Karlsruhe Institute of Technology

KASTEL

More Powerful Constant Value Checking with SMT Solving

Bachelor's Thesis by

Jonas Mittnacht

At the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Examiner: Prof. Dr. Bernhard Beckert
Advisor: Florian Lanzinger, M. Sc.

03 November 2025 – 03 March 2026

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed sources and auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT) in their currently valid version.

Karlsruhe, 03 March 2026

. . . . . .

(Jonas Mittnacht)

Abstract

Pluggable type systems extend the basic type system of a programming language by introducing additional type hierarchies to specify more complex properties and dependencies between variables. For Java, the *Checker Framework* provides various pluggable type systems corresponding with so-called *Checkers* that verify the correct use of their types. One such Checker is the Constant Value Checker, with which a user can specify restrictions on the values a primitive integer or boolean variable can hold using a set or range of constant values. However, the current implementation of the Value Checker makes use of a limited set of manual typing rules that, in practice, often fail to verify well-typedness of more complex code. The aim of this thesis is to address this limitation using *SMT (Satisfiability Modulo Theories) solvers* by creating corresponding first-order logic formulas for expressions whose type checks fail with the current implementation and determining their well-typedness based on the satisfiability of these formulas. Furthermore, we introduce new *dependent types* for specifying allowed values with expressions that can depend on other variables in the program. We formally prove the correctness of this extension on the basis of a simple theoretical programming language and assess whether such an SMT value checking subsystem is able to reduce the amount of false negative type checking results without introducing excessive overhead using a set of sample programs.

Contents

1 Introduction

1.1 Motivation

In an effort to reduce the number of program errors only discovered at runtime, many modern programming languages rely on static type checking as a way of ensuring that the values passed around in the program match the expected format such that operations on them are well-defined and do not cause unintended consequences. However, while such static type systems make guarantees about the basic structure of a value (i.e. the size and interpretation of a primitive type or the fields and methods of a class type behind a reference), they often lack the ability to specify more precise semantics and relations between variables in the program. This also applies to the type system in the Java programming language - for example, a user can specify a class type `Range` that has two `long` fields `from` and `to`, but cannot specify that `from` should always be less or equal to the value of `to`.

To verify and strengthen guarantees about a Java program ahead of execution, the Checker Framework [16] project provides a set of so-called *pluggable type systems*. Pluggable type systems are type systems defined on top of a language's base types and allow the specification of additional properties. In the Checker Framework, specifiable properties include whether a reference type may or may not be `null` [15], whether a variable's value is known to be a key for a `Map` [14] or whether certain methods have been called on an object before a specific operation [11]. The Checker Framework then uses so-called *Checkers* to attempt to verify the correct usage of these types and issues warnings or errors for failed type checks.

The foundation of this thesis lies within the framework's Constant Value Checker [12]. It provides a set of annotations for denoting that the value of a variable of a primitive or `String` type (or the length of an array) is restricted a set of constant values specified in the annotation. However, the type checking capabilities of the underlying Checker implemented in the framework are rather limited in power, particularly because it does not track correlations (e.g. equality) between variables. In expressions with strongly correlated variables, the Checker therefore often evaluates the resulting value to be within a much broader range than required. An example of this can be seen in Figure 1.1. The annotation `@IntVal({`x_1`, ..., `x_n`})` specifies that a value x of this type satisfies $x \in \{ x_1, ..., x_n \}$, while `@IntRange(from=`$x_l$`, to=`$x_u$`)` specifies that a value x of this type satisfies $x \in [x_l, x_u]$. With the values of the parameter `x` being restricted to the range $[-2, 2]$, the expression `(x + 1) * (x - 1)` can only result in values between -1 and 3. Despite this, the type check at the return statement fails with the Value Checker, because it anticipates values as low as -9.

```
@IntRange(from=-1, to=3) int polynomial(@IntRange(from=-2, to=2) int x) {
    return (x + 1) * (x - 1);
    // type error: Value Checker expects type:
    // @IntVal({-9, -6, -4, -3, -2, -1, 0, 1, 2, 3})
}
```

Figure 1.1: A simple program annotated with Value Checker types that the Value Checker issues a type error for, despite the specified types being semantically correct.

Nevertheless, fully tracking the effects each operation in an expression has on its potential values depending on the state of and relations between the referenced variables would essentially require implementing a symbolic execution system that is compatible with all code relevant to the supported language types to prevent losing information and thereby introducing inaccuracies. Thus, augmenting the manual typing and inference rules in order to address this issue would be a rather complex task.

1.2 Foundations

The aim of this thesis will be to alleviate the aforementioned limitations specifically for type checks on expressions of the Java integer (`byte`, `short`, `int`, `long` and `char`) and `boolean` types using SMT (*Satisfiability Modulo Theories*) solving instead. For this purpose, we will reduce the correctness of a given value type on an expression e to the satisfiability of a corresponding first-order logic formula which we will call its *type check formula*. Such a formula will incorporate information from the assignments and types of variables in the given program, contain a translation of e into a corresponding term in first-order logic, and express the conditions under which e would be ill-typed. As an example, considering the method in Figure 1.1, we could create a type check formula of the form

$$x \geq -2 \wedge x \leq 2 \wedge e \doteq (x + 1) * (x - 1) \wedge (e < -1 \vee e > 3).$$

This formula incorporates the fact that our parameter `x` is restricted to the values between -2 and 2 according to its value type as well as the translation of the expression at the `return` statement, and further includes the condition that e does not satisfy the specified return value type `@IntRange(from=-1, to=3)`. Therefore, if we can find values of e and x that satisfy the formula (under the condition that the operations in our formula are defined according to their Java semantics), we know that there are cases where this return type is violated. Conversely, if this formula is unsatisfiable, we know that no such values can exist, and we can thus conclude that the return type for the expression is correct. While we will not always be able to create formulas for which satisfiability implies an ill-typed expression, we will design our formulas in a way such that unsatisfiability *does* imply well-typedness.

To decide the satisfiability of such formulas, we will be using *SMT solvers*. SMT solvers are programs that receive first-order logic formulas that make use of boolean predicates over terms with function symbols as inputs and attempt to decide whether these formulas are

satisfiable within a specific *theory*, that is, whether there is an *interpretation* of the boolean predicates, function symbols and free variables in the formula that fulfils the restrictions of the theory and satisfies the formula. In the remainder of this thesis, we reduce an interpretation of a formula to its mapping β of free variables v to a value $\beta(v)$ within its domain, as all other predicates and functions in our formulas will already be defined by the relevant theories. Furthermore, while some SMT solvers can deal with quantifiers in their input formulas, for the purposes of this thesis, we will only regard *quantifier-free* formulas (formulas without $\forall$ and $\exists$). Common theories for quantifier-free formulas supported by SMT solvers are:

- *Uninterpreted functions*: a theory that does not make restrictions on the relevant interpretations (and which is therefore often also called the *empty theory*), i.e. all interpretations are considered for the satisfiability of a formula.
- *Integer arithmetic*: a theory that specifies the domain of mathematical integers and defines various arithmetic functions such as addition, subtraction, multiplication and division. Variants of this theory without non-linear arithmetic operations also exist.
- *Bitvectors*: a theory that defines the domain of finite-length bit vectors as well as their arithmetic operations (including overflow behaviour) and bitwise operations, the semantics of which closely match the ones for the integer types commonly used in programming languages.

A drawback of SMT solving is that not all theories are *decidable*, i.e. for some theories, there is no algorithm that can always decide the satisfiability of any given formula adhering to that theory within finite time. Details about the theories used for the implementation in this thesis and their decidability will be discussed in Section 4.4. Furthermore, since all instances of the NP-complete *SAT* (satisfiability) problem [3] for propositional logic are also instances for the satisfiability problem of quantifier-free first-order logic, we know that even for decidable cases, the latter is often still NP-hard. In practice, it can therefore often be difficult to predict the runtime for solving a given problem instance.

In addition to (partially) deciding whether formulas are satisfiable or not, most SMT solvers are also capable of providing a *model*, i.e. a satisfying interpretation, for a given satisfiable formula. We will use this capability to provide further information on the specific values of expressions that violate a given type, instead of just returning a positive or negative result.

1.3 Research questions

This thesis aims to answer the following two research questions:

RQ1. Can we develop a type-checking system that makes use of first-order logic formulas and SMT solvers to perform sound value type checking on integer and boolean types?

RQ2. Does extending the Value Checker with such an SMT value type checking system meaningfully increase its power by reducing the amount of false negative type checking results (cases where the regular Value Checker reports an error despite the statement or expression being semantically well-typed) without also substantially increasing the overhead of the type checking process?

We will answer RQ1 by introducing a simple formal language in Chapter 2, on which we then formally define our constant value type system as well as its type checking process in Chapter 3. Afterwards, we implement the SMT type checking extension within the Checker Framework for Java, the details of which are discussed in Chapter 4. Finally, we dedicate Chapter 5 to an evaluation of the extension and discuss its impact as well as performance aspects regarding RQ2.

1.4 Related work

The field of pluggable type systems is well studied, with both general and more specialized approaches having been developed for the purpose of verifying additional properties about programs. As detailed in Section 1.1, we base our work on the existing Checker Framework project, which provides pluggable type systems for various additional properties that are not specified within the base type system of Java. Its Value Checker follows a rather standard procedure in the field of type checking, using predefined manual typing rules to propagate and transfer the types to expressions based on its operands. However, this approach reaches its limits rather quickly due to the lack of correlation tracking. With our nonstandard approach using SMT type checking, we intend to alleviate some of its problems by offloading the inference to SMT solvers, which are usually optimized for dealing with arithmetic and boolean operations and inferring facts from the former.

