \leq 0 \wedge \omega_a^{to} \geq 0 \wedge \omega_b^{to} < 0 \implies \omega_c^{fr} = \omega_a^{to} \div \omega_b^{to} \wedge \omega_c^{to} = \omega_a^{fr} \div \omega_b^{to})\\
&\wedge\ (\omega_a^{fr} \leq 0 \wedge \omega_a^{to} \geq 0 \wedge \omega_b^{fr} < 0 \wedge \omega_b^{to} > 0 \implies\\
&\qquad\qquad (\omega_a^{fr} \leq -\omega_a^{to} \implies \omega_c^{fr} = \omega_a^{fr})\\
&\qquad\quad \wedge\ (\omega_a^{fr} \geq -\omega_a^{to} \implies \omega_c^{fr} = -\omega_a^{to})\\
&\qquad\quad \wedge\ (-\omega_a^{fr} \geq \omega_a^{to} \implies \omega_c^{to} = -\omega_a^{fr}\\
&\qquad\quad \wedge\ (\omega_a^{fr} \leq -\omega_a^{to} \implies \omega_c^{to} = -\omega_a^{to}))
\end{aligned}$$

– *Modulo* (%): the resulting range is smaller than the upper bound of the divisor if the upper bound is larger than zero, and larger than the lower bound of the divisor if the lower bound is smaller than zero.

$$\begin{aligned}
&\alpha_c^\omega = T_a^\omega \ \%\ T_b^\omega \ := \ (\omega_b^{fr} \geq 0 \implies \omega_c^{fr} = 0 \wedge \omega_c^{to} = \omega_b^{to} - 1)\\
&\quad \wedge\ (\omega_b^{to} \leq 0 \implies \omega_c^{to} = 0 \wedge \omega_c^{fr} = \omega_b^{fr} + 1)\\
&\quad \wedge\ (\omega_b^{fr} < 0 \wedge \omega_b^{to} > 0 \implies \omega_c^{fr} = \omega_b^{fr} + 1 \wedge \omega_c^{to} = \omega_b^{to} - 1)
\end{aligned}$$

– *Left-Shift* ($\ll$): shifting operations in Java depends on the type of the left-hand operand. If the left-hand operand is an `int` type, only the 5 lowest-order bits of the right-hand operand are used. If the left-hand is a `long` type, then only the 6 lowest-order bits of the right-hand operand are used. For this reason, if the right-hand operand is not within 0 to 31, then we give up on trying to evaluate the precise range and return the full range.

$$\begin{aligned}
&\alpha_c^\omega = T_a^\omega \ll T_b^\omega \ := \ (\omega_b^{fr} < 0 \vee \omega_b^{fr} > 31 \implies \mathit{bound}(c))\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{to} < 0 \implies \omega_c^{fr} = \omega_a^{fr} \ll \omega_b^{to} \wedge \omega_c^{to} = \omega_a^{to} \ll \omega_b^{fr})\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{fr} < 0 \wedge \omega_a^{to} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \ll \omega_b^{to} \wedge \omega_c^{to} = \\
&\omega_a^{to} \ll \omega_b^{to})\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{fr} \geq 0 \implies \omega_c^{fr} = \omega_a^{fr} \ll \omega_b^{fr} \wedge \omega_c^{to} = \omega_a^{to} \ll \omega_b^{to})
\end{aligned}$$

– *Right-Shift* ($\gg$): if the right-hand operand is not within 0 to 31, then we give up on trying to evaluate the precise range and return the full range.

$$\begin{aligned}
&\alpha_c^\omega = T_a^\omega \gg T_b^\omega \ := \ (\omega_b^{fr} < 0 \vee \omega_b^{fr} > 31 \implies \mathit{bound}(c))\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{to} < 0 \implies \omega_c^{fr} = \omega_a^{fr} \gg \omega_b^{fr} \wedge \omega_c^{to} = \omega_a^{to} \gg \omega_b^{to})\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{fr} < 0 \wedge \omega_a^{to} > 0 \implies \omega_c^{fr} = \omega_a^{fr} \gg \omega_b^{fr} \wedge \omega_c^{to} = \\
&\omega_a^{to} \gg \omega_b^{fr})\\
&\quad \wedge\ (\omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \wedge \omega_a^{fr} \geq 0 \implies \omega_c^{fr} = \omega_a^{fr} \gg \omega_b^{to} \wedge \omega_c^{to} = \omega_a^{to} \gg \omega_b^{fr})
\end{aligned}$$

– *Unsigned Right-Shift* ($\ggg$): we only calculate the precise range when the left-hand operand is positive and the right-hand operand is within 0 to 31. The resulting range is the same as calculating the operands with the rights-shift operator.

$$\begin{aligned}
&\alpha_c^\omega = T_a^\omega \ggg T_b^\omega \ := \ (\omega_a^{fr} < 0 \vee \omega_b^{fr} < 0 \vee \omega_b^{fr} > 31 \implies \mathit{bound}(c))\\
&\quad \wedge\ (\omega_a^{fr} \geq 0 \wedge \omega_b^{fr} \geq 0 \wedge \omega_b^{to} \leq 31 \implies \omega_c^{fr} = \omega_a^{fr} \gg \omega_b^{to} \wedge \omega_c^{to} = \omega_a^{to} \gg \omega_b^{fr})
\end{aligned}$$

  - *Bitwise-and* (&): the range is only analyzed if one side is a positive constant. The resulting range is from 0 to that constant.

    $$\begin{aligned}\alpha_c^\omega = T_a^\omega \mathbin{\&} T_b^\omega \ := \ & (\omega_a^{fr} = \omega_a^{to} \wedge \omega_a^{fr} \geq 0 \implies \omega_c^{fr} = 0 \wedge \omega_c^{fr} = \omega_a^{to}) \\ & \wedge (\omega_b^{fr} = \omega_b^{to} \wedge \omega_b^{fr} \geq 0 \implies \omega_c^{fr} = 0 \wedge \omega_c^{fr} = \omega_b^{to}) \\ & \wedge (\omega_a^{fr} = \omega_a^{to} \wedge \omega_a^{fr} < 0 \implies bound(c)) \\ & \wedge (\omega_b^{fr} = \omega_b^{to} \wedge \omega_b^{fr} < 0 \implies bound(c)) \\ & \wedge (\omega_a^{fr} \neq \omega_a^{to} \wedge \omega_b^{fr} \neq \omega_b^{to} \implies bound(c))\end{aligned}$$

  - *Bitwise-or* ($|$): no analysis is done for bitwise-or operations. We give up on evaluating the precise range for this operation.

    $$\alpha_c^\omega = T_a^\omega \mid T_b^\omega \ := \ bound(c)$$

  - *Bitwise-xor* ($\oplus$): no analysis is done for bitwise-xor operations. We give up on evaluating the precise range for this operation.

    $$\alpha_c^\omega = T_a^\omega \oplus T_b^\omega \ := \ bound(c)$$

- *Comparison Constraints* ($\alpha_c \ \lhd \ T_a \ comp \ T_b$): if the left-hand operand is $\perp$, or the right-hand operand is not an integral type, then α_c is the same as the left-hand type T_a. Otherwise, we attempt to estimate a more precise range of the resulting type, as described below.

  $$\begin{aligned}\alpha_c \ \lhd \ T_a \ comp \ T_b \ := \ & (\beta_a^{bot} \wedge \alpha_c = T_a) \vee (\neg\beta_b^{int} \wedge \alpha_c = T_a) \\ & \vee (\neg\beta_a^{bot} \wedge \beta_b^{int} \wedge \ \alpha_c \ \lhd \ T_a^\omega \ comp \ T_b^\omega)\end{aligned}$$

  - *Equals To* ($==$): the resulting range equals to the maximum of the two lower-bounds and the minimum of the two upper-bounds. If the two ranges do not overlap, then the result is $\perp$.

    $$\begin{aligned}\alpha_c \ \lhd \ T_a^\omega == T_b^\omega \ := \ & (\omega_a^{fr} \geq \omega_b^{fr} \wedge \omega_a^{fr} \leq \omega_b^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \beta_c^{int}) \\ & \wedge (\omega_a^{to} \geq \omega_b^{fr} \wedge \omega_a^{to} \leq \omega_b^{to} \implies \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int}) \\ & \wedge (\omega_b^{fr} \geq \omega_a^{fr} \wedge \omega_b^{fr} \leq \omega_a^{to} \implies \omega_c^{fr} = \omega_b^{fr} \wedge \beta_c^{int}) \\ & \wedge (\omega_b^{to} \geq \omega_a^{fr} \wedge \omega_b^{to} \leq \omega_a^{to} \implies \omega_c^{to} = \omega_b^{to} \wedge \beta_c^{int}) \\ & \wedge (\omega_a^{to} \leq \omega_b^{fr} \vee \omega_b^{to} \leq \omega_a^{fr} \implies \beta_c^{bot}))\end{aligned}$$

  - *Not Equals To* ($\neq$): if the right-hand operand can only be a single value and is equal to the lower-bound or the upper-bound of the left-hand operand, then that number is excluded in the resulting range. If the ranges of the two operands are the same, then the result is $\perp$.

    $$\begin{aligned}\alpha_c \ \lhd \ T_a^\omega \neq T_b^\omega \ := \ & (\omega_a^{fr} = \omega_a^{to} \wedge \omega_b^{fr} = \omega_b^{to} \wedge \omega_a^{fr} = \omega_b^{fr} \implies \beta_c^{bot}) \\ & \wedge (\omega_b^{fr} \neq \omega_b^{to} \vee \omega_a^{fr} \neq \omega_b^{fr} \vee \omega_a^{to} \neq \omega_b^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})\end{aligned}$$

$$\wedge\ (\omega_b^{fr} = \omega_b^{to} \wedge \omega_a^{fr} = \omega_b^{fr} \implies \omega_c^{fr} = \omega_a^{fr} + 1 \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
$$\wedge\ (\omega_b^{fr} = \omega_b^{to} \wedge \omega_a^{to} = \omega_b^{fr} \implies \omega_c^{to} = \omega_a^{to} - 1 \wedge \omega_c^{fr} = \omega_a^{fr} \wedge \beta_c^{int}))$$

- *Greater Than* ($>$): if the lower-bound of the left-hand operand is larger than the upper-bound of the right-hand operand, then the result range is the same as the left-hand operand. If the range of the left-hand operand is smaller than the range of the right-hand operand and does not overlap, then the result is $\perp$. Otherwise, the result is the lower-bound of the right-hand operand increase by 1 to the upper-bound of the left-hand operand.