Lanzinger et al. [6] provide a more general approach to refining base type systems with the *Kukicha* method, which introduces the concept of *property types* as a generic way to specify new type hierarchies for properties directly using first-order logic constraint formulas. Verification of types is achieved in a two-step process: first, a set of rules defined manually for a given property type hierarchy is used to try and syntactically verify the correctness. If the syntactic checker fails to prove correctness, the Property Checker instead translates the unproven refinements into *JML (Java Modeling Language)* [17] specifications that can then be proven manually with the *KeY deductive verifier* [18]. While this approach drastically increases the expressiveness of the combined type system, it generally requires user input when verifying more complex property types as KeY is not fully automatic, thus making it less accessible to developers that are not familiar with formal verification and prover tools. Similar approaches to refining a language's base type system exist in other languages, such as with the LiquidHaskell [7] project for the Haskell language that also allows for the definition of refinement types with first-order logic, but makes use of SMT solving to perform type checking instead of manual verification tools.

Zuber [19], similar to the approach for this thesis, extends the Kukicha method by adding another step between syntactic type checking and deductive verification that uses an SMT solver to try to automatically verify some of the types not provable with the syntactic checker. Nevertheless, due to the potential complexity and generic nature of property types, the SMT translation was kept rather simple and does not recognize conditional contexts and correlations between variables (e.g. because of assignments). By reducing the scope to constant value types on primitive variables, the SMT layer to be developed as part of this thesis aims to provide more context to the SMT solver.

2 Language

Before realizing SMT-based constant value checking within the Checker Framework, we introduce a formal basis for the type system and the translation of programs to the first-order logic formulas used for type checking. For this purpose, we first define a simple programming language called *Field Assignment Language* (FAL) alongside its syntax and semantics. This allows us to abstract away from some of the complexity and edge-case behaviour present in real-world languages like Java and keeps the SMT translation concise.

The language described in this chapter is an adapted version of the Field Assignment Language defined by Lanzinger et al. [6]. It is designed to be a simple object-oriented language that allows for structured class types and method calls. For the purposes of the constant value type system, we extend the language to include the base types `int` and `bool` as well as a simple `if` conditional statement, while omitting the packing and uniqueness type systems as they are not relevant to our type system.

2.1 Syntax

We declare our set of *base types* $\mathfrak{B} \coloneqq \{ \texttt{int}, \texttt{bool} \} \cup \mathfrak{C}$ to contain the primitive integer `int` and boolean `bool` types as well as the set of *class types* $\mathfrak{C}$. For each class type $\beta \in \mathfrak{C}$, we have a set of *methods* $methods(\beta)$ and a set of *fields* $fields(\beta)$ that, together, define the structure of the class. Each method $m \in methods(\beta)$ has a *definition* $methdef(m) = m(\texttt{this}, \overline{x})(\overline{l})[s]$ that includes the receiver of the method `this`, a list of *parameters* $\overline{x}$, a list of *local variables* $\overline{l}$, and the *method body* $s = methbody(m)$, where the syntax of s is defined by the grammar in Figure 2.1. We will denote the fact that $s = methbody(m)$ contains a statement $\hat{s}$ as $\hat{s} \in s$, and additionally restrict method bodies such that the labels of atomic statements are unique. Given such a method body, we can define a relation $\prec$ on the labels of its atomic statements:

Definition 1 (Predecessors and successors)**.** *Let ℓ_1, ℓ_2 be two labels of a method body $s \coloneqq methbody(m)$. ℓ_1 is an* immediate predecessor *to ℓ_2 (denoted as $\ell_1 \prec \ell_2$) iff one of the following statements hold:*

- $\ell_1 : s_1; \ell_2 : s_2 \in methbody(m)$,
- $\ell_1 : \texttt{if}\ (e)\ [\ell_2 : s_2; \dots]\ \texttt{else}\ [\dots] \in methbody(m)$,
- $\ell_1 : \texttt{if}\ (e)\ [\dots]\ \texttt{else}\ [\ell_2 : s_2; \dots] \in methbody(m)$,

e	::=	$v \mid e.f$	expression
	|	$e \oplus e$	binary operation
	|	$\odot e$	unary operation
	|	c	constant
$\oplus$	::=	$+ \mid - \mid * \mid / \mid \%$	arithmetic operators
	|	$\ll \mid \gg \mid \ggg \mid \& \mid \mid \mid \hat{}$	bitwise operators
	|	$= \mid \neq \mid > \mid < \mid \geq \mid \leq$	comparison operators
	|	$\wedge \mid \vee$	boolean operators
$\odot$	::=	$\sim \mid \neg$	unary operators
v	::=	`this`	current object
	|	x	x parameter
	|	l	l local variable
c	::=	$n \mid$ `true` $\mid$ `false`	$n \in [-2^{31}, 2^{31} - 1]$
γ	::$\in$	$\mathfrak{C}$	class types $\mathfrak{C}$
s	::=	$\ell : s_{atom}$	atomic statement
	|	$s; s$	composition
ℓ	::$\in$	$\mathbb{N}$	label
s_{atom}	::=	$l = e$	local variable assignment
	|	$v.f = e$	field assignment
	|	$l =$ `new` γ	object creation
	|	$l = v.m(e_1, \ldots, e_n)$	method call
	|	`return`	return statement
	|	`if` (e) $[s]$ `else` $[s]$	conditional statement
	|	`nop`	no operation

Figure 2.1: Syntax of FAL method bodies

- $\hat{\ell} :$ `if` (e) $[\ldots; \ell_1 : s_1]$ `else` $[\ldots]; \ell_2 : s_2 \in methbody(m)$,
- $\hat{\ell} :$ `if` (e) $[\ldots]$ `else` $[\ldots; \ell_1 : s_1]; \ell_2 : s_2 \in methbody(m)$.

ℓ_1 is a predecessor *to ℓ_2 (denoted as $\ell_1 \prec^* \ell_2$) iff either $\ell_1 \prec \ell_2$ or $\ell_1 \prec^* \ell_3 \wedge \ell_3 \prec^* \ell_2$ for a third label ℓ_3. ($\prec^*$ is the* transitive closure *of $\prec$.)*

FAL distinguishes between *parameters* and *local variables*, the differences being that local variables are reassignable within the method body while parameters are not, and the values of parameters are passed to a method on a method call, while local variables are uninitialized at the start of a method. Furthermore, a method call itself does *not* constitute a valid expression (unlike in Java), and the returned value must always be assigned to a local variable for it to be reusable. This restriction ensures that expressions are always side-effect free, i.e. evaluating an expression does not change the program state. To return values from a method, the value must be assigned to a designated `retval` variable before returning with `return`.

2.2 Semantics

For base type declarations, we define a function $decl_B$ that maps each field $f \in fields(\beta), \beta \in \mathfrak{C}$ to a base type $decl_B(f) \in \mathfrak{B}$ and each method $m \in methods(\beta)$ with $methdef(m) = m(\texttt{this}, \overline{x})(\overline{l})[s]$ (where $\overline{x} = (x_1, \ldots, x_i)$ and $\overline{l} = (l_1, \ldots, l_j)$) to a function

$$decl_B(m) : \{ x_1, \ldots, x_i, l_1, \ldots, l_j \} \to \mathfrak{B}$$

that declares a base type for each parameter and local variable. Furthermore, we declare a *domain* $\mathfrak{D}(\tau)$ for each type $\tau \in \mathfrak{B}$, where:

- $\mathfrak{D}(\texttt{int}) = [-2^{31}, 2^{31} - 1]$, which is the range of values representable with two's complement 32-bit bitvectors and matches the value range of the Java `int` type.
- $\mathfrak{D}(\texttt{bool}) = \{ \texttt{true}, \texttt{false} \}$
- $\mathfrak{D}(\beta) = \mathbb{N}_0, \beta \in \mathfrak{C}$.

Similarly, we declare a function $decl_F$ that maps fields f as well as local variables v of methods m to a value $x \coloneqq decl_F(f)$ or $x \coloneqq decl_F(m)(v)$ with $x \in \{ \texttt{final}, \texttt{nonfinal} \}$ indicating whether the variable element is *final*. Unless specified otherwise, we assume variables and fields to be declared as `nonfinal`. If a variable or field is declared `final`, it can only be assigned a value if it has not yet been initialized. Based on this definition, we call an expression *final* iff for all subexpressions v that refer to a field or variable, v is declared as `final`.

Given these definitions, we can now introduce the notion of a *program state*:

Definition 2 (Program state)**.** *A program state σ, ℓ for a method m consists of a function σ mapping*

- *each parameter or local variable v to a value $\sigma(v) \in \mathfrak{D}(decl_B(m)(v)) \cup \{ \bot \}$ either within its declared type or the symbol $\bot$ signifying that v has not been initialized yet,*
- `this` *to the unique identifier $\sigma(\texttt{this}) \in \mathbb{N}_0$ of the receiver object of the method call,*

- *a special value heap to a partial function $\sigma(heap)$ mapping identifiers n of objects and each field f of the type of that object to a value $\sigma(heap)(n, f) \in \mathfrak{D}(decl_B(f)) \cup \{\perp\}$ ($\perp$, again, signifying that the value has not been initialized yet),*

and a label ℓ denoting the statement that is executed next.

Definition 3 (Initial program states and the end label)**.** *An initial program state $\sigma^m_{init}, \ell^m_{init}$ for a method m is a program state in which*

- *ℓ^m_{init} is the label of the first statement in m,*
- *$\sigma^m_{init}(x) \neq \perp$ for all parameters x, i.e. all parameters are* initialized*,*
- *$\sigma^m_{init}(l) = \perp$ for all local variables l, i.e. all local variables are* uninitialized*,*
- *$\sigma^m_{init}(heap)(n, f) \neq \perp$ for all $(n, f) \in dom(\sigma^m_{init}(heap))$, i.e. all fields on the heap are* initialized*.*

ℓ_{end} refers to a placeholder label that indicates that a method has finished executing, i.e. any state σ', ℓ_{end} is a final state of a method execution.