  $$\alpha_c \lhd T_a^\omega > T_b^\omega := (\omega_a^{fr} > \omega_b^{fr} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_b^{fr} \geq \omega_a^{fr} \wedge \omega_b^{fr} < \omega_a^{to} \implies \omega_c^{fr} = \omega_b^{fr} + 1 \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_a^{to} \leq \omega_b^{fr} \implies \beta_c^{bot}))$$

- *Less Than* ($<$): if the upper-bound of the left-hand operand is smaller than the upper-bound of the right-hand operand, then the result range is the same as the left-hand operand. If the range of the left-hand operand is larger than the range of the right-hand operand and does not overlap, then the result is $\perp$. Otherwise, the result is the lower-bound of the left-hand operand to the upper-bound of the right-hand operand decreased by 1.

  $$\alpha_c \lhd T_a^\omega < T_b^\omega := (\omega_a^{to} < \omega_b^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_b^{to} > \omega_a^{fr} \wedge \omega_b^{to} \leq \omega_a^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_b^{to} - 1 \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_a^{fr} \geq \omega_b^{to} \implies \beta_c^{bot}))$$

- *Greater Then and Equals To* ($\geq$): if the lower-bound of the left-hand operand is larger than the upper-bound of the right-hand operand, then the result range is the same as the left-hand operand. If the range of the left-hand operand is smaller than the range of the right-hand operand and does not overlap, then the result is $\perp$. Otherwise, the result is the lower-bound of the right-hand operand to the upper-bound of the left-hand operand.

  $$\alpha_c \lhd T_a^\omega \geq T_b^\omega := (\omega_a^{fr} \geq \omega_b^{fr} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_b^{fr} \geq \omega_a^{fr} \wedge \omega_b^{fr} \leq \omega_a^{to} \implies \omega_c^{fr} = \omega_b^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int})$$
  $$\wedge\ (\omega_a^{to} < \omega_b^{fr} \implies \beta_c^{bot}))$$

- *Less Then and Equals To* ($\leq$): if the upper-bound of the left-hand operand is smaller than the upper-bound of the right-hand operand, then the result range is the same as the left-hand operand. If the range of the left-hand operand is larger than the range of the right-hand operand and does not overlap, then the result is $\perp$. Otherwise, the result is the lower-bound of the left-hand operand to the upper-bound of the right-hand operand.

$$\begin{aligned}\alpha_c \ \lhd\ T_a^{\omega} \le T_b^{\omega} \ :=\ & (\omega_a^{to} \le \omega_b^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_a^{to} \wedge \beta_c^{int}) \\ & \wedge (\omega_b^{to} \ge \omega_a^{fr} \wedge \omega_b^{to} \le \omega_a^{to} \implies \omega_c^{fr} = \omega_a^{fr} \wedge \omega_c^{to} = \omega_b^{to} \wedge \beta_c^{int}) \\ & \wedge (\omega_a^{fr} > \omega_b^{to} \implies \beta_c^{bot}))\end{aligned}$$

In annotate mode, Value Inference generates additional breakable clauses that express preferences. The solver will attempt to satisfy all breakable clauses. The additional breakable clauses (enclosed in []) with weight (indicated by the index after []) for the well-formed and subtype constraints are:

- Well-formed Constraint ($wf(\alpha)$): the breakable constraints prefer solutions that contain an integral range for integral primitive types, and prefer solutions that are $\top$ or $\bot$ for all other types.

  $wf(\alpha_t) \quad := \ldots \wedge [\beta^{int} \wedge pf(t)]_1$ where $t = byte, short, char, int, long$

  $wf(\alpha_t) \quad := \ldots \wedge [\neg\beta^{int}]_1$ where $t \neq byte, short, char, int, long$

  Predicate $pf(t)$ prefers that the constraint variable α_t is the max range of data type t if there are no mandatory constrictions on the variable. This ensures the inferred types provide a more accurate representation of the integral range for user inspection, not some arbitrary value.

  $$\begin{aligned} pf(byte) &:= \ \omega^{fr} = -128 \wedge \omega^{to} = 127 \\ pf(short) &:= \ \omega^{fr} = -32768 \wedge \omega^{to} = 32767 \\ pf(char) &:= \ \omega^{fr} = 0 \wedge \omega^{to} = 65535 \\ pf(int) &:= \ \omega^{fr} = -2147483648 \wedge \omega^{to} = 2147483647 \\ pf(long) &:= \ \omega^{fr} = long_{min} \wedge \omega^{to} = long_{max} \end{aligned}$$

- Subtype Constraint ($T_a <: T_b$): the breakable constraint prefers that the subtype and the supertype are equal.

  $T_a^{constant} <: T_b^{variable} := \ldots \wedge [T_a = T_b]_4$

  $T_a^{variable} <: T_b^{constant} := \ldots \wedge [T_a = T_b]_3$

  $T_a^{variable} <: T_b^{variable} := \ldots \wedge [T_a = T_b]_2$

The preference constraints in the subtype constraint weights more than the preference constraints in the well-formed constraints because t is preferred $\top$ for five of the data types. A subtype constraint would infer the solution of a float or double to a type of `IntRange`, and should be the preferred solution as it is more precise. The reason why *constant* <: *variable* weights more than *variable* <: *constant* is that for cases where a variable is directly connected to both a supertype constant and a subtype constant, the subtype constant is preferred to be the solution of this variable as it contains a more precise value. *variable* <: *constant* weights more than *variable* <: *variable* because it is preferred for the solution of `from` and `to` to be specified values rather than random values.

4.5 Implementation

Value Inference for Integral Values is implemented as a type inference using Checker Framework and Checker Framework Inference. Excluding empty lines and comments, 3976 lines of Java code are used to implement this inference. The additional annotations provided for the byte-code libraries are the same as the ones used for the Narrowing and Widening Type System described in Sec. 3.4 to ensure consistency between the experiments.

The Checker Framework Inference does not generate comparison constraints and comparison constraint variables for refining the variables in a condition expressions prior to this type system, as it was not necessary for previous type systems built using the Checker Framework Inference. For most type systems, condition expressions will not change the type of the variable after evaluation, unlike value types. 393 lines of code are added to the Checker Framework Inference for the framework to support comparison constraints.

4.6 Experiments

This case study is performed on a cloud instance with a four-core CPU and 16GB RAM, running 64-bit Ubuntu 20.04.

Inference is run and all 18 Apache Commons projects from Table 3.2. Table 4.1 summarizes the numbers of integral primitive types, narrowing operations between primitives, arithmetic operations, and comparison operations used in each project. The integral primitives include the declared variables, parameters, and method returns. Table 4.2 shows the number of constraint variables and mandatory constraints generated for each of the 18 projects. Out of these 18 projects, inference successfully inferred 5 of the projects while the other 13 projects reached UNSAT. The results are presented in Table 4.3. The 5 projects are then evaluated in annotation mode and the results are presented in Table 4.4.

Table 4.3 also reports the execution times of the Value Inference for Integral Values in whole-program inference mode for the 18 projects. Table 4.4 reports the execution times of

Project	Primitives	Boxed	Cast	Arithmetic	Comparison
common-bcel	3142	89	101	478	1353
commons-beanutils	985	253	212	337	1155
commons-bsf	308	39	139	83	237
commons-codec	948	37	80	338	502
commons-compress	3728	347	584	1223	1809
commons-crypto	510	7	27	98	166
common-csv	254	40	36	39	190
commons-email	113	2	4	27	184
commons-exec	262	1	21	55	181
commons-fileupload	323	3	35	49	155
commons-imaging	4251	282	416	1808	2201
commons-io	1579	71	335	470	862
commons-jxpath	999	5	16	262	1259
commons-logging	140	11	0	39	324
common-net	2185	36	207	516	1021
commons-ognl	848	39	103	213	976
commons-text	858	58	289	339	609
commons-validator	737	737	24	327	635
Total	23475	2207	2710	7245	15520

Table 4.1: Summary of the numbers of integral primitive, integral primitive wrappers, narrowing conversions between primitives, arithmetic operators, and comparison operators used in each projects

	Constraint Variables						**Constraints**			
Project	α_{cons}	α_{var}	α_{refin}	α_{lub}	α_{op}	α_{comp}	Σ_{sub}	Σ_{eq}	Σ_{op}	Σ_{comp}
commons-bcel	298	18551	4523	3257	908	3084	43132	5241	909	1758
commons-beanutils	49	7721	1483	1341	136	1336	12988	2601	136	1190
commons-bsf	87	2980	2980	739	273	588	6162	1427	274	488
commons-codec	4786	4800	1340	1258	1082	796	14892	1912	1082	614
commons-compress	2955	13460	4569	4452	2633	3928	36196	5771	2633	2370
commons-crypto	29	1834	404	304	113	304	2951	608	113	254
commons-csv	40	891	247	357	67	306	2205	372	67	184
commons-email	46	1246	267	263	27	194	1955	505	27	144
commons-exec	36	1045	275	260	34	192	1855	496	34	140
commons-fileupload	119	1481	376	380	102	310	2939	589	102	208
commons-imaging	1072	20753	6796	5597	4618	4820	50588	8633	4618	3158
commons-io	131	5943	1286	1726	556	1594	11859	2228	556	1272
commons-jxpath	455	8598	2928	3189	707	2814	25482	3747	707	1870
commons-logging	29	1252	328	343	15	308	2475	568	15	254
commons-net	286	9357	2912	3017	723	2156	20984	4081	723	1396
commons-ognl	262	9455	2649	2562	565	2094	21365	3660	565	1294
commons-text	77	3909	1084	1499	624	1422	9911	1461	624	1068
commons-validator	125	4227	829	1142	169	948	7984	1457	169	728
Total	11144	128856	38603	35223	14267	27486	309506	46729	14268	20294

Table 4.2: Constraint variables and constraints generated for each of the 18 Apache Commons projects.

	SMT			**Inferred Values**			
Project	Bool	Int	Assert	$\top$	$\bot$	`Int`	**Time**
commons-bcel	180558	120372	163366	UNSAT			320.94
commons-beanutils	37086	24724	29294	2625	238	2355	218.83
commons-bsf	34356	22904	28340	UNSAT			54.86
commons-codec	56724	37816	56406	UNSAT			91.33
commons-compress	176850	117900	155158	UNSAT			379.40
commons-crypto	18096	12064	14088	UNSAT			38.79
commons-csv	11424	7616	9544	UNSAT			59.70
commons-email	6120	4080	4650	337	97	517	47.81
commons-exec	5640	3760	4414	255	23	404	38.41
commons-fileupload	16248	10832	13168	UNSAT			33.86
commons-imaging	258558	172372	222960	UNSAT			447.80
commons-io	67650	45100	55072	UNSAT			104.48
commons-jxpath	110970	73980	101334	UNSAT			171.18
commons-logging	6831	4554	5599	438	64	421	49.70
commons-net	110700	73800	91874	UNSAT			165.22
commons-ognl	111612	74408	91312	UNSAT			223.19
commons-text	52464	34976	44736	UNSAT			106.34
commons-validator	22281	14854	17990	1246	399	1536	129.53
Total	1260807	946648	1231697	4901	821	5233	3704.22

Table 4.3: Inference mode results for the projects. Column Assert shows the number of (mandatory) formulas encoded for SMT. The inferred values are the solutions given by the solver in inference mode. The time column reports the total time taken in seconds by Value Inference for Integral Values to encode a constraint system into SMT formulas and for Z3 to solve the formulas in inference mode.

	SMT	**Inferred Values**			**Time**	
Project	Soft Assert	$\top$	$\bot$	`Int`	Encode	Solve
commons-beanutils	38321	3969	194	1048	12.762	33792.66
commons-email	5918	592	69	263	3.153	698.74
commons-exec	5590	444	49	205	4.105	476.10
commons-logging	7227	643	43	214	2.732	362.83
commons-validator	23382	2177	876	123	5.282	31331.86
Total	80438	7825	1231	1583	28.034	66662.19

Table 4.4: Annotate mode results for the projects that passed inference mode. Column Soft Assert shows the number of breakable formulas encoded for MaxSMT. The inferred values are the solutions given by the solver in annotation mode. The Encode and Solve columns, respectively, report the time taken in seconds by Value Inference for Integral Values to encode a constraint system into SMT formulas, and for Z3 to solve the formulas in annotation mode. The Encode column does not count the time required to traverse the AST to generate constraints

the Value Inference for Integral Values in whole-program inference annotation mode for the 5 successfully inferred projects. The time is separated into encoding time and solving time, which respectively represent the time taken by the Value Inference for Integral Values to encode a constraint system into SMT formulas, and the time for Z3 to solve those formulas.

When a program reaches UNSAT, names are added to each formula so that the solver can return a minimal subset of the unsatisfiable constraints which are used by Value Inference to issue errors. The reasons why these projects reached UNSAT are manually analyzed. Overall, there are three reasons for a project to reach UNSAT in inference mode:

- **Presence of narrowing type errors**

  All failed projects in the experiments contain a UNSAT constraint where the range and precision are lost from the narrowing of primitives. They do not check the range of the variable before narrowing. These errors can lead to serious failures if the libraries were used incorrectly in mission-critical systems. Value Inference for Integral Values helps identify such issues and forces developers to restructure their applications in a way that allows the safe use of narrowing conversions on primitive values.

- **Insufficient annotations for library methods**

  Applications have large dependencies on binary-only libraries, like the JDK. Value Inference for Integral Values can infer annotations for the available source code, but requires manual annotations for binary-only dependencies. Value Inference for Integral Values can use optimistic defaults, for example by assuming that unannotated method signatures have $\top$ receiver and parameter types, and $\perp$ return types. Optimistic defaults can temporarily help a developer pinpoint whether the problem is in their code or is due to an unannotated API. Manual annotation efforts for libraries can be re-used to type check and infer units in other projects utilizing the same APIs.

- **False positives**

  The false positives mentioned in Sec. 3.5.3 also contributed to the UNSAT produced by the solver as the constraints generated are limited to its type system. These include imprecision for loop iterations, unable to handle post-conditions on external function checks, and imprecision with functions in Math and Number libraries.

4.6.1 Performance Overhead

In inference mode, the 5 projects, `commons-beanutils`, `commons-email`, `commons-exec`, `commons-logging`, and `commons-validator` that successfully pass whole-program inference, take 484.28 seconds (8.07 minutes) to check and infer a type-safe solution in inference mode. The time taken to type check these 5 projects is 102.79 seconds (1.71 minutes) as shown in Table 3.2. The overhead is approximately 4.71x compared to modular type checking. The solving time generally takes longer for larger projects, but if the solver finds an unsatisfiable constraint early, the solving time can be fast.

In annotate mode, successful projects take 66690.224 seconds (1111.50 minutes) to infer the more precise and relevant values. `commons-beanutils` and `commons-validator` contribute the most to the tally. The overhead is approximately 650.00x vs. type check mode, and 137.71x vs. inference mode. The time is spent by the SMT solver to optimize the solutions. Value Inference for Integral Values takes more time to generate SMT encoding for larger projects. The major performance bottleneck is the SMT solver. The solving time increases dramatically with the increased numbers of soft asserts.

The performance in inference mode for whole-program satisfiability checking enables it to be used in continuous integration workflows. The performance in annotate mode is slow, but this mode is expected to be used less frequently. The annotations inserted into source code enable subsequent uses of the Value Inference for Integral Values in inference or annotate modes to be executed faster, as fewer constraint variables are generated. Developers can also incrementally infer and annotate their projects, starting with core libraries. The Value Inference for Integral Values performs adequately in inference and annotation mode. Its performance is suitable for use in a real-world software development environment. Improvements to SMT encoding, the Checker Framework, Checker Framework Inference, and the z3 solver will improve the performance of the Value Inference and other type inference developed using the frameworks. We can also experiment with different solvers for future work.

Chapter 5

PUnits - Units Type System Improvement and Related Works

5.1 Introduction

PUnits is an expressive type system to enforce units of measurement and a precise whole-program type inference approach that helps developers annotate code with unit types. *PUnits* is implemented for Java as an optional type system [8]. It handles all of Java's language features and works on real-world Java applications. Nevertheless, the idea for *PUnits* is not limited to Java and can be widely adopted.

This type system is a collaborative work with Ph.D. student Jeff Yucong Luo. Jeff designed and implemented the normalized vector representations, the core type features, and the type inference of the Unit Type system. A brief background on this type system is described in Sec. 2.4. This chapter focuses on the additional features added to the system and how *PUnits* compares to existing unit systems. This chapter is part of the paper published in OOPSLA 2020 [46].

PUnits supports annotation defaulting, method-local flow-sensitive type refinement, and parametric polymorphism over units. These features minimize the need for unit annotations in method bodies, removing unnecessary clutter. This chapter introduces a new feature that is added to *PUnits*, receiver-dependent units, in Sec. 5.2, for handling cases where the method return and method parameter type are context-sensitive to the method receiver type.

Two general approaches have been applied in prior work to add support for units to a programming language: (1) through designing abstract data types and libraries to encapsulate numbers with units, (2) through modifying a language to add unit syntax and static analysis.

The encapsulation approach uses the existing type system of a language to perform limited static analysis, with the capability to perform fully run-time-based unit analysis for units and quantities that are given as run-time inputs. The design and features of 38 different units libraries are extensively compared in a recent paper [3]. Using such libraries incurs additional performance and memory overhead. For example, using boxed number types for numeric computations in Java is 3x slower and uses 3x more memory compared to using primitive types [37]. JSR 275/363 [15, 16] is the most popular option for Java. It defines an API for units of measurement and provides a reference implementation for SI units. Sec. 5.3 discusses the benefits and trade-offs of using *PUnits* versus such libraries.

Sec. 5.4 compares the static approaches in prior works.

5.2 Receiver Dependent Units

A return type or method parameter type sometimes needs to be sensitive to the actual method receiver type. *PUnits* supports receiver-dependent units by introducing the `@RDU` type qualifier for return types or method parameter types. A `@RDU` type is resolved using the actual receiver type, similar to how viewpoint adaptation works in ownership types [19, 20]. The viewpoint adaptation operation takes two inputs, the receiver type and the declared type, and yields a single result type. When type checking a method invocation, any occurrence of `@RDU` in the signature is substituted with the receiver type; all non-`@RDU`-types stay unchanged. Type checking happens against the viewpoint adapted signature of the method. `@RDU` is never inferred. In inference and annotate mode, any occurrence of `@RDU` in the signature introduces a new constraint variable and an equality constraint is introduced between the receiver type and the new constraint variable. This constraint variable is used for further checks instead of the declared parameter or return type.

The type rule of receiver dependent unit is as follows:

```
q ▷ RDU = q
_ ▷ q = q (otherwise)
```

Consider the example in Fig. 5.1. The return type of `convert()` on line 4 and the parameter type of `toNanos()` on line 5 are `@RDU`. When these methods are invoked, we enforce the return type of `convert()` and the argument to `toNanos()` to be the same as

```
 1 enum TimeUnit {
 2   @s SECOND,
 3   @ns NANOSECOND;
 4   @RDU long convert(long duration, TimeUnit unit) {...}
 5   @ns long toNanos(@RDU long duration) {...}
 6 }
 7 @s int good1 = SECOND.convert(10, NANOSECOND);
 8 @ns int bad1 = SECOND.convert(10, NANOSECOND); // Error
 9 @ns int good2 = SECOND.toNanos(s);
10 @ns int bad2 = SECOND.toNanos(ns); // Error
```

Figure 5.1: Receiver-dependent units examples. Variables s has unit type `@s` (second) and variable ns has unit type `@ns` (nanosecond).

```
1 class Unsound {
2   @RDU int field;
3 }
4 @m Unsound a = new @m Unsound();
5 @⊤ Unsound b = a;
6 b.field = (@s int) 0;
```

Figure 5.2: Example of unsoundness if receiver-dependent units were allowed to be used on member fields. `@RDU` is forbidden on any program element aside from method parameter and method return types.

their method receivers. An invalid type assignment is issued on line 8 as `SECOND.convert()` returns type `@s`. An invalid type argument is issued on line 10 as `SECOND.toNanos()` accepts type `@s` as its argument.

Note that we forbid the use of `@RDU` on any program element aside from method parameters and method returns; otherwise soundness is not guaranteed. Fig. 5.2 shows an example of unsoundness if `@RDU` is used on a member field. At line 6, `b.field` is assigned a `@s` value, which is visible to `a.field` since object b is an alias of object a, and breaks that `a.field` should be `@m`.

5.3 JScience vs. *PUnits*

We discuss the benefits and trade-offs of using *PUnits* versus existing Java unit libraries from three aspects: error detection, program execution performance (time and memory), and features.

We focus on analyzing *PUnits*'s effects on the GasFlow project, the only project that uses a unit library. GasFlow uses the JScience library, or JSR 275/363 [15, 16], which is one of the most popular unit libraries for Java. We replace uses of JScience unit wrapper classes with *PUnits* specifications and primitive types and run modular type checking on the annotated code.

The JScience API uses generics to provide unit type-safety. The `Amount<Quantity`[1]`>` class in JScience is used for storing the exact `Unit`[2] and performing arithmetic operations with the units. Table 5.1 give a summary of the ranges of dimensions and units used in the GasFlow project.