Therefore, the value associated with a class-typed variable is a reference identifier to an object stored on the heap, where the values of its fields are found, as is the case with Java. Evaluation of an expression e in a state σ, ℓ is defined by the function $eval_\sigma$:

$$
\begin{aligned}
eval_\sigma(v) &= \sigma(v) && \text{for } e \text{ local variable, parameter or } \texttt{this} \\
eval_\sigma(e.f) &= \sigma(heap)(eval_\sigma(e), f) && \text{for expression } e \text{ and field } f \\
eval_\sigma(e_1 \oplus e_2) &= eval_\sigma(e_1) \oplus_{\mathit{Java}} eval_\sigma(e_2) && \text{for binary operations} \\
eval_\sigma(\odot e) &= \odot_{\mathit{Java}} eval_\sigma(e) && \text{for unary operations} \\
eval_\sigma(c) &= c && \text{for constant } c
\end{aligned}
$$

where $\oplus_{\mathit{Java}}$ and $\odot_{\mathit{Java}}$ are the corresponding Java operators for $\oplus$ and $\odot$ respectively, with the additional property that any operation where an operand has the value $\perp$ results in the value $\perp$.

Using this definition of the evaluation of an expression, we define the *type* of an expression as follows:

Definition 4 (Type of an expression)**.** *An expression e is of base type τ_B in a state σ (denoted as $\sigma \vDash e : \tau_B$) iff its value is either uninitialized or within the domain of τ_B: $eval_\sigma(e) \in \mathfrak{D}(\tau_B) \cup \{\perp\}$.*

An expression e is of base type τ_B in a method m (denoted as $m \vDash e : \tau_B$) iff it is of base type τ_B in any *state σ resulting from a program execution $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$.*

In addition, we introduce the concept of a *sound* expression:

Definition 5 (Sound expressions)**.** *We consider an expression e* sound *at a label ℓ of a method m iff in any method execution $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$, all referenced variables and fields occurring in e are initialized, i.e. $eval_\sigma(e) \neq \bot$.*

The execution of an FAL method statement s from a program state σ, ℓ in a method m with $methdef(m) = m(\texttt{this}, \overline{x})(\bar{l})[\_]$ is defined by the semantic rules in Figure 2.2, where $\sigma, \ell, s \rightsquigarrow \sigma', \ell'$ signifies that the execution of statement s (atomic or composition) in state σ, ℓ leads to the post-execution state σ', ℓ'. We will often abbreviate $\sigma, \ell, s \rightsquigarrow \sigma', \ell'$ with $\sigma, \ell \rightsquigarrow \sigma', \ell'$ if the precise statement is not relevant in the current context. In this case, $\sigma, \ell \rightsquigarrow \sigma', \ell'$ means that there is a sequence of atomic statements $s \in methbody(m)$ of the form $s = \ell : s_1; s'$ (i.e. a sequence that occurs in the program and starts at ℓ) whose execution in σ, ℓ leads to the state σ', ℓ'. Furthermore, we will use the notation $f' = f[x \triangleright v]$ to mean $f'(x) = v$ and $f'(y) = f(y)$ for $y \neq x$. $create(\sigma, l, \beta)$ is a function that creates a new object of type β on the heap of σ and assigns its unique identifier to the variable l:

$$create(\sigma, l, \beta) = \sigma[l \triangleright n, heap \triangleright \mu]$$

such that

- $n \notin \{ n' : \exists f' : (n', f') \in dom(\sigma(heap)) \}$ (i.e. n is unique),
- $\mu(n, f) = \bot$ for all fields $f \in fields(\beta)$ (all fields are uninitialized),
- $\mu(n', f') = \sigma(heap)(n', f')$ for all $(n', f') \in dom(\sigma(heap)), n' \neq n$ (the heap has not changed in any other way).

$$\frac{eval_\sigma(e) \neq \bot \quad decl_F(m)(l) \neq final \vee eval_\sigma(l) = \bot \quad \sigma' = \sigma[l \triangleright eval_\sigma(e)] \quad eval_\sigma(e) \in \mathfrak{D}(decl_B(m)(l))}{\sigma, \ell, l = e \rightsquigarrow \sigma', \ell'}$$

$$\frac{eval_\sigma(e) \neq \bot \quad decl_F(f) \neq final \vee eval_\sigma(v.f) = \bot \quad \sigma(v) = r \quad \mu = \sigma(heap)[(r, f) \triangleright eval_\sigma(e)] \quad \sigma' = \sigma[heap \triangleright \mu] \quad eval_\sigma(e) \in \mathfrak{D}(decl_B(f))}{\sigma, \ell, v.f = e \rightsquigarrow \sigma', \ell'}$$

$$\frac{decl_F(m)(l) \neq final \vee eval_\sigma(l) = \bot \quad \sigma' = create(\sigma, l, \beta)}{\sigma, \ell, l = \texttt{new}\ \beta \rightsquigarrow \sigma', \ell'}$$

$$\frac{eval_\sigma(e) = \texttt{true} \quad \sigma, \ell, s_1 \rightsquigarrow \sigma', \ell'}{\sigma, \ell, \texttt{if}\ (e)\ [s_1]\ \texttt{else}\ [s_2] \rightsquigarrow \sigma', \ell'} \qquad \frac{eval_\sigma(e) = \texttt{false} \quad \sigma, \ell, s_2 \rightsquigarrow \sigma', \ell'}{\sigma, \ell, \texttt{if}\ (e)\ [s_1]\ \texttt{else}\ [s_2] \rightsquigarrow \sigma', \ell'}$$

$$\frac{\begin{array}{c} decl_F(m)(l) \neq final \vee eval_\sigma(l) = \bot \quad \forall (n, f) \in dom(\sigma(heap)) : \sigma(heap)(n, f) \neq \bot \\ methdef(m') = m'(\texttt{this}, x'_1, \ldots, x'_k)(\_)[s] \quad \forall e = e_1, \ldots, e_k : eval_\sigma(e) \neq \bot \\ \forall l = 1, \ldots, k : \sigma \vdash e_k : decl_B(m')(x_k) \quad decl_B(m')(\texttt{retval}) = decl_B(m)(l) \\ \sigma^{m'}_{init}[\texttt{this} \triangleright eval_\sigma(v), x'_1 \triangleright eval_\sigma(e_1), \ldots, x'_k \triangleright eval_\sigma(e_k), heap \triangleright \sigma(heap)], \ell^{m'}_{init}, s \rightsquigarrow \sigma^*, \ell_{end} \quad \sigma' = \sigma[l \triangleright \sigma^*(\texttt{retval}), heap \triangleright \sigma^*(heap)] \end{array}}{\sigma, \ell, l = v.m'(e_1, \ldots, e_k) \rightsquigarrow \sigma', \ell'}$$

$$\frac{\forall (n, f) \in dom(\sigma(heap)) : \sigma(heap)(n, f) \neq \bot \quad \sigma(\texttt{retval}) \neq \bot}{\sigma, \ell, \texttt{return} \rightsquigarrow \sigma, \ell_{end}} \qquad \frac{\sigma, \ell, s_1 \rightsquigarrow \sigma', \ell' \quad \sigma', \ell', s_2 \rightsquigarrow \sigma'', \ell''}{\sigma, \ell, s_1; s_2 \rightsquigarrow \sigma'', \ell''}$$

Figure 2.2: Runtime semantics of FAL.

3 Formalization

Having established our theoretical base language, we now formally define our constant value type system. Section 3.1 will introduce the constant value types itself. Afterwards, we define the process used to decide whether an assignment to a variable or field is sound w.r.t. its constant value type in Section 3.2. Lastly, Section 3.3 is dedicated to a proof of the soundness of the type checking process.

3.1 Constant value types

In our constant value type system, we differentiate between two kinds of types:

Non-dependent value types Non-dependent types declare the set of allowed values for a type using constants. The set of non-dependent value types $\mathscr{V}_{nondep}(\tau)$ for a base type τ is therefore defined as $\mathscr{V}_{nondep}(\tau) \coloneqq \{ \texttt{any} \} \cup \mathbb{P}_{finite}(\mathfrak{D}(\tau))$, where `any` refers to the default value type that does not restrict a variable's values, and $\mathbb{P}_{finite}(A)$ refers to the set of finite subsets of a set A. A non-dependent constant value type for a variable or field v of primitive base type τ, then, is satisfied iff in any possible program state achievable from any initial state $\sigma^m_{init}, \ell^m_{init}$, the variable is uninitialized (i.e. has the value $\bot$) or holds a value within the set of the specified constant value type.

Dependent value types Dependent value types, in contrast, declare the set or range of allowed values in relation to final expressions, meaning that the set of allowed values for those types depends on a state of the program rather than predetermined values. For these, we first define:

$$
\begin{aligned}
\mathcal{E} &:= \{\, e : e \text{ expression} \,\} && \text{expressions} \\
\mathcal{E}^*_{range} &:= \{\, [e_l, e_r] : e_l, e_r \in \mathcal{E} \,\} && \text{range expression types} \\
\mathcal{E}^*_{set} &:= \{\, A \in \mathbb{P}_{finite}(\mathcal{E}) \,\} && \text{set expression types} \\
\mathscr{V}_{dep}(\texttt{int}) &:= \mathcal{E}^*_{range} \cup \mathcal{E}^*_{set} && \text{valid dependent types for } \texttt{int} \\
\mathscr{V}_{dep}(\texttt{bool}) &:= \mathcal{E}^*_{set} && \text{valid dependent types for } \texttt{bool}
\end{aligned}
$$

Additional restrictions apply depending on where these types are declared: all references to parameters, local variables or fields in the expressions of a dependent value type must be applicable at all statements at which an expression of this type occurs. This means that types declared on a field cannot refer to any local variables or parameters of a method since

it exists in the context of various different methods, and the types of the `retval` return variable and the parameters of a method cannot refer to any local variables. Furthermore, the expressions used in dependent types must be final to avoid cases where the type of a variable or field might suddenly become invalid due to the value of another variable or field having changed, and they must be sound at any use of the corresponding variable, parameter or field. Lastly, the expressions of the dependent type must be of the base type of the underlying variable, parameter or field in all methods of the program.

A type $A \in \mathcal{E}^*_{set}$ is satisfied for a variable or field iff in any state σ, ℓ of a method execution $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell \rightsquigarrow \sigma_{end}, \ell_{end}$, the value of that variable or field is either uninitialized or within $\{ x : e \in A, x = eval_\sigma(e) \}$, i.e. one of the expressions in A evaluates to the value of the variable or field. A type $[e_l, e_r]$, meanwhile, is satisfied iff in any state of any method execution of the aforementioned form, for the value x of the variable or field, $x = \bot$ or $eval_\sigma(e_l) \leq x \leq eval_\sigma(e_r)$, i.e. the value is always either uninitialized or lies between the values of e_l and e_r.

Together, the set of constant value types for a primitive base type τ is $\mathscr{V}(\tau) \coloneqq \mathscr{V}_{nondep}(\tau) \cup \mathscr{V}_{dep}(\tau)$. We denote the declared constant value type of a local variable or parameter v in a method m as $decl_C(m)(v)$ and the constant value type of a field f as $decl_C(f)$, similarly to the $decl_B$ function for base type declarations.

3.2 Type checking process

To type-check an expression e at a label ℓ against a constant value type τ_C, we create a first-order logic formula of the form

$$res \doteq fol_\ell(e) \land context_\ell(e, \emptyset) \land typecontext_\ell(\tau_B, \tau_C, \emptyset) \land conds_\ell \land \neg constrain_\ell(res, \tau_B, \tau_C),$$

where

- $fol_\ell(e)$ refers to the translation of the expression e to an equivalent first-order logic term where variables and field accesses in e are represented using corresponding formula variables,
- $context_\ell(e, \emptyset)$ is a logical conjunction of constraints that limit the possible values for all variables and field accesses in e according to their types and previous assignments in the program,
- $typecontext_\ell(\tau_B, \tau_C, \emptyset)$, similarly to $context_\ell$, includes context for formula variables that occur in the expressions of the constant value type τ_C, if τ_C is a dependent type,
- $conds_\ell$ encodes the branching conditions that lead to the execution of the statement at label ℓ,
- $\neg constrain_\ell(res, \tau_B, \tau_C)$ is a formula that expresses that *res* does *not* conform to its constant value type τ_C.

```
   foo(this, x)(l)[
1 :    if (x > 10 ∨ x < −10) [
2 :        retval = 1
3 :        return
       ] else [
4 :        if (x ≥ 0) [
5 :            l = x
           ] else [
6 :            l = −x
           ]
7 :        retval = l + 1
8 :        return
       ]
   ]
```

Figure 3.1: A sample FAL program method.

For sound type checking, this formula needs to be designed in a way such that for any program state σ at a label ℓ in which $eval_\sigma(e)$ does not conform to τ_C, there must be a corresponding interpretation of the formula that makes it satisfiable. If all type check formulas fulfill this property, we can always conclude by contraposition that e is of type τ_C if the corresponding type check formula is not satisfiable.

3.2.1 Collecting branching conditions and reaching definitions

In order to minimize false negative type check results (cases in which an expression e at label ℓ satisfies type τ_C, but the corresponding type check formula is satisfiable), the type check formula needs to contain as much information about the program as possible. Consider the sample program in Figure 3.1. Assuming $decl_B(foo)(x) = decl_B(foo)(l) = decl_B(foo)(\texttt{retval}) = \texttt{int}$, $decl_C(foo)(x) = decl_C(foo)(l) = \texttt{any}$ and $decl_C(foo)(\texttt{retval}) = [1, 11]$, to sufficiently prove that the expression $l + 1$ at label 7 indeed matches the constant value type $[1, 11]$ of the left hand side `retval`, the formula needs to include the following facts:

1. $-10 \leq x \leq 10$ at label 7 ,
2. l at label 7 is equal to x if $x \geq 0$,
3. l at label 7 is equal to $-x$ if $x < 0$.

Fact 1 stems from what we call the *branching condition* of label 7: its statement is only executed if the branching condition $x > 10 \vee x < -10$ at label 1 was not fulfilled. We formally define the branching conditions as follows:

Definition 6 (Branching conditions)**.** *The* branching conditions *$branch(\ell)$ for a label ℓ of a method m with $s_m \coloneqq methbody(m)$ are a set of tuples of the form (e, ℓ_e), where e represents an FAL expression at label ℓ_e that evaluates to* `true` *in any method execution that executed the statement at label ℓ. They are defined recursively as follows:*

$$
\begin{aligned}
branch(\ell^m_{init}) &= \emptyset \\
branch(\ell) &= branch(\ell') && \ell' : s'; \ell : s \in s_m \\
branch(\ell) &= branch(\ell') \cup \{ (e, \ell') \} && \ell' : \texttt{if } (e) \texttt{ [} \ell : s; \dots \texttt{] else [} \dots \texttt{]} \in s_m \\
branch(\ell) &= branch(\ell') \cup \{ (\neg(e), \ell') \} && \ell' : \texttt{if } (e) \texttt{ [} \dots \texttt{] else [} \ell : s; \dots \texttt{]} \in s_m
\end{aligned}
$$

Facts 2 and 3, meanwhile, give information about which value l holds, depending on which branch was executed. We call these facts the *reaching definitions* of the variable l at label 7:

Definition 7 (Reaching definitions)**.** *A tuple (C, ℓ') is called a* reaching definition *for a local variable l at a label ℓ iff*

- *ℓ' is a label at which l is assigned a value, i.e. the statement at label ℓ' is either of the form $l = e$, $l = v.m(p_1, \dots, p_n)$ or $l =$* `new` *γ,*
- *no statement between ℓ' and ℓ is an assignment to l,*
- *$C = \{ (e_1, \ell_1), \dots, (e_k, \ell_k) \}$ is a set of tuples of expressions and labels that represent the conditions for this reaching definition where $C \cap branch(\ell) = \emptyset$,*
- *any given method execution $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma_1, \ell_1 \rightsquigarrow \dots \rightsquigarrow \sigma_k, \ell_k \rightsquigarrow \sigma, \ell$ with $eval_{\sigma_i}(e_i) =$* `true` *$(i = 1, \dots, k)$ has reached label ℓ' and therefore executed the assignment to l at label ℓ', and there is no label $\hat{\ell}$ with $\hat{\ell} \notin \{ \ell_1, \dots, \ell_k \} \cup branch(\ell)$ that has been reached within that method execution and is a conditional statement.*

We call

$$reaching(\ell, l) \coloneqq \{ (C, \ell') : (C, \ell') \text{ is a reaching definition for } l \text{ at } \ell \}$$

the (set of) reaching definitions for l at ℓ.

In the example from Figure 3.1, the reaching definitions of l at label 7 are

$$reaching(7, l) = \big\{ \big(\{ (x \geq 0, 4) \}, 5 \big), \big(\{ (\neg(x \geq 0), 4) \}, 6 \big) \big\}.$$

The reaching definitions for a variable at a label can be calculated using a set of inference rules that have to be applied until saturation, i.e. until no further facts can be inferred. These rules are defined in Figure 3.2 using the previously defined branching conditions and the predicate $assigns(\ell, l)$, which holds iff the statement at label ℓ assigns a value to the local variable l, i.e. iff the value of l is changed during the execution of the statement at label ℓ.

$$\frac{\ell \prec \ell' \qquad assigns(\ell, l)}{(\emptyset, \ell) \in reaching(\ell', l)}$$

$$\frac{\ell \prec \ell' \qquad \neg assigns(\ell, l) \qquad (C, \ell'') \in reaching(\ell, l)}{((C \cup branch(\ell)) \setminus branch(\ell'), \ell'') \in reaching(\ell', l)}$$

Figure 3.2: Rules for gathering the reaching definitions of a variable at a label. These rules are based on the reaching definition analysis defined by Pfenning and Platzer [10].

3.2.2 Creating type check formulas

Given the branching conditions and reaching definitions for all variables at all labels in the program method, we can define the structure of our type check formulas.

FAL expression translation We begin by defining the function fol_ℓ that recursively maps each FAL expression at label ℓ to a corresponding term[1] in first-order logic:

$$\begin{aligned}
fol_\ell(e_1 \oplus e_2) &= fol_\ell(e_1) \oplus_{fol} fol_\ell(e_2) && \text{for binary operations on expressions } e_1, e_2 \\
fol_\ell(\odot e) &= \odot_{fol} fol_\ell(e) && \text{for unary operations on expressions } e \\
fol_\ell(x) &= x_{fol} && \text{for parameters } x \\
fol_\ell(e.f) &= \begin{cases} (e.f)_{fol} & \text{if } e \text{ and } f \text{ are final} \\ (e.f)^{\ell}_{fol} & \text{else} \end{cases} && \text{for field accesses } e.f \text{ on expressions } e \\
fol_\ell(l) &= \begin{cases} l_{fol} & \text{if } l \text{ is final} \\ l^{reaching(\ell,l)}_{fol} & \text{else} \end{cases} && \text{for local variables } l \\
fol_\ell(\texttt{this}) &= \texttt{this}_{fol} \\
fol_\ell(n) &= n_{fol} && \text{for constant literals } n \\
fol_\ell(\texttt{true}) &= \texttt{true}_{fol} \\
fol_\ell(\texttt{false}) &= \texttt{false}_{fol}
\end{aligned}$$

$\oplus_{fol}$ and $\odot_{fol}$ refer to the corresponding operators for $\oplus$ and $\odot$ in the theories for first-order logic formulas, while x_{fol}, l_{fol}, $(e.f)_{fol}$ (and the variants with superscripts) refer to variables and $\texttt{this}_{fol}$, n_{fol}, $\texttt{true}_{fol}$ and $\texttt{false}_{fol}$ refer to corresponding constants in the formula's terms. Local variables, parameters and field accesses are translated into equivalent formula variables, where the base types `int` and `bool` use a corresponding integer respective boolean type in the formula's theory, and all other types (class types) are translated into "identifier" integer variables, the values of which will not be constrained any further. Care has to be taken when converting non-final field accesses and local variables into their formula

[1] The values of this function are always *terms* in the given theory, even for the translation of Java `boolean` expressions. To incorporate terms into a formula, we therefore always need to use them in a *predicate*, for example with the $\doteq$ predicate, which is true iff both operands' terms represent the same value (in a given interpretation).

equivalents: since we often include context from previous labels into our formula, we need to differentiate between the states these variables could be in at different points of the program. We therefore create a different formula variable for each reaching definition set of a non-final program variable, since in a given method execution, the reaching definitions uniquely identify its value. For non-final field accesses, we use different variables for each label at which the field is referenced, because we cannot easily identify when a given field could have changed due to the fact that different expressions can refer to the same object on the heap. While this handling of fields is very conservative as it assumes a field could change at any label, it eliminates the need for complex reference tracking and simplifies the type checking process. Furthermore, a fixed value of a field can always be referenced at different labels by first assigning it to a local variable.