GasFlow only uses ten functionalities within the JScience library:

- Initialization: `valueOf(double, Unit)` and `valueOf(String)` can take in a double and its corresponding `Unit` or a string in the format of `"value unit"` (e.g., `"50 m"`). 181 uses.
- Conversion: `to(Unit)` and `doubleValue(Unit)` return the value converted to the unit specific by the argument. It is required for the argument to be in the same dimension as its receiver. 81 uses.
- Arithmetic: `plus(Amount)`, `minus(Amount)`, `times(Amount)`, and `divide(Amount)` perform arithmetic operations between two Amounts. Plus and minus operations require the receiver and the argument to be in the same dimension. 236 uses.
- Comparison: `isGreaterThan(Amount)` and `isLessThan(Amount)` perform comparison operations between two Amount. 12 uses.

To replace the abstract data types with primitives, we create helper class `UnitsTools`, which stores a list of units that are represented using annotated primitives with a value of 1 (eg. `@m int m = 1`). All the wrapper initialization are replaced with UnitsTools

[1]Part of the `javax.measure` package, provides dimension handling.

[2]Part of the `javax.measure` package, provides unit handling.

annotated primitives. For example, `Amount.valueOf(2, METER)` is replaced with `2 * UnitsTools.m`. All the wrapper arithmetic operations are replaced with the standard Java operations +, -, *, /. All the wrapper comparison operations are replaced with the standard Java operations ¿ and ¡. Initialization, arithmetic operations, and comparison operations are replaced using a small script with sed. For conversion, we manually ensured that the variable is in the unit specified by the argument by specifying the declaration to its specified unit. For example, `Amount m = amountVal.to(METER)` becomes `@m double m = amountVal` with `amountVal` now a primitive. `UnitsTools` also contains conversion methods that are used to replace unit conversion functions in JScience when it is possible to determine the type to be converted statically. All the unit conversions that appear in the GasFlow project can be determined statically.

In total, 510 JScience method invocations and 503 variables containing 17 dimensions and 22 units are replaced with 647 *PUnits* qualifiers and 242 `UnitTools` uses. Additional *PUnits* qualifiers were needed for casting unit-wise heterogeneous methods to specific units.

Error Detection

After replacing unit wrappers with *PUnits* specifications and primitive types, we used *PUnits* in type checking mode to see whether there are any units errors. We found three units errors, whereas JScience failed to detect them.

Two of these errors are related. Function `computeSiamCoefficient` is declared as dimensionless. However, the return type is $m^{-1}s^{-2}$ and an invalid return error is issued. This function is then invoked by a function that requires the return type to be $m^{-1}s^{-2}$, but as `computeSiamCoefficient` is declared as dimensionless, another return type error is issued. We changed the return type to $m^{-1}s^{-2}$ to fix these two errors. The other error is related to function `getReynoldsNumber`. A Reynolds Number [4] is a dimensionless value, but the function returns type m^3/kg. This function is missing the density (volume per weight) variable to give the correct Reynolds Number. This error causes all future computations in this project to be incorrect.

The reason why JScience cannot detect these errors is due to the use of casting and raw types. The original GasFlow contains 130 raw unit data types and 51 unit casts. Line 3 in Fig. 5.3 illustrates an illegal cast of an area to a length. The return type of `times()` is `Amount<?>`, because JScience cannot statically express the return unit. Developers are therefore required to add casts that cannot be checked statically, to make the result usable. Similarly, on line 4, a raw type is used to circumvent this weakness. The cast from a wildcard to a type produces an unchecked warning, as does the usage of a raw type. 181

Dimension	**Unit**	**Original**			***PUnits***	
		Amount	Javax	String	Anno	UTools
Length	`m`	75	79	1	46	25
	`mm`		5	1	26	17
	`km`		2	0	0	1
Pressure	`Pa`	90	3	0	0	3
	`Ba`		46	3	111	31
Duration	`s`	16	24	12	14	11
	`hr`		4	0	0	48
Temperature	`K`	18	8	1	18	14
	`°C`		1	2	3	3
Mass	`g`	2	4	0	0	13
	`kg`		16	0	6	9
Angle	`rad`	0	0	0	10	0
Velocity	`m/s`	16	4	1	11	5
Area	$\texttt{m}^2$	4	0	0	1	2
Volume	$\texttt{m}^3$	12	7	0	9	59
Power	`W`	4	4	0	4	1
Mass Flow Rate	`kg/s`	8	2	0	14	0
Molar Mass	`g/mol`	5	2	2	12	0
Volumetric Flow	$\texttt{m}^3\texttt{/hr}$	71	0	52	49	0
Calorific Value	$\texttt{MJ/m}^3$	3	0	1	5	0
Heat Transfer Coefficient	$\texttt{W/m}^2\texttt{/K}$	3	0	0	3	0
Dimensionless	-	46	21	0	39	0
Raw Data Type	-	130	9	0	266	0
Total	-	503	220	105	647	242

Table 5.1: A summary of all the dimensions and units used in the GasFlow project. Column Amount counts the uses of `Amount<Q>`. Column Javax counts uses of `javax.measure.unit` and column String counts units represented in string format. Column Anno counts unit qualifiers after converting to use *PUnits*. The qualifiers replace the `Amount<Q>` uses in the project. Column UTools counts uses of `UnitsTools` that are used to replace `javax.measure.unit` and units in string format. Velocity can be represented as `UnitsTools.m/UnitsTools.s` and therefore is counted toward `m` and `s` instead of `m/s`.

```
 1 Amount<Length> bad() {
 2     Amount<Length> l;
 3     l = (Amount<Length>) valueOf(1,M).times(valueOf(1,M));
 4     Amount a = valueOf(1, M).times(valueOf(1, M));
 5     return a;
 6 }
 7
 8 @m double good() {
 9     double l = (@m double) 1*UnitsTools.m * 1*UnitsTools.m;
10     double a = 1*UnitsTools.m * 1*UnitsTools.m;
11     return a;
12 }
```

Figure 5.3: A simple example comparing the difference between JScience and *PUnits*. `M` is unit `METER`. Errors are issued on line 9 and 11 by *PUnits*.

such warnings are produced in GasFlow. Making matters worse, no runtime exception is raised within method `bad()`, as all method invocations and casts are valid. However, even though the type argument is incorrect, the unit is still preserved dynamically. If we try to read the return value of `bad()` in a unit that is in the dimension of the signature on line 1, such as meter or kilometre, a runtime exception will be raised. As such invocations can happen a long time after the call to `bad()`, it is difficult to pinpoint where the incorrect `Amount` object came from.

With *PUnits*, this kind of casting will not be allowed since `m` is not a subtype of $\texttt{m}^2$. Not only that, with *PUnits*, such casts are not needed. Variable *a* has type $\texttt{m}^2$ and it is not a subtype of `m`. An invalid cast error will be issued on line 9 and an invalid return type will be issued on line 11. *PUnits* is able to detect errors early and it pinpoints the exact location of the problem.

Execution Time & Memory Consumption

We executed the GasFlow project by invoking the main method of the program. The program calculates and generates data points at every time step over a fixed amount of time. The time step was originally set to 1 second. To better analyze its performance, we decrease the time step to 1 millisecond to increase the program execution time. This case study is performed using a laptop with Intel i7-6700HQ 2.60 GHz a four-core CPU and 16GB DDR4 RAM, running 64-bit Ubuntu 18.10.

The average execution time is 3966ms when we are using JScience, and 3367ms when using *PUnits* and all the abstract data types are replaced with primitives. The execution time has significantly reduced by 15.1%. The execution time is measured using Java's System library.

The average memory consumption is 794 kBytes when we are using JScience, and 697 kBytes when using *PUnits* with primitive types. The memory consumption has significantly reduced by 12.2%. The memory consumption is measured using Java's Runtime library.

The performance overhead of `UnitsTools` is insignificant compared to JScience, as all of the `UnitsTools` operations are using primitives. As *PUnits* is a static analysis tool, it will not add any overhead to the program during runtime and reap the benefits of using primitives, unlike JScience which uses abstract data types to keep track of units.

Features

GasFlow contains one heterogeneous method, which accepts or returns a range of dimensions that differ in its specific class or type argument. For this method, *PUnits* loses the unit of that variable, while JScience preserves the unit dynamically through abstract data types. Nevertheless, we believe it is good coding practice to avoid type-unsafe heterogeneous methods when possible, as heterogeneous methods increase the risk of a runtime exception. *PUnits* enforces this programming practice.

PUnits uses primitives instead of abstract data types, which also prevents problems with null values. As a value should always have a unit, we do not see this as a restriction. One simple way to allow the primitives to have a null-like behaviour is by creating a boolean flag for each variable to indicate if a variable has been initialized or not. In GasFlow, there are five variables that required this work-around.

It is possible to combine *PUnits* with a units library, by annotating the abstract data types of the units library with *PUnits* specifications to gain the benefits of both static error detection and dynamic library features. However, this will lose the performance benefits of using primitive types. Overall, for projects that require minimum uses of dynamic unit computations, *PUnits* is a better alternative than JSR 275/363.

5.4 Related Work in Static Units Systems

The static approaches in prior work differ in how units are internally represented and their expressiveness, how types are inferred, and how annotation burden is reduced. Fig. 5.4

	PUnits	**Osprey**	**F#**	**Simulink's**	**B's**	***ableC*'s**	**CPF[UNITS]**
Dimension analysis	✓		✓	✓			✓
Unit-wise analysis	✓	✓	✓		✓	✓	✓
Support non-SI	✓		✓		✓		✓
Polymorphism	✓	✓	✓	✓		✓	
Receiver dependent	✓						
$\top/\bot$ (top/bottom)	✓				✓		
`<kN/km>` = `<N/m>`	✓	✓		✓			✓
Modular type inference	✓	✓	✓	✓	✓		✓
Whole-program inference	✓						
Reduce annotation burden	✓	✓	✓				

Figure 5.4: An overview of features provided by each unit of measurement system.

gives an overview of the supported features between the related works.