In addition to the fol_ℓ function, we also define a function $vars_\ell$ that maps each FAL expression e at label ℓ to the subexpressions in e that are mapped to a new formula variable:

$$
\begin{aligned}
vars_\ell(e_1 \oplus e_2) &= vars_\ell(e_1) \cup vars_\ell(e_2) && \text{for binary operations on expressions } e_1, e_2 \\
vars_\ell(\odot e) &= vars_\ell(e) && \text{for unary operations on expressions } e \\
vars_\ell(x) &= \{\, (x, fol_\ell(x)) \,\} && \text{for parameters } x \\
vars_\ell(e.f) &= \{\, (e.f, fol_\ell(e.f)) \,\} && \text{for field accesses } e.f \text{ on expressions } e \\
vars_\ell(l) &= \{\, (l, fol_\ell(l)) \,\} && \text{for local variables } l \\
vars_\ell(\texttt{this}) &= \{\, (\texttt{this}, fol_\ell(\texttt{this})) \,\} \\
vars_\ell(n) &= vars_\ell(\texttt{true}) && \text{for constant literals } n \\
&= vars_\ell(\texttt{false}) \\
&= \emptyset
\end{aligned}
$$

The first entry in each tuple of the resulting set represents the FAL expression for which the formula variable in the second entry of the tuple is created.

Type adaptation When using the type of a field in the context of a method or when type-checking a method call with its parameters, we need to potentially adapt dependent types to refer to the correct expressions. For example, given a class type β with fields $f, g \in fields(\beta)$ and $decl_C(f) = \{\, \texttt{this}.g \,\} \in \mathcal{E}^*_{set}$, when we reference f in an expression $e.f$, we need to replace `this` in the dependent type with e in the current context. For this purpose, we introduce the syntax $\tau_C[v_1/e_1, \ldots, v_k/e_k]$, which, for each $i = 1, \ldots, k$, replaces all references to v_i (which is either `this` or a parameter) in dependent type expressions in τ_C with the corresponding expression e_i.

First-order logic formulas for type constraints Having defined the translation of FAL to first-order logic expressions, we now detail the formula $constrain_\ell(v, \tau_B, \tau_C)$ constraining a formula variable v to values within the specified constant value type τ_C over the base type τ_B:

$$constrain_\ell(v, \texttt{int}, \tau_C) = \begin{cases} \texttt{true} & \text{if } \tau_C = \texttt{any} \\ v \geq fol_\ell(e_l) \wedge v \leq fol_\ell(e_r) & \text{if } \tau_C = [e_l, e_r] \in \mathcal{E}^*_{range} \\ \bigvee_{e \in A} v \doteq fol_\ell(e) & \text{if } \tau_C = A \in \mathcal{E}^*_{set} \\ v \geq a_{fol} \wedge v \leq b_{fol} & \text{if } \tau_C = [a, b] \\ \bigvee_{x \in \tau_C} v \doteq x_{fol} & \text{else} \end{cases}$$

$$constrain_\ell(v, \texttt{bool}, \tau_C) = \begin{cases} \texttt{true} & \text{if } \tau_C = \texttt{any} \\ \bigvee_{e \in A} v \doteq fol_\ell(e) & \text{if } \tau_C = A \in \mathcal{E}^*_{set} \\ \bigvee_{x \in \tau_C} v \doteq x_{fol} & \text{else} \end{cases}$$

$$constrain_\ell(v, \tau_B, *) = \texttt{true} \qquad \text{for } \tau_B \notin \{\, \texttt{int}, \texttt{bool} \,\}$$

For the non-dependent value types, we either create a disjunction of equivalence formulas, each comparing the formula variable to a value in the constant value type's set, or in the case of the base type being `int` and the constant value type being a range, we compare against the range's bounds. For dependent types, we instead compare against the translations of the expressions in τ_C. Note that in order to ensure the variables occurring in the dependent types expressions are constrained to the correct values, we need to use additional formulas to include the correct context. If our base type is neither `int` nor `bool`, we cannot infer any information about the values of the variable.

Adding program context So far, the translations have introduced new formula variables without restricting their values. To incorporate additional information from the program, we therefore introduce assignment and type contexts for the variables in an expression with the $context_\ell$ function:

- For all expressions e at a label ℓ, we include context for each formula variable in e:

$$context_\ell(e, \omega) = \bigwedge_{(v_{fal}, v_{fol}) \in vars_\ell(e) \setminus \omega} context_\ell(v_{fal}, v_{fol}, \omega \cup vars_\ell(e)).$$

  Because we will use $context_\ell$ recursively, we keep track of the pairs of FAL expressions and corresponding term variables that we have already introduced context for using a set ω to prevent endless recursion.

- For local variables l, we include the reaching definitions using an *expand* function:

$$context_\ell(l, fol_\ell(l), \omega) = expand\big(l, fol_\ell(l), reaching(\ell, l), \omega\big)$$

- For parameters x with base type $\tau_B \coloneqq decl_B(m)(x)$ and constant value type $\tau_C \coloneqq decl_C(m)(x)$, we include the context for its potential dependent type and constrain the value to its constant value type:

$$context_\ell(x, fol_\ell(x), \omega) = typecontext_\ell(\tau_B, \tau_C, \omega) \wedge constrain_\ell(fol_\ell(x), \tau_B, \tau_C)$$

- For field accesses $e.f$ with $\tau_B \coloneqq decl_B(f)$, $\tau_C \coloneqq decl_C(f)$, we also include type information, taking care to replace references to `this` in potential dependent type expressions with the expression from which we access f with $\hat{\tau}_C \coloneqq \tau_C[\texttt{this}/e]$:

$$context_\ell(e.f, fol_\ell(e.f), \omega) = typecontext_\ell(\tau_B, \hat{\tau}_C, \omega) \wedge constrain_\ell(fol_\ell(e.f), \tau_B, \hat{\tau}_C)$$

- For all other $(v_{fal}, v_{fol}) \in vars_\ell(e)$, there is no more information to include:

$$context_\ell(v_{fal}, v_{fol}, \omega) = \texttt{true}$$

For local variables l with base type τ_B and constant value type τ_C, we provide information using the reaching definitions $reaching(\ell, l)$. The *expand* function is a conjunction of implications over each reaching definition (C_i, ℓ_i), with the premise being a translation of the conditions C_i and the conclusions being the translation of the corresponding assignment at label ℓ. We, again, pass on information about which variables we have already included information for using a set ω consisting of pairs of formula expressions and corresponding formula variables:

$$expand(l, v, D, \omega) = \bigwedge_{(C,\ell)\in D} expand\big(l, v, (C, l), \omega\big)$$

$$expand(l, v, (C, \ell), \omega) = \Big(\bigwedge_{(e,\ell_e)\in C} fol_{\ell_e}(e) \doteq \texttt{true}_{fol} \Big) \rightarrow \begin{pmatrix} v \doteq v_{fol}^{\{(C,\ell)\}} \wedge def(v_{fol}^{\{(C,\ell)\}}, l, \ell, \omega) \\ \wedge \bigwedge_{(e,\ell_e)\in C} context_{\ell_e}(e, \omega) \end{pmatrix}$$

$$def(v, l, \ell, \omega) = \begin{cases} \texttt{true} & \text{if } \tau_B \notin \{\, \texttt{int}, \texttt{bool} \,\} \\ v \doteq fol_\ell(e) \wedge context_\ell(e, \omega) & \text{if } \ell : l = e \\ typecontext_\ell(\tau_B, \tau_C, \omega) & \\ \quad \wedge constrain_\ell(v, \tau_B, \tau_C) & \text{if } \ell : l = v'.m(e_1, \ldots, e_n) \end{cases}$$

Note that in the case of the reaching definition being a method call assignment, the only inferable information is the type of the local variable. In the case of a reaching definition referring to a method assignment or if the formula variable represents a parameter or field access, we instead restrict the values to ones valid within their constant value type using the $typecontext_\ell$ and $constrain_\ell$ functions, where $typecontext_\ell$ includes the context for expressions in the declared constant value type if it is a dependent type:

$$typecontext_\ell(\tau_B, \tau_C, \omega) = \begin{cases} context_\ell(e_l, \omega) \wedge context_\ell(e_r, \omega) & \text{if } \tau_C = [e_l, e_r] \in \mathcal{E}^*_{range} \\ \bigwedge_{e\in A} context_\ell(e, \omega) & \text{if } \tau_C = A \in \mathcal{E}^*_{set} \\ \texttt{true} & \text{else} \end{cases}$$

Branching conditions When type-checking an expression at a label ℓ, we know that the branching conditions $branch(\ell)$ hold. We include this context into our type check formulas with the translation defined by

$$conds_\ell \coloneqq \bigwedge_{(e,\ell_e)\in branch(\ell)} fol_{\ell_e}(e) \doteq \texttt{true}_{fol} \wedge context_{\ell_e}(e, \emptyset).$$

3.2.3 Type-checking statements

Combining the previous definitions, to check whether an expression e of base type τ_B is of constant value type τ_C, we can now create the type check formula as described before:[2]

$$violation_\ell(e, \tau_B, \tau_C) \coloneqq$$
$$res \doteq fol_\ell(e) \land context_\ell(e, \emptyset) \land conds_\ell \land typecontext_\ell(\tau_B, \tau_C, \emptyset) \land \neg constrain_\ell(res, \tau_B, \tau_C)$$

Using an SMT solver, we can (attempt to) decide the satisfiability of this formula. If $violation_\ell(e, \tau_B, \tau_C)$ is deemed unsatisfiable, we know that the corresponding expression e must be of type τ_C, since no interpretation of the formula (and thus of the variables in the program) could be found in which the variable *res* does not conform to its constant value type. If the formula is deemed satisfiable, we assume e to not be of type τ_C. If we are additionally given an interpretation of the formula, we can trace back the values of each formula variable to the original FAL expressions and thus potentially find the source of the type error.