Osprey is a constraint-based units type checker for C [**?**]. *PUnits*'s representation of units is similar to Osprey's. Osprey introduces a units language whereby compound units are represented as products and inverses of SI base units, a "dimensionless" unit, and constant factors. Osprey does not allow developers to utilize a different set of base units, and stores only the exponents of the SI units. It also does not have a $\top$ or $\bot$ type. There is no subtyping between units in Osprey. Assignments, additions, subtractions, and comparisons all require the arguments' units to be equal to each other while *PUnits* allows more flexibility with $\top$ and $\bot$. Any return value and parameter without annotations are treated by Osprey to be polymorphic while in *PUnits*, functions that are intended to be polymorphic must be explicitly annotated. Osprey's minimized constraint set is translated into a system of linear equations and then solved using Gaussian elimination. We also implemented a Gauss-Jordan elimination solver for the integer subsets of the constraints. However, the initial performance results of this solver did not look promising and we did not pursue the effort further. To reduce annotation burden, Osprey can suggest "critical" variables for users to annotate. These are variables whose units are inconsistent with the units of their representatives. In *PUnits*, if inference fails, the set of constraints causing unsatisfiability is provided. The user can examine the constraints and determine whether there is a unit error or there are insufficient annotations in the binary-only libraries.

F#'s units type system [45, 32, 31] represents units as symbols or products of symbols, each with an optional integer exponent. The set of symbols can be declared by the developer, however, there is no support for representing prefixed units in terms of their base units, as prefixed units are considered distinct unit symbols. Consequently, a value with unit `<kN/km>` cannot be assigned to a variable with unit `<N/m>` in F# whereas it is permitted in *PUnits*. F# infers under-constrained type variables as polymorphic unit types, the most general typing in F#. In *PUnits*, the most general type is $\top$, which is not a useful annotation to insert into source code. *PUnits* chooses to infer the most precise type, and prefers to infer `@Dimensionless` for under-constrained type variables. Similar to *PUnits*, functions that are intended to be polymorphic must be explicitly annotated. F# allows polymorphic functions to express more rich relationships such as polymorphic multiplication and division where the resulting unit is a function of two or more polymorphic units given as the parameters. *PUnits* currently supports a less expressive form of polymorphism, but scaling the current design to support F# style polymorphism is interesting future work. We did not need F# style polymorphism in the case studies, but plan to extend our implementation as future work.

F#'s inference algorithm was improved upon in a type system for Fortran [39]. This system makes two improvements. First, it always assumes that functions without any unit annotations are implicitly polymorphic over the units. It infers the unit relationships between a function's parameters and its return as a general relationship, and then infers the specific units at each call-site. Second, under-constrained variables are reported as critical variables that require manual annotation by a user. Critical variables provide the maximal amount of unit information with the minimal number of explicit annotations. This improvement reduces some annotation burden. *PUnits* inserts all inferred non-default units into source code, reducing the annotation burden for subsequent type checking or inference.

A dimensional analysis system for Simulink [**?**] expresses dimensions as products of SI base dimensions, each with an integer exponent. Dimensional consistency can be enforced through this representation. However, unit-wise consistency cannot be enforced: an expression of 1 `km` + 1 `m` would not result in any errors as both are in the dimension of length. A proper calculation would require either unit to be converted prior to the addition. This system infers dimensions by collecting equations from the program and then solving the set of equations through Gauss-Jordan elimination.

A units type system for the B language [33] represents units as products of SI units where each base unit has a prefix and an exponent. Some derived units can be expressed via multiple representations in this design. For example, a kilo-newton can be defined as both $\texttt{Mg} * \texttt{m} * \texttt{s}^{-2}$ and $\texttt{kg} * \texttt{km} * \texttt{s}^{-2}$ with corresponding triples. This leads to a combinatorial explosion of representations since multiplication and division produce new units as a function of the units of its two arguments. *PUnits* extracts all prefixes into one constant, forcing each unit to have a unique normalized representation, which avoids the combinatorial explosion problem. This type system does not handle polymorphism and all unannotated units are treated as $\perp$. It uses a simple unification-based type inference, in addition to constraint solving for multiplication, division, and exponentiation operations. These constraints cannot be collected and are solved one at a time. The $\top$ qualifier in this unit type system is used to indicate a type error as only one inferred unit is acceptable. In *PUnits*, $\top$ is accepted to accommodate heterogeneous methods and arrays.

Type Qualifiers as Composable Language Extensions [12] proposes adding pluggable type checkers to the *ableC* [29] language as grammar and type checking extensions, including a units of measurement system. In that system, units are parsed as symbols as part of an extended *ableC* grammar, and then represented as a set of tuples similar to the units type system for the B language[33] for its type checking phase. Each tuple is of the form $[b^p * u^e]$ for some conversion factor b raised to the power of p and some base unit u raised to the power of e. In *ableC*, units are not organized in a type lattice and there is no concept of $\top$ or $\perp$ types. *ableC*'s units type system is implemented exclusively for SI units, and does not support other units of measurements without further extending the grammar and implementing the extension. The pluggable type systems of *ableC* lack type inference capabilities.

CPF[UNITS] [27] defines a units analysis policy, annotation language, and specification that plugs into the C Policy Framework (CPF) to debug and verify C programs. The system encodes the abelian group properties of units as a rewriting system. Units are represented as products of base units, each with an exponent. The set of base units is customizable and can be given long and short names. Users provide annotations that specify the units of variables, objects, and function parameters and returns as pre- and post-conditions. Unannotated function parameters and objects are treated as having a "fresh" unique unit by the system to eagerly prevent misuses. The framework extracts one verification task per function. The tasks are solved through symbolic execution, using the rewriting system to solve unit relationships. Units are checked at key operations such as additions and comparisons. The system is modular, and can infer units for local variables. However, it does not perform whole-program inference.

Chapter 6

DataFlow Framework Precision Improvements

6.1 Introduction

Data-flow analysis is considered a terminating, imprecise abstract interpretation of a program [5]. Having an analysis that is decidable is important, e.g. for compiler optimization. The Checker Framework uses the Dataflow Framework for program verification, which is a data-flow analysis of Java programs. The Dataflow Framework for the Java programming languages is used to estimate the values a variable might contain. It is a path insensitive analysis. The Dataflow Framework transforms the abstract syntax tree from an input Java program, and turns it into a control-flow graph. The background of the data-flow analysis and the structure of the Dataflow Framework is described in Sec. 2.1.

Sec. 3.5.3 mentions one of the false positives issued by the Narrowing and Widening Checker is due to loop imprecision. To resolve this false positive, one of the solutions is making changes to algorithms of the Dataflow Framework. This chapter first explores other kinds of false positive results from the Dataflow Framework's imprecise algorithm and solutions in improving its precision. Sec. 6.2 presents the addition of a dead branch analysis to the Dataflow Framework to resolve the false positive show in Fig. 6.1. Sec. 6.3 proposes a path-sensitive analyse to resolve the false positive show in Fig. 6.3. Sec. 6.4 then describes how these two additional analyses to the Dataflow Framework resolve some loop imprecision such as the example shown in Fig. 6.5.

```
@IntVal(1) int dead_branch() {
    int x = 0;
    int y = 0;
    if (x == 0) {
        y = 1;
    }
    return y;
}
```

```
Error: [return.type.incompatible] incompatible types in return.
  type of expression: @IntVal({0, 1}) int
  method return type: @IntVal(1L) int
```

Figure 6.1: An example of a false positive error due to lack of dead branch analysis.

6.2 Dead Branch Analysis

A branch is "dead" the branch is unreachable. The Dataflow Framework does not perform dead branch analysis. Fig. 6.1 is an example of a false positive error produced by the Constant Value type system due to the lack of dead branch analysis. After the boolean expression `x == 0`, two different stores are created. The then store maps `x` to `IntVal(0)` and the else store maps `x` to $\perp$ since no values goes there. The value of y is mapped to `IntVal(0)` in both stores. The stores are propagated to their respective branches. In the then-branch, y changes to `IntVal(1)` due to the expression $y = 1$. After the stores merged, y is mapped to `IntVal(0,1)` as it is the least upper bound of `IntVal(0)` and `IntVal(1)`. The checker then issues a `return.type.incompatible` error since `IntVal(0,1)` is not a subtype of `IntVal(1)`.

Fig. 6.2 shows the stores in the control-flow graph generated by the Dataflow Framework in every block. The special blocks are represented using circles. The regular blocks are represented using rectangles and the conditional blocks are represented using octagons. The expression beside the regular blocks are the nodes, or expressions, analyzed. The values in each block are the transfer results after analyzing all nodes within that block. The stores with a mapping of the variables to their type are shown within the block.

Dead branch analysis is added to the Dataflow Framework to improve its precision and reduces false positives. An indication of whether the branch is dead or not is added to the stores. Upon encountering a boolean expression, the transfer function will also determine

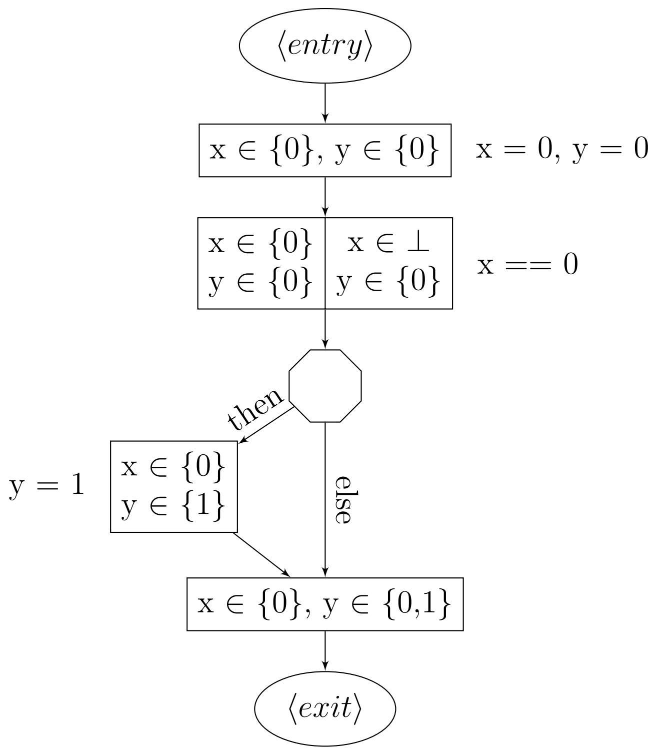


Figure 6.2: Control-flow graph of the example in Fig. 6.1 containing the transfer result in each block produced after evaluating the expressions.

the result of the expression to either be true, false, or unknown. If the expression can be evaluated to either true or false at compile-time, then there exists a dead branch in the program. During the merging of the two branches, incoming stores from the dead branch should not merge with the other store. Only the stores from the reachable branch are propagated and used for further analysis. Going back to Fig. 6.1, the store produced after the branches merge will be $\{x \in \{0\}, y \in \{1\}\}$ instead and the correctness of the program can be verified.

6.2.1 Implementation

A new flag is added to the Store interface in the Dataflow Framework to indicate whether this store is in a dead branch or not. If the flag returns `true`, then the store is in a dead branch.

```
public interface Store<S extends Store<S>> {
    public boolean isDeadBranch();
    public void setDeadBranch();
    /* ... */
}
```

A new class `ConditionEvaluator` is created for statically evaluating boolean expressions. This class visits a boolean expression and returns an enum indicating the path of the dataflow. If the boolean expression is evaluated to be always true, then the visitor will return `ConditionFlow.TRUE`. If the boolean expression is evaluated to be always false, then the visitor will return `ConditionFlow.FALSE`. If the path is unknown, then the visitor will return `ConditionFlow.BOTH`. If the `ConditionFlow` is evaluated to be either `TRUE` or `FALSE`, then the `deadBranch` flag in the other store is set to true during value propagation. When the branches merge and the least-upper-bound between the stores are calculated, the store with a `deadBranch` flag is ignored.

```
public class ConditionEvaluator<A extends AbstractValue<A>,
        S extends Store<S>> {
    public enum ConditionFlow {
        TRUE,
        FALSE,
        UNKNOWN
    }
    public ConditionFlow visit(Node node, TransferInput<A, S> in)
       { /* ... */ }
}
```

In total, 440 lines of code are added to the Dataflow Framework and the Checker Framework to support a dead branch analysis interface for other type systems.

6.2.2 Applying to Constant Value Type System

Including dead branch analysis for interval analysis is beneficial as it increases precision in the analysis. The following flow rules are applied to the `ConditionEvaluator` for the Constant Value Type System. T_L and T_R are the type qualifier of the left and right operators. from_L and from_R indicate the lower-bound of the left and right-hand operand and to_L and to_R indicate the upper-bound of the left and right-hand operand.

- Greater Than ($T_L > T_R$): if the two ranges do not overlap or if the only overlapping values are the upper bound of the left-hand operand and the lower bound of the right-hand operand, then we can be certain of the propagating path. Otherwise, the path is unknown.
  $\mathit{from}_L > \mathit{to}_R \implies$ `TRUE`
  $\mathit{to}_L \leq \mathit{from}_R \implies$ `FALSE`
  otherwise $\implies$ `UNKNOWN`

- Greater or Equal ($T_L \geq T_R$): if the two ranges do not overlap or if the only overlapping values are the lower bound of the left-hand operand and the upper bound of the right-hand operand, then we can be certain of the propagating path. Otherwise, the path is unknown.
  $\mathit{from}_L \geq \mathit{to}_R \implies$ `TRUE`
  $\mathit{to}_L < \mathit{from}_R \implies$ `FALSE`
  otherwise $\implies$ `UNKNOWN`

- Less Than ($T_L < T_R$): if the two ranges do not overlap or if the only overlapping values are the lower bound of the left-hand operand and the upper bound of the right-hand operand, then we can be certain of the propagating path. Otherwise, the path is unknown.
  $\mathit{to}_L < \mathit{from}_R \implies$ `TRUE`
  $\mathit{from}_L \geq \mathit{to}_R \implies$ `FALSE`
  otherwise $\implies$ `UNKNOWN`

- Less or Equal ($T_L \leq T_R$): if the two ranges do not overlap or if the only overlapping values are the upper bound of the left-hand operand and the lower bound of the right-hand operand, then we can be certain of the propagating path. Otherwise, the

path is unknown.
$to_L \leq from_R \implies$ `TRUE`
$from_L > to_R \implies$ `FALSE`
otherwise $\implies$ `UNKNOWN`

- Equal ($T_L = T_R$): if the two ranges do not overlap or if both operands contain only one possible value and they overlap, then we can be certain of the propagating path. Otherwise, the path is unknown.
  $from_L = to_L \wedge from_R = to_R \wedge from_L = from_R \implies$ `TRUE`
  $from_L > to_R \vee to_L < from_R \implies$ `FALSE`
  otherwise $\implies$ `UNKNOWN`

- Not Equal ($T_L \neq T_R$): if the two ranges do not overlap or if both operands contain only one possible value and they overlap, then we can be certain of the propagating path. Otherwise, the path is unknown.
  $from_L > to_R \vee to_L < from_R \implies$ `TRUE`
  $from_L = to_L \wedge from_R = to_R \wedge from_L = from_R \implies$ `FALSE`
  otherwise $\implies$ `UNKNOWN`

In total, 192 lines of code are added to the Constant Value Type System to support dead branch analysis in the type system.

6.2.3 Experiments

This case study is performed on a cloud instance with a four-core CPU and 16GB RAM, running 64-bit Ubuntu 20.04. The impact of the changes on the Narrowing and Widening Checker is evaluated as it uses the constant value type system. We analyze its effect on precision and performance using one test file and eight real-world projects. The test file contains 9 false positives. Table 6.1 shows the total numbers of errors issued and the evaluation time by the Constant Value type system with the dead branch analysis and without the dead branch analysis.

The total increase in evaluation time for 10 projects with the addition of the dead branch analysis is 5.63 seconds, almost negligible. The errors issued are examined manually. All 9 false positives from the test file are resolved. None of the errors issued from the real-world projects are false positives due to missing dead branch analysis. No impact on the errors issued for those projects are expected. The impact on errors may seem negligible with dead-branch analysis in this experiments as most projects may not contain dead branches,

	Without		With	
Project	Errors	Time (s)	Errors	Time (s)
common-bcel	2	70.492	2	71.529
common-bsf	21	26.903	21	26.876
commons-compress	19	52.639	19	53.104
commons-imaging	19	78.943	19	78.462
commons-jxpath	1	37.012	1	39.108
commons-logging	11	14.403	11	16.633
commons-net	18	40.814	18	39.863
common-ognl	1	47.894	1	49.035
dead-branch.java	9	2.854	0	2.974
Total	101	371.954	92	377.584

Table 6.1: Column `Without` shows the number of errors issued and evaluation time by the Narrowing and Widening Checker on the Apache Commons projects without dead branch analysis. Column `With` is when dead branch analysis is included. The times are in seconds.

but this analysis is crucial as it is part of the overarching solution in resolving false-positives issued due to loop imprecision.

6.3 Path-Sensitive Analysis

The path-sensitive analysis tracks data-flow depending on the path taken. The Dataflow Framework uses a path-insensitive analysis. The transfer functions merge the incoming stores together. The merged store cannot distinguish the mapped values from their respective paths. Fig. 6.3 is an example of a false positive produced by the Constant Value type system as the Dataflow Framework is path-insensitive. Fig. 6.4 shows the control-flow graph generated by the Dataflow Framework for Fig. 6.3. When the two branches merge, a new property is computed by finding the least upper bound of the two stores from the two paths, which results in $\{x \in \{0, 1\}, y \in \top, z \in \{0, 1\}\}$. This store is used as the transfer input for node $(x - z)$ and produces a store evaluating this expression to be `@IntVal(-1,0,1)`. From the result, the correctness of the program in terms of a division by zero check cannot be established for expression $10/(x - z)$ as $(x - z)$ are evaluated to zero.

The Dataflow Framework does not allow distinctions between different stores since the transfer input and result may only contain either a single store or a pair of stores. When

```
int divide_by_zero(int y) {
  int x = 0;
  int z = 0;
  if (y == 1) {
    x = 1;
  } else {
    z = 1;
  }
  return 10/(x-z);
}
```

```
warning: [divide.by.zero] Possible division by zero.
        return 10 / (x - z);
                  ^
  denominator: @IntVal({-1, 0, 1})
```

Figure 6.3: An example of false positive error due to path-insensitive analysis.

the branching merges, the store merges by computing the least upper bound of the two stores. To account for path sensitivity, the transfer result and transfer input should be able to distinguish the stores from different paths. To add path-sensitivity to the Dataflow Framework, stores should not merge. The transfer input and the transfer result should contain a set of stores or a set of then and else stores.

Going back to Fig. 6.3, when the branch merges, the stores from the two branches are collected into a set. The collected stores are $\{x \in \{0\}, y \in \top, z \in \{1\}\}$ and $\{x \in \{1\}, y \in \top, z \in \{0\}\}$. The expression $(x - z)$ takes in the two stores collected as the transfer input and produces two stores evaluating $(x - z)$ to `@IntVal(-1)` and `@IntVal(1)` as transfer result. The correctness of the program can now be correctly established for expression $10/(x - z)$ as $(x - z)$ does not evaluates to zero.

6.4 Loop Imprecision

The Dataflow Framework cannot statically predict the exact number of times a loop in the program will be executed. When encountering a loop, the analysis will iterate to a fix-point, by iteratively applying a set of transfer functions to the nodes in the CFG.

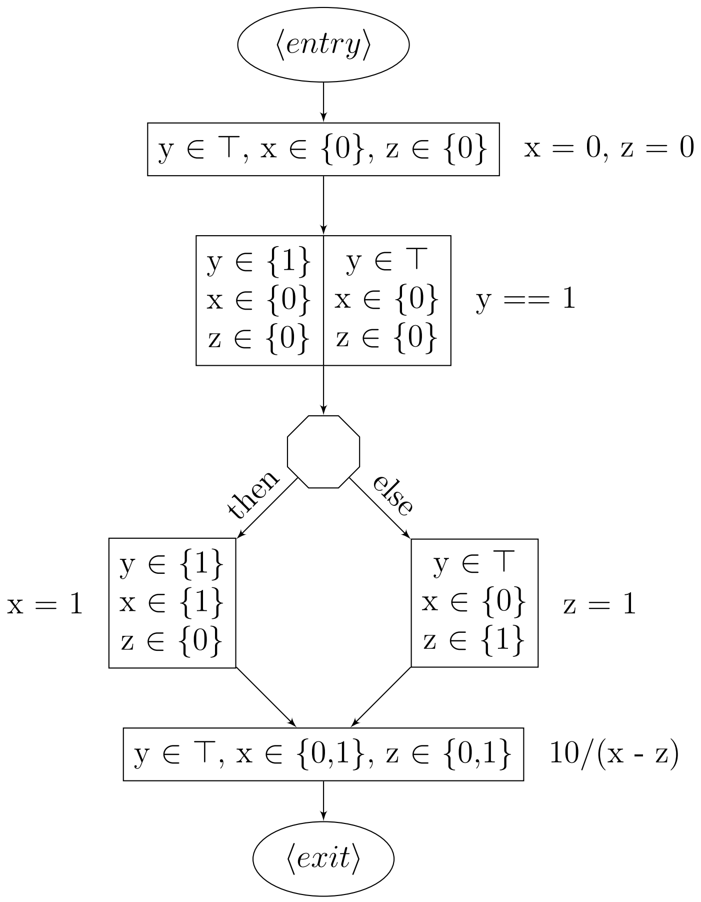


Figure 6.4: Control-flow graph of the example in Fig. 6.3 containing the transfer result in each block produced after evaluating the expressions.