Type-checking an assignment To type-check an expression assignment to a local variable of the form $\ell : l = e$, we simply need to check whether e satisfies the constant value type $\tau_C \coloneqq decl_C(m)(l)$. When type-checking a field assignment of the form $\ell : v.f = e$, we need to potentially perform type adaptation on the constant value type $\tau_C \coloneqq decl_C(m)(f)$, since it might be using `this` to refer to the owner of f. When type-checking, we therefore need to use the adapted type $\hat{\tau}_C \coloneqq \tau_C[\texttt{this}/v]$, since v is how we reference the object in our current context. The type adaptation of the right hand side is done as part of the formula translation and thus doesn't need to be handled here.

Type-checking a method call Type-checking a method call is more involved: it requires checking the compatibility of the arguments with the specified method parameters as well as the compatibility of the variable that the result is assigned to with the return value of the method. Let $l = v.m(e_1, \ldots, e_k)$ be a method call to a method m with $methdef(m) = m(\texttt{this}, x_1, \ldots, x_k)(\_)[\_]$. In this case, we need to perform k type checks to verify the correctness of the method call:

- For each $i = 1, \ldots, k$, we need to ensure the argument e_i is compatible with the type $decl_C(m)(x_i)$. Since the declared type on that parameter might refer to the instance of the class on which this method is called or to another parameter, we need to replace all references to `this` with the method target v and all references to other parameters with the corresponding argument expressions. Therefore, we need to check e_i against the type $decl_C(m)(x_i)[\texttt{this}/v, x_1/e_1, \ldots, x_k/e_k]$.
- We also need to ensure the type of the return value is compatible with l. Since the return type may make references to the object v using `this` or to the parameters $x_1, \ldots, x_k$, we perform the same type adaptation. To simplify the process, we require l to have exactly the adapted type $decl_C(m)(\texttt{retval})[\texttt{this}/v, x_1/e_1, \ldots, x_k/e_k]$.

[2] Note that due to simplifications, the $context_\ell$, $conds_\ell$ and $typecontext_\ell$ uses might independently include context for the same formula variables. This, however, is not an issue - our definition of $context_\ell$ still ensures that each use does not recurse infinitely and is thus well-defined.

$$
\begin{aligned}
& \mathit{violation}_7(l+1, \texttt{int}, [1,11]) \\
= \; & \mathit{res} \doteq \mathit{fol}_7(l+1) \wedge \mathit{context}_7(l+1, \emptyset) \\
& \wedge \mathit{typecontext}_7(\texttt{int}, [1,11], \emptyset) \wedge \mathit{conds}_7 \wedge \neg \mathit{constrain}_7(\mathit{res}, \texttt{int}, [1,11]) \\
= \; & \mathit{res} \doteq l^D + 1 \\
& \wedge (x \geq 0 \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_1\}} \wedge \mathit{def}(l^{\{d_1\}}, l, 5, \{(l, l^D)\}) \\
& \quad \wedge \mathit{context}_4(x \geq 0, \{(l, l^D)\}))) \\
& \wedge (\neg(x \geq 0) \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_2\}} \wedge \mathit{def}(l^{\{d_2\}}, l, 6, \{(l, l^D)\}) \\
& \quad \wedge \mathit{context}_4(\neg(x \geq 0), \{(l, l^D)\}))) \\
& \wedge \mathit{typecontext}_7(\texttt{int}, [1,11], \emptyset) \wedge \mathit{conds}_7 \wedge \neg \mathit{constrain}_7(\mathit{res}, \texttt{int}, [1,11]) \\
= \; & \mathit{res} \doteq l^D + 1 \\
& \wedge (x \geq 0 \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_1\}} \wedge l^{\{d_1\}} \doteq x \wedge \texttt{true})) \\
& \wedge (\neg(x \geq 0) \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_2\}} \wedge l^{\{d_2\}} \doteq -x \wedge \texttt{true})) \\
& \wedge \texttt{true} \wedge \neg(x < -10 \vee x > 10) \doteq \texttt{true} \wedge \neg(\mathit{res} \geq 1 \wedge \mathit{res} \leq 11) \\
= \; & \mathit{res} \doteq l^D + 1 \\
& \wedge (x \geq 0 \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_1\}} \wedge l^{\{d_1\}} \doteq x)) \\
& \wedge (x < 0 \doteq \texttt{true} \rightarrow (l^D \doteq l^{\{d_2\}} \wedge l^{\{d_2\}} \doteq -x)) \\
& \wedge (x \geq -10 \wedge x \leq 10 \doteq \texttt{true} \wedge \neg(\mathit{res} \geq 1 \wedge \mathit{res} \leq 11)
\end{aligned}
$$

Figure 3.3: The formula generated for type-checking the assignment at line 7 in the program from Figure 3.1. The *fol* subscript for the various constants, variables and operations in the formula's terms have been omitted for readability. Terms in the formula are instead marked in green, while FAL expressions are highlighted in orange.

Example 1. Consider the sample program from Figure 3.1. For type-checking the assignment at label 7 with its reaching definitions $D := \mathit{reaching}(7, l) = \{d_1, d_2\}$ with $d_1 = (\{(x \geq 0, 4)\}, 5)$ and $d_2 = (\{(\neg(x \geq 0), 4)\}, 6)$, we arrive at the formula seen in Figure 3.3. which is unsatisfiable since $l^D_{fol} \doteq |x_{fol}|$, $|x_{fol}| \leq 10$ and $\mathit{res} = l^D_{fol} + 1$ and thus $\mathit{res} \geq 1 \wedge \mathit{res} \leq 11$.

Example 2. Note, however, that a satisfiable type check formula does not conclusively prove that an expression can actually take on a value outside its constant value type. Consider the following method:

$$
\begin{aligned}
& \mathit{false\_negative}(\texttt{this}, x)(l_1, l_2)[\\
1: \quad & l_1 = x \\
2: \quad & l_2 = x.f - l_1.f \\
&]
\end{aligned}
$$

Let $decl_B(false\_negative)(x) = decl_B(false\_negative)(l_1) = c$ be a class type with $fields(c) = \{ f \}$, $decl_B(f) = \texttt{int}$, $decl_B(false\_negative)(l_2) = \texttt{int}$ and $decl_C(false\_negative)(l_2) = \{ 0 \}$. Since x and l_1 refer to the same object, we know that $x.f$ and $l_1.f$ have the same value, meaning l_2 will always be 0. However, the created type check formula will use different variables for the two expressions and thus be satisfiable:

$$violation_\ell(x.f - l_1.f, \texttt{int}, \{ 0 \}) = res \doteq ((x.f)^2_{fol} -_{fol} (l_1.f)^2_{fol}) \wedge \neg(res \doteq 0)$$

3.3 Correctness of SMT type checking

Having established the type checking process, we now want to prove the soundness of our constant value type system. We start by defining some terms that we use in our proof:

Definition 8 (Well-typed states)**.** *Let σ, ℓ be a state of a program method m. σ, ℓ is said to be* well-typed *iff*

- *for every possible access e to a declared field, parameter or local variable of base type τ_B and constant value type τ_C that is initialized, $eval_\sigma(e)$ is a value that satisfies τ_B and τ_C, and*
- *each subexpression e occurring at the statement at ℓ that is* not *equal to the left hand side of a potential assignment is* sound*, i.e. $eval_\sigma(e) \neq \bot$.*

Definition 9 (Well-typed method execution)**.** *Let $\sigma_a, \ell_a \rightsquigarrow \sigma_b, \ell_b$ be a method execution of a method m. $\sigma_a, \ell_a \rightsquigarrow \sigma_b, \ell_b$ is* well-typed *iff any state σ, ℓ reached during this method execution is well-typed.*

We then first prove that any state σ, ℓ achieved in a normal method execution of a method m can be mapped to a corresponding interpretation for any formula of the form $res \doteq fol_\ell(e)$ (where e is an expression) such that the interpretation of res has the value $eval_\sigma(e)$:

Lemma 1. *Let σ, ℓ be a valid state resulting from a well-typed method execution $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$, e be a sound FAL expression, and β an interpretation such that*

$$\forall (v_{fal}, v_{fol}) \in vars_\ell(e) : \beta(v_{fol}) = eval_\sigma(v_{fal}) \tag{3.1}$$

Then:

$$\beta \text{ is a model for } res \doteq fol_\ell(e) \Leftrightarrow \beta(res) = eval_\sigma(e)$$

(where res is a formula variable that does not occur in $fol_\ell(e)$).

Proof. We prove this lemma by structural induction over the expression e.

Base case

Case 1 e is a constant, i.e. `true`, `false`, $n \in \mathbb{N}$:

Trivial, since the translation $fol_\ell(e)$ will use a constant with the corresponding value.

Case 2 $e = v$ with v being a local variable, parameter or field access:

Trivial, as $fol_\ell(v)$ is a formula variable and any valid interpretation of $res \doteq fol_\ell(v)$ must satisfy the condition $\beta(res) = \beta(fol_\ell(v))$.

Induction step

Case 1 $e = e_1 \oplus e_2$:

We know that for all β_1 that follow 3.1 and satisfy $res_1 \doteq fol_\ell(e_1)$ and all β_2 that follow 3.1 and satisfy $res_2 \doteq fol_\ell(e_2)$, $\beta_1(res_1) = eval_\sigma(e_1)$ and $\beta_2(res_2) = eval_\sigma(e_2)$. Since both interpretations use the same program state σ, their definitions do not clash and can be combined into a new interpretation β for

$$res \doteq fol_\ell(e_1 \oplus e_2) = res \doteq fol_\ell(e_1) \oplus_{fol} fol_\ell(e_2) \tag{3.2}$$

As the semantics of $\oplus_{fol}$ in our first-order logic theory match the semantics of $\oplus$ in FAL, we thus know that the only way to satisfy 3.2 is to set $\beta(res) = eval_\sigma(e_1) \oplus eval_\sigma(e_2)$.