The fix-point is determined by checking the semantic equivalence of the previous and current stores. Fig. 6.5 is an example of a false positive produced by the Constant Value type system when encountering loops. Fig. 6.6 is the control-flow graph produced by the Dataflow Framework for that program. In the first iteration (Fig. 6.6a, the boolean expression `x < max` produces a *then* store that contains $\{x \in \{0\}, sum \in \{0\}, max \in \{2\}\}$, and an *else* store that contains $\{x \in \bot, sum \in \{0\}, max \in \{2\}\}$. The *then* store is then propagated through the *then* branch and used as the transfer input to the set of transfer functions to the nodes in the loop body. The transfer result produces a store containing $\{x \in \{1\}, sum \in \{1\}, max \in \{2\}\}$. The transfer results from the loop body then propagates back to the condition expression. The two incoming stores merge and the boolean expression produces a *then* store containing $\{x \in \{0,1\}, sum \in \{0,1\}, max \in \{2\}\}$. Since this new store is not semantically equivalent to the previous store, the transfer functions are again applied and analyzed. The second iteration produces a store containing $\{x \in \{0,1,2\}, sum \in \{0,1,2\}, max \in \{2\}\}$ after analyzing the *then* branch the second time and merging with the incoming store. The boolean expression produces a *then* store containing $\{x \in \{0,1\}, sum \in \{0,1,2\}, max \in \{2\}\}$. Note that even though the value of x in this store is the same as the previous store, a fix-point is not reached because the value of *sum* changed. Therefore, the set of transfer functions are applied again until a fix-point for *sum* is reached as well. The final control-flow graph with all stores reaching a fix-point is illustrated in Fig. 6.6b.

To prevent infinite iterations for computing the fix-point, widening occurs after a certain number of iterations. For the Constant Value type system, the max number of iterations before widening is 10. The range of the value is first verified against the byte range. If it is within the byte range, then it is widened to full byte range $\top_{byte}$. If the value is within the short range then it is widened to the full short range $\top_{short}$. If the value is within the int range, then it is widened to the full int range $\top_{int}$. Otherwise, to the full long range $\top_{long}$. After 10 iterations, *sum* is widened to `IntRange(-128,127)`, then to `IntRange(-32768,32767)`, and finally to `IntRange(from=-2147483648, to=2147483647)` or $\top_{int}$ since sum is type `int`. Fix-point is reached after 13 iterations.

Not only does this process give an imprecise value for variable *sum* by evaluating it to $\top$, it also impacts performance as more than necessary iterations are analyzed in order to reach the fix-point. For this program, we want $sum \in \{2\}$ at the return statement and only two iterations of the loop body are sufficient to determine the correct value of the program.

Classical path-sensitive analysis usually stores the path of the program and evaluates the feasibility of the paths. We do not store the path of the program in this design. We instead store the values from different paths in separate stores. Storing the path of the

```
@IntVal(2) int whileloop() {
    int max = 2;
    int sum = 0;
    int x = 0;
    while (x < max) {
        x++; sum++;
    }
    return sum;
}
```

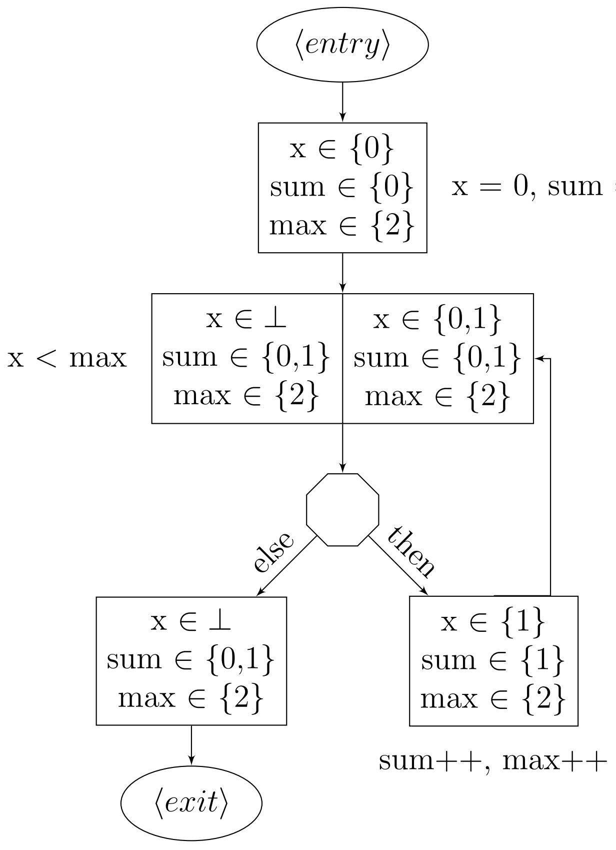
```
Error: [return.type.incompatible] incompatible types in return.
  type of expression: @IntRange(from=-2147483648, to=2147483647)
  method return type: @IntVal(2L) int
```

Figure 6.5: An example of false positive error in a while loop. The error is produced by the Narrowing and Widening Checker.

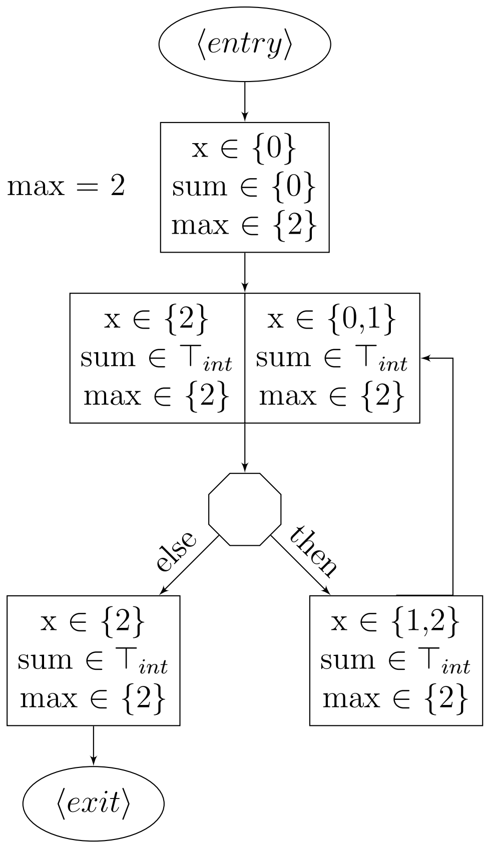
program can be beneficial in understanding the correlations among the variables and path feasibility. In this design, we rely on the abstract interpretation of the type system and the dead branch analysis to determine whether a path is reachable. The state of the variables in the boolean expression may change and storing the path means we are also required to store the state changes of the variables. This will increase memory and evaluation time. This design may not be as precise as the classical path-sensitive analysis since we do not have information on correlations among the variables, only the current state the variables are in, but this design is sufficient in resolving common false positives issued due to imprecision in Dataflow Analysis, a good balance between precision and performance.

This imprecision is fixed by applying the dead branch analysis described in Sec. 6.2 and the path-sensitive analysis described in Sec. 6.3. These two additional improvements ensure precision for loop iterations that do not exceed the maximum number of iterations before widening. The final control-flow graph produced by the program from Fig. 6.5 is illustrated in Fig. 6.7:

1. The transfer input at expression `x < max` starts with a single store containing $\{x \in \{0\}, sum \in \{0\}, max \in \{2\}\}$. The transfer input is analyzed by the Condition Evaluator which determines that this expression is always true since `0 < 2`. Two stores are produced at the transfer result of expression `x < max` with $\{x \in \{0\}, sum \in$



(a) First Iteration CFG



(b) Final CFG

Figure 6.6: CFG produced by the Dataflow Framework for the example in Fig. 6.5. (a) is the graph after the first iteration of the loop stopping before the conditional block. (b) is the final CFG after the values in the store have reached a fixed point.

$\{0\}, max \in \{2\}\}$ in the `then` store and $\{x \in \bot, sum \in \{0\}, max \in \{2\}\}$ in the `else` store marked as inactive.

2. The transfer function in the `then` branch is applied to store $\{x \in \{0\}, sum \in \{0\}, max \in \{2\}\}$ which produces a transfer result storing $\{x \in \{1\}, sum \in \{1\}, max \in \{2\}\}$. This results flows back to the boolean expression. The transfer input at expression `x < max` now contains two stores.

3. Once again the stores from the transfer input at the boolean expression are evaluated by the Condition Evaluator. The Conditional Evaluator determines that both of these stores will always evaluate the boolean expression to true and mark the two else stores as inactive. The two `then` stores become the transfer input for the expressions in the `then` branch which produces a transfer result with two stores containing $\{x \in \{1\}, sum \in \{1\}, max \in \{2\}\}$ and $\{x \in \{2\}, sum \in \{2\}, max \in \{2\}\}$.

4. The transfer input at the boolean expression now contains 3 regular stores. Two of the stores evaluates the expression to be always true and one of the stores will evaluate the expression to be always false. The stores are marked as 'active' and 'inactive' accordingly. Since the `then` store already contains active stores, the inactive then store produced by the expression will not be added to the sets of active `then` stores in the results of the boolean expression. Therefore, the then store still only contains 2 stores. Since the else store now contains active stores, all the inactive stores in the else store are removed and the else store now contains only one store.

5. As the set of then stores did not change, a fix-point has been reached for the `then` branch so no more iterations are required. The else store containing $\{x \in \{2\}, sum \in \{2\}, max \in \{2\}\}$ is used as the transfer input for further analysis.

Only two iterations are analyzed to reach the fix-point and the variable *sum* is also evaluated to 2 instead of $\top_{int}$.