Case 2 $e = \odot e_1$:

Analogous to Case 1.

□

We now want to show that any two interpretations with the condition above satisfied for two different states that are part of the same method execution can be combined into a single interpretation without conflicts:

Lemma 2. *Let e_1, e_2 be sound FAL expressions occurring at labels ℓ_1, ℓ_2, respectively, σ_1, ℓ_1 and σ_2, ℓ_2 be two program states that are part of a common well-typed method execution, and β_1, β_2 be two interpretations for the first-order logic variables in $vars_{\ell_1}(e_1)$, $vars_{\ell_2}(e_2)$, respectively, such that*

- $\forall (v_{fal}, v_{fol}) \in vars_{\ell_1}(e_1) : \beta_1(v_{fol}) = eval_{\sigma_1}(v_{fal})$
- $\forall (v_{fal}, v_{fol}) \in vars_{\ell_2}(e_2) : \beta_2(v_{fol}) = eval_{\sigma_2}(v_{fal})$.

Then β_1, β_2 can be combined into a single interpretation β without definition conflicts, i.e.

- $\forall (v_{fal}, v_{fol}) \in vars_{\ell_1}(e_1) : \beta(v_{fol}) = \beta_1(v_{fol})$
- $\forall (v_{fal}, v_{fol}) \in vars_{\ell_2}(e_2) : \beta(v_{fol}) = \beta_2(v_{fol})$

is a well-defined interpretation.

Proof. We need to show that all variables occurring in both e_1 and e_2 have the same value in both β_1 and β_2. Let $(v_{fal}, v_{fol}) \in vars_{\ell_1}(e_1) \cap vars_{\ell_2}(e_2)$. W.l.o.g. let $\sigma_1, \ell_1 \rightsquigarrow \sigma_2, \ell_2$.

Case 1 v_{fal} is a local variable l:

In this case, since the formula variables for l in both expressions are the same, l must either be final (in which case it never changes its value throughout the execution) or the accesses at ℓ_1 and ℓ_2 must have the same reaching definitions, i.e. $reaching(\ell_1, l) = reaching(\ell_2, l)$ (as formula variables for non-final local variables are unique per reaching definition set). In the former case, we already know that $\beta_1(v_{fol}) = eval_{\sigma_1}(l) = eval_{\sigma_2}(l) = \beta_2(v_{fol})$. In the latter case, since both states σ_1, σ_2 are part of the same method execution, there cannot have been an assignment to l on the path between ℓ_1 and ℓ_2, as this assignment would be present in the reaching definitions for l in σ_2 but cannot be present in the reaching definitions for l in σ_1. Furthermore, all assignments to l before ℓ_1 are on the common path to both σ_1 and σ_2. Therefore, $eval_{\sigma_1}(l) = eval_{\sigma_2}(l)$, and the definition of $\beta(v_{fol})$ is well-defined.

Case 2 v_{fal} is a parameter x:

In this case, the definition of $\beta(v_{fol})$ is trivially well-defined since parameters cannot be assigned to and thus share the same value across all states in the same method execution.

Case 3 v_{fal} is a field access:

In this case, v_{fal} is either a final expression, in which case its value never changes throughout the execution and thus $\beta_1(v_{fol}) = eval_{\sigma_1}(v_{fal}) = eval_{\sigma_2}(v_{fal}) = \beta_2(v_{fol})$, or $\ell_1 = \ell_2$ and thus $\sigma_1 = \sigma_2$ because non-final field accesses at different labels must have different representing formula variables per definition of $fol_{\ell_1}/fol_{\ell_2}$. Therefore, $\beta(v_{fol})$ is well-defined.

□

Furthermore, we show that our function $constrain_\ell$ indeed only allows the constrained formula variable to be of a value within the constant value type it is constrained to:

Lemma 3. *Let $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$ be a well-typed method execution, β an interpretation, v_{fol} a formula variable for a sound expression at ℓ, $\tau_B \in \{\texttt{int}, \texttt{bool}\}$ and τ_C a constant value type for τ_B such that either:*

- $\tau_C \in \mathscr{V}_{nondep}(\tau_B)$
- $\tau_C = A \in \mathcal{E}^*_{set}$ *such that* $\beta(v'_{fol}) = eval_\sigma(v'_{fal})$ *for all* $e' \in A, (v'_{fal}, v'_{fol}) \in vars_\ell(e')$
- $\tau_C = [e_l, e_r] \in \mathcal{E}^*_{range}$ *such that* $\beta(v'_{fol}) = eval_\sigma(v'_{fal})$ *for* $e' = e_l, e_r, (v'_{fal}, v'_{fol}) \in vars_\ell(e')$

Then $constrain_\ell(v_{fol}, \tau_B, \tau_C)$ is satisfiable $\Leftrightarrow$ v_{fol} adheres to the constant value type τ_C in β, i.e.

$$\beta(v_{fol}) \in \begin{cases} \tau_C & \text{if } \tau_C \in \mathscr{V}_{nondep}(\tau_B) \\ \{\, eval_\sigma(e) : e \in A \,\} & \text{if } \tau_C = A \in \mathcal{E}^*_{set} \\ [eval_\sigma(e_l), eval_\sigma(e_r)] & \text{if } \tau_C = [e_l, e_r] \in \mathcal{E}^*_{range} \end{cases}$$

Proof. τ_C matches one of the following cases:

Case 1 $\tau_C = [a, b], a, b \in \mathbb{N}$:

Then $constrain_\ell(v_{fol}, \tau_B, \tau_C) = v_{fol} \geq a_{fol} \wedge v_{fol} \leq b_{fol}$, which is satisfied iff $\beta(v_{fol}) \in [a, b]$.

Case 2 $\tau_C \subseteq \mathbb{N}$, but τ_C is not an interval:

Then $constrain_\ell(v_{fol}, \tau_B, \tau_C) = \bigvee_{x \in \tau_C} v_{fol} \doteq x_{fol}$, which is satisfied iff $\exists x \in \tau_C : \beta(v_{fol}) = x$, which is the case iff $\beta(v_{fol}) \in \tau_C$.

Case 3 $\tau_C = A \in \mathcal{E}^*_{set}$:

Then

$$constrain_\ell(v_{fol}, \tau_B, \tau_C) = \bigvee_{e \in A} v_{fol} \doteq fol_\ell(e).$$

We know that $\bigvee_{e \in A} v_{fol} \doteq fol_\ell(e)$ is satisfied in β iff there is a $\hat{e} \in A$ such that $v_{fol} \doteq fol_\ell(\hat{e})$ is satisfied. By Lemma 1 and by the hypothesis, we know that this is the case iff $\beta(v_{fol}) = eval_\sigma(\hat{e})$ for a $\hat{e} \in A$, which is equivalent to $\beta(v_{fol}) \in \{\, eval_\sigma(e) : e \in A \,\}$.

Case 4 $\tau_C = [e_l, e_r] \in \mathcal{E}^*_{range}$:

Then

$$constrain_\ell(v_{fol}, \tau_B, \tau_C) = v_{fol} \geq fol_\ell(e_l) \wedge v_{fol} \leq fol_\ell(e_r).$$

We know by Lemma 1 that $fol_\ell(e_l)$ evaluates to $eval_\sigma(e_l)$ and $fol_\ell(e_r)$ to $eval_\sigma(e_r)$ in β. Therefore, we can conclude that $v_{fol} \geq fol_\ell(e_l) \wedge v_{fol} \leq fol_\ell(e_r)$ is satisfied iff $\beta(v_{fol}) \in [eval_\sigma(e_l), eval_\sigma(e_r)]$.

Case 5 $\tau_C = \texttt{any}$:

Then v_{fol} always adheres to τ_C, and $constrain_\ell(v_{fol}, \tau_B, \tau_C) = \texttt{true}$, which is trivially satisfiable in all interpretations.

□

Lastly, we need to prove that in any interpretation corresponding to a valid program state, our context formulas $context_\ell$ are satisfiable:

Lemma 4. *Let $\sigma^m_{init}, \ell^m_{init} = \sigma_1, \ell_1 \rightsquigarrow \cdots \rightsquigarrow \sigma_k, \ell_k = \sigma, \ell$ be a well-typed method execution with $\ell_1 \prec \ell_2 \prec \cdots \prec \ell_k$ and e be a sound expression at label ℓ. Let ω be a set of tuples of FAL expressions with corresponding formula term variables. Then there is a model with an interpretation β such that*

$$\forall \ell_k = \ell_1, \ldots, \ell_n, \text{expressions } e' \text{ at label } \ell_k \text{ and } (v_{fal}, v_{fol}) \in vars_{\ell_k}(e') : \beta(v_{fol}) = eval_{\sigma_k}(v_{fal}) \tag{3.3}$$

that satisfies the formula

$$context_\ell(e, \omega) = \bigwedge_{(v_{fal}, v_{fol}) \in vars_\ell(e) \setminus \omega} context_\ell(v_{fal}, v_{fol}, \omega \cup vars_\ell(e)).$$

Proof. Let β be an interpretation that fulfils the condition 3.3, and let $\omega' \coloneqq \omega \cup vars_\ell(e)$. This definition of β is well-defined by Lemma 2. We will be able to recursively use this lemma for any use of the form $context_*(*, \omega')$ because, due to the definition of *context*, context is never added for a variable which is already contained in ω'. Therefore, the recursion depth is bound, and we always arrive at a base case.

We now want to show that for all $(v_{fal}, v_{fol}) \in vars_\ell(e) \setminus \omega$, $context_\ell(v_{fal}, v_{fol}, \omega')$ is satisfiable with β:

Case 1 v_{fal} is a local variable l with $\tau_B \coloneqq decl_B(m)(l)$, $\tau_C \coloneqq decl_C(m)(l)$:

In this case,

$$\begin{aligned} context_\ell(v_{fal}, v_{fol}, \omega') &= expand\big(l, v_{fol}, reaching(\ell, l), \omega'\big) \\ &= \bigwedge_{(C,\ell_C)\in reaching(\ell,l)} expand(l, v_{fol}, (C, \ell_C), \omega'). \end{aligned}$$

To prove that the formula above is satisfiable under β, we show the satisfiability of

$$\begin{aligned} &expand(l, v_{fol}, D, \omega') \\ = &\underbrace{\Big(\bigwedge_{(e',\ell_{e'})\in C} fol_{\ell_{e'}}(e') \doteq \texttt{true}_{fol}\Big)}_{P(C,\ell_C)} \rightarrow \Big(\underbrace{v \doteq v_{fol}^{\{D\}}}_{Q_1(C,\ell_C)} \wedge \underbrace{def(v_{fol}^{\{D\}}, l, \ell_C, \omega')}_{Q_2(C,\ell_C)} \wedge \underbrace{\bigwedge_{(e',\ell_{e'})\in C} context_{\ell_{e'}}(e', \omega')}_{Q_3(C,\ell_C)}\Big) \end{aligned}$$

for each $D \coloneqq (C, \ell_C) \in reaching(\ell, l)$. We know that there is exactly one reaching definition $(C_i, \ell_{C_i}) \in reaching(\ell, l)$ such that

- $\forall (e', \ell_{e'}) \in C_i : \ell_{e'}$ is on the path from $\sigma^m_{init}, \ell^m_{init}$ to σ, ℓ in a state $\sigma_{e'}$, i.e. $\ell_{e'} \in \{\ell_1, \ldots, \ell_n\}$
- In all such $\sigma_{e'}$: $eval_{\sigma_{e'}}(e') = \texttt{true}$, i.e. all conditions for the reaching definition on the path to the current state evaluated to `true`.

With our definition of β alongside Lemma 1, we can conclude that $P(C, \ell_C)$ evaluates to true in the model with β iff $(C, \ell_C) = (C_i, \ell_{C_i})$[3]. Therefore, if $(C, \ell_C) \neq (C_i, \ell_{C_i})$, the premise of the implication evaluates to false, meaning $expand(l, v_{fol}, (C, \ell_C), \omega')$ is already satisfiable in the chosen interpretation. Thus, we now assume $D = (C, \ell_C) = (C_i, \ell_{C_i})$.

We can recursively apply this lemma to conclude that for each $(e', \ell_{e'}) \in C$ independently, $context_{\ell_{e'}}(e', \omega')$ is satisfiable in β, because all $\ell_{e'}$ are known to be part of the method execution and all e' must be sound because they are used at a conditional statement at label $\ell_{e'}$ in the program. Therefore, $Q_3(C, \ell_C)$ is also satisfied under β.

Next, we will perform a case distinction on the definition cases of $Q_2(C, \ell_C)$:

[3] If $(C, \ell_C) \neq (C_i, \ell_{C_i})$, we know that there must be at least one condition $(e', \ell_{e'}) \in C$ for which $\ell_{e'}$ is part of the method execution, because the path towards ℓ_C must have deviated from the label path of the current execution at a conditional statement that is common to both paths, and the label of this branching statement must be part of the conditions C. Since $eval_{\sigma_{e'}}(e') \neq \texttt{true}$ for the corresponding state $\sigma_{e'}$, we thus cannot satisfy $fol_{\ell_{e'}}(e')$ in our current interpretation.

Case 1.1 $decl_B(m)(l) \notin \{ \texttt{int}, \texttt{bool} \}$:

In this case, $Q_2(C, \ell_C) = \texttt{true}$ and is thus trivially satisfiable under β.

Case 1.2 The statement at ℓ_C is an expression assignment $\ell_C : l = e'$:

In this case,

$$Q_2(C, \ell_C) = v_{fol}^{\{D\}} \doteq fol_{\ell_C}(e') \wedge context_{\ell_C}(e', \omega').$$

The satisfiability of this formula in β can be followed (recursively) from this lemma as well as Lemma 1, since ℓ_C is part of the well-typed method execution and e' must be sound at label ℓ_C.

Case 1.3 The statement at ℓ_C is a method call $\ell_C : l = v'.m(e_1, \ldots, e_n)$:

In this case,

$$Q_2(C, \ell_C) = typecontext_{\ell_C}(\tau_B, \tau_C, \omega') \wedge constrain_{\ell_C}(v_{fol}^{\{D\}}, \tau_B, \tau_C).$$

$constrain_{\ell_C}(v_{fol}^{\{D\}}, \tau_B, \tau_C)$ is satisfied by Lemma 3 if the method variable's corresponding formula variable follows its constant value type in β, which is implied by the well-typedness of the method execution. Meanwhile, τ_C is either a non-dependent type, in which case the type context is trivially satisfiable, or is a dependent type in which all its expressions are sound at the label ℓ_C, which then implies satisfiability of $typecontext_{\ell_C}(\tau_B, \tau_C, \omega')$ by Lemma 4.

Due to our definition of β and the fact that (C, ℓ_C) is the correct reaching definition for l in our state σ, ℓ, $Q_1(C, \ell_C)$ is also satisfied. In combination with the previous arguments, we can therefore conclude that $expand(l, v_{fol}, (C, \ell_C), \omega')$ is also satisfied if $(C, \ell_C) = (C_i, \ell_{C_i})$.

Case 2 v_{fal} is a parameter v with $\tau_B \coloneqq decl_B(m)(v)$, $\tau_C \coloneqq decl_B(m)(v)$:

In this case,

$$context_\ell(v_{fal}, v_{fol}, \omega') = typecontext_\ell(v_{fol}, \tau_B, \tau_C, \omega') \wedge constrain_\ell(v_{fol}, \tau_B, \tau_C).$$

If $\tau_B \notin \{ \texttt{int}, \texttt{bool} \}$, both parts of the formula are trivially satisfiable. Otherwise, by Lemma 3 and 4 we know that this formula is satisfiable if v follows the constant value type τ_C, which is implied by the well-typedness of the method execution.

Case 3 v_{fal} is a field access $e.f$ with $\tau_B \coloneqq decl_B(f)$, $\tau_C \coloneqq decl_B(f)$:

In this case,

$$\begin{aligned} context_\ell(v_{fal}, v_{fol}, \omega') = {} & typecontext_\ell(v_{fol}, \tau_B, \tau_C[\texttt{this}/e], \omega') \\ & \wedge constrain_\ell(v_{fol}, \tau_B, \tau_C[\texttt{this}/e]). \end{aligned}$$

This case is analogous to Case 2, although we have to replace references to `this` in the potentially dependent type τ_C with e to make the type applicable to the current method.

Case 4 None of the other cases hold:

Then $context_\ell(v_{fal}, v_{fol}, \omega') = \texttt{true}$, which is trivially satisfiable with β.

□

Given our lemmas, we can now prove the soundness of our constant value type system:

Theorem 1 (SMT constant value type checking correctness)**.** *Let e be a sound expression at a label ℓ of base type $\tau_B \in \{\texttt{int}, \texttt{bool}\}$, and let τ_C be a constant value type such that all potential dependent type expressions occurring in τ_C are sound and final at ℓ and are of base type τ_B. Then the following holds:*

$$violation_\ell(e, \tau_B, \tau_C) \text{ is unsatisfiable}$$
$$\Downarrow$$
$$eval_\sigma(e) \text{ satisfies } \tau_C \;\forall \text{ well-typed method executions } \sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$$

Proof. Proof the contraposition:

$$\exists \text{ program state } \sigma \text{ with } \sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell \text{ well-typed} : eval_\sigma(e) \text{ does not satisfy } \tau_C$$
$$\Downarrow$$
$$violation_\ell(e, \tau_B, \tau_C) \text{ has a model}$$

Let σ be such a state, and let $x \coloneqq eval_\sigma(e)$ be a value that is not of type τ_C. By Lemmas 1 and 4, we know that there is a model for $res \doteq fol_\ell(e) \wedge context_\ell(e, \emptyset)$ with an interpretation β such that $\beta(res) = x$. Similarly, since we know that all expressions that are part of the branching conditions $branch(\ell)$ evaluated to true in the given method execution, we can follow the satisfiability of $conds_\ell = \bigwedge_{(e,\ell_e)\in branch(\ell)} fol_{\ell_e}(e) \doteq \texttt{true}_{fol} \wedge context_{\ell_e}(e, \emptyset)$ in β from the same lemmas. For $typecontext_\ell(\tau_B, \tau_C, \emptyset)$, we know that $\sigma^m_{init}, \ell^m_{init} \rightsquigarrow \sigma, \ell$ is well-typed and that if τ_C is a dependent type, then its expressions are all valid at the current label. We can thus also follow its satisfiability from Lemma 4. Since x does not follow its constant value type τ_C, we know by Lemma 3 that $constrain_\ell(res, \tau_B, \tau_C)$ is not satisfied under β; conversely, $\neg constrain_\ell(res, \tau_B, \tau_C)$ must be satisfied under β. Therefore, we can conclude that β forms an interpretation that satisfies the formula

$$res \doteq fol_\ell(e) \wedge context_\ell(e, \emptyset) \wedge conds_\ell \wedge typecontext_\ell(\tau_B, \tau_C, \emptyset) \wedge \neg constrain_\ell(res, \tau_B, \tau_C)$$
$$= violation_\ell(e, \tau_B, \tau_C)$$

giving us a model for the formula above. □

Having proved that our value type system is indeed sound, we can conclusively answer RQ1: We were successfully able to develop a system making use of first-order logic formulas and SMT solving to perform sound type checking on types specifying restrictions on the possible values of integer and boolean expressions.

4 Implementation

We implement our constant value type system within a fork of the Checker Framework [16] for Java, a type-checking framework for verifying various additional program properties using pluggable types that are declared with annotations. As discussed before, the Checker Framework already includes a basic type checker for constant values, however, its capabilities are rather limited as it relies on simple inference rules. Our implementation [8] extends