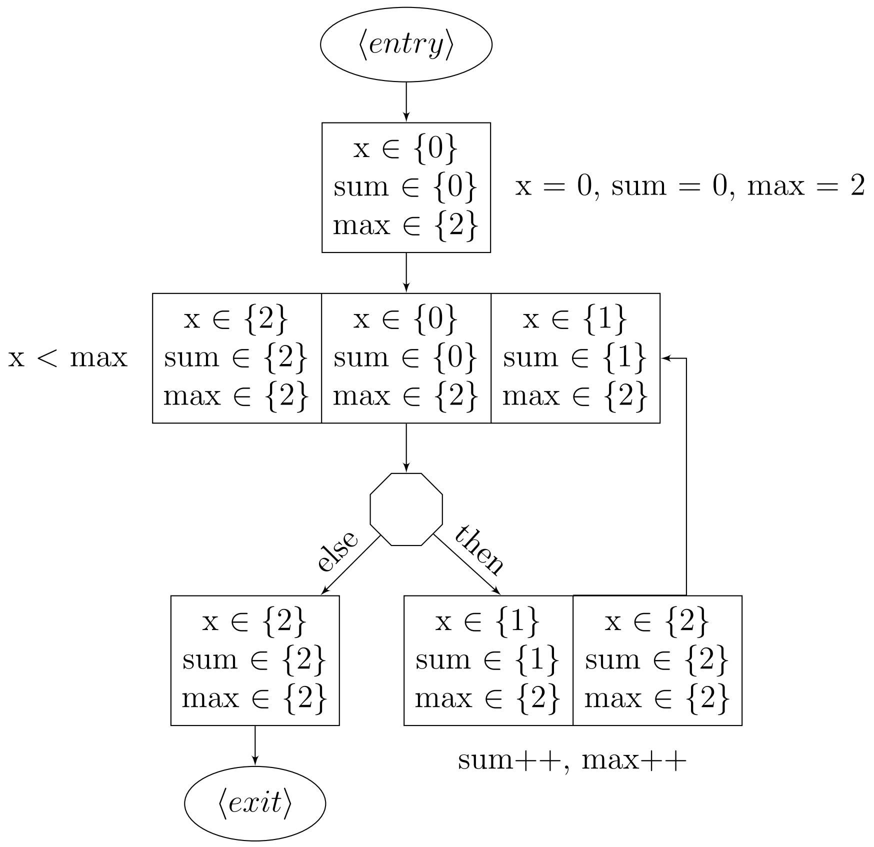


Figure 6.7: Final Control-flow Diagram output generated by the modified Dataflow Framework for program 6.5

In the above case, efficiency was improved and memory is the only trade-off for precision. However, if variable *max* is an unknown range, then the number of iterations and the result of x and *sum* would be the same as the original design at the cost of increased memory. The memory increase is based on the defined number of iterations before widening. If the number of iterations is 10 as defined by the Constant Value type system, then the `then` branch will contain 13 stores, one for each iteration. The `else` branch will still only contain a single store as only one store satisfies the `else` condition.

6.5 Related Work

Many software formal verification languages, such as Dafny [35], a programming language with built-in specification constructs, contain logic for handling loop invariants. Loop invariants must be defined to formally verify the program. Dafny can infer simple loop invariants where possible and reduce some burdens on the developer.

SeaHorn [24], a verification framework for C, does not use loop invariants. Instead, it detects the invariants and tries to determine if they are inductive invariants [42]. If they are then they can be used to prove program properties.

Bounded model checking [6] tools like CBMC [34], unwind the loops during program verification. The result of the verification, whether the program evaluates to SAT or UNSAT, is dependent on the number of times the loop is unrolled. The proposed approach in improving loop precision described in Sec. 6.4 is similar to the concept of loop unwinding in bounded model checking. The precision of the estimated range of variables inside the loop is dependent on the number of times the loop is evaluated before widening.

Chapter 7

Conclusions and Future Work

Value range analysis is important in many software domains. In this thesis, we presented the Narrowing and Widening Checker, the Value Inference for Integral Values, *PUnits*, and improvements to the algorithm of the Dataflow Framework. These pluggable type systems and improvements help programmers and other program analysis tools to better understand their programs.

The work in this domain is not yet finished.

A more advanced post-condition type qualifier can be designed and implemented to support a more expressive form of relationship between the method inputs and output. A more expression form of `@IntRange` can also be implemented to support multiple interval ranges rather than just one interval range. Instead of using `@PolyValue`, an analysis can be implemented for statically analyzing the results of each function in the Math Library, given the input parameters, and reduce some of the false positives produced from evaluating the Math functions. The Constant Value type system can evaluate String variables, and that can help statically analyze the results of a `Integer.parseInt` function. The Constant Value type system can also statically execute methods annotated with `@StaticallyExecutable`. Future work can investigate how this qualifier can help resolve some of the issues. These improvements not only improve the false positive rate of the Narrowing and Widening Checker, but also increase the expressiveness and capabilities for future type systems created using the Checker Framework.

PUnits is the first units type system to adapt the concept of receiver-dependent types. *PUnits* currently does not infer whether a method parameter or return type should be receiver dependent over units. Finding an efficient encoding is left as future work. Implementing prefix p using floating-point arithmetic with safe comparisons instead of the current base-10 prefix would allow *PUnits* to support Imperial units more easily. *PUnits* performance in annotation mode also warrants further investigation. We remain optimistic in this regard: SMT solver performance has been steadily improving, we can explore different constraint encodings, and we can employ alternative solvers.

We plan to extend support for a more expressive form of polymorphism for the Narrowing and Widening type system and *PUnits*. This will enable support for methods such as `myDivide(x, y)`, which returns a unit or interval computed as a function of the method arguments. Relationships between variables could be explicitly declared via declaration annotations or automatically inferred from the typing constraints in the method body.

The Value Inference also does not contain a post-condition qualifier. Finding a way to build the constraints for defining pre- and post-condition of method invocation with an efficient encoding is left as future work. The Value Inference currently only infers integral ranges, but can be extended to support type inference of floating-point values as well, with type qualifiers similar to `@DoubleVal` in the Constant Value type system.

Improvements to the loop algorithm in the Dataflow Framework will greatly benefit the type systems introduced in this thesis and provides more precise results during refinement. For future work, other than the solution discussed in Sec. 6.4, other solutions in resolving the loop imprecision can be explored. For example, adding support for loop invariants, or adding additional type qualifiers specifically for handling loops, and behave similarly to the post-condition type qualifier. Future work can explore how the information provided by the Constant Value type system can be used in automatically determining the loop invariant.

All the presented type systems build on techniques from type qualifier systems and constraint-based type inference. Our implementation and evaluation of these type systems show that these techniques are necessary and are effective in ensuring correctness of real-world programs.

References

[1] Kaw Autar, E Kalu Egwu, and Nguyen Duc. Numerical methods with applications: Abridged, chapter 04.06 gaussian elimination. `http://mathforcollege.com/nm/mws/gen/04sle/mws_gen_sle_txt_gaussian.pdf`, 2011.

[2] 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.

[3] Oscar Bennich-Björkman and Steve McKeever. The next 700 unit of measurement checkers. In *Software Language Engineering, (SLE)*, pages 121–132, 2018.

[4] Tom Benson. Reynolds number. `https://www.grc.nasa.gov/WWW/BGH/reynolds.html`, 2014.

[5] Dirk Beyer, Sumit Gulwani, and David A Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, pages 493–540. Springer, 2018.

[6] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. 2003.

[7] Mars Climate Orbiter Mishap Investigation Board. *Mars climate orbiter mishap investigation board: Phase I report*. Jet Propulsion Laboratory, 1999.

[8] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.

[9] Percy Williams Bridgman. *Dimensional analysis*. Yale University Press, 1922.

[10] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 123–124. IEEE, 2009.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[12] Travis Carlson and Eric Van Wyk. Type qualifiers as composable language extensions. In *Generative Programming: Concepts & Experiences (GPCE)*, pages 91–103. ACM, 2017.

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

[14] Zhuo Chen. Pluggable properties for program understanding: Ontic type checking and inference. Master's thesis, 2018. Available at `https://uwspace.uwaterloo.ca/handle/10012/13181`.

[15] Jean-Marie Dautelle and Werner Keil. JSR 275: Units specification API. `https://jcp.org/en/jsr/detail?id=275`, 2010.

[16] Jean-Marie Dautelle, Werner Keil, and Leonardo Lima. JSR 363: Units of measurement API. `https://jcp.org/en/jsr/detail?id=363`, 2016.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

[18] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*, pages 681–690. ACM, 2011.

[19] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 28–53. Springer, 2007.

[20] 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. Springer, 2011.

[21] Michael D Ernst, Alex Buckley, Werner Dietl, Doug Lea, Srikanth Sankaran, and Oracle. JSR 308: Annotations on Java types. `https://jcp.org/en/jsr/detail?id=308`, 2012.

[22] Narain H. Gehani. Ada's derived types and units of measure. *Software: Practice and Experience*, 15(6):555–569, 1985.

[23] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification.* Addison-Wesley Professional, 2000.

[24] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *Computer Aided Verification (CAV)*, pages 343–361. Springer, 2015.

[25] Sudheendra Hangal and Monica S Lam. Automatic dimension inference and checking for object-oriented programs. In *International Conference on Software Engineering (ICSE)*, pages 155–165. IEEE Computer Society, 2009.

[26] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[27] Mark Hills, Feng Chen, and Grigore Roşu. A rewriting logic approach to static checking of units of measurement in c. *Electronic Notes in Theoretical Computer Science*, 290:51–67, 2012.

[28] Nahid Juma, Werner Dietl, and Mahesh Tripunitara. A computational complexity analysis of tunable type inference for generic universe types. *Theoretical Computer Science*, 814:189–209, 2020.

[29] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to c: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98, 2017.

[30] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D Ernst. Lightweight verification of array indexing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 3–14, 2018.

[31] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*, pages 268–305. Springer, 2009.

[32] Andrew J Kennedy. Relational parametricity and units of measure. In *Principles of Programming Languages (POPL)*, pages 442–455. ACM, 1997.

[33] Sebastian Krings and Michael Leuschel. Inferring physical units in B models. In *Software Engineering and Formal Methods (SEFM)*, pages 137–151. Springer, 2013.

[34] Daniel Kroening and Michael Tautschnig. CBMC-C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[35] K Rustan M Leino. Developing verified programs with dafny. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1488–1490. IEEE, 2013.

[36] Jianchu Li. A general pluggable type inference framework and its use for dataflow analysis. Master's thesis, 2017. Available at https://uwspace.uwaterloo.ca/handle/10012/11771.

[37] Jens Melzer. Autoboxing performance. https://effective-java.com/2015/01/autoboxing-performance, 2015.

[38] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[39] Dominic Orchard, Andrew Rice, and Oleg Oshmyan. Evolving Fortran types with inferred units-of-measure. *Journal of Computational Science*, 9:156–162, 2015.

[40] Checker Framework Organization. A dataflow framework for java. 2020.

[41] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212. ACM, 2008.

[42] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[43] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26, Beijing, China, June 2012.

[44] Mier Ta. Context sensitive typechecking and inference: Ownership and immutability. Master's thesis, 2018. Available at https://uwspace.uwaterloo.ca/handle/10012/13185.

[45] Scott Wlaschin. Units of measure - type safety for numerics. https://fsharpforfunandprofit.com/posts/units-of-measure, 2012.

[46] Tongtong Xiang, Jeff Yucong Luo, and Werner Dietl. Precise inference of expressive units-of-measurement types (to appear). *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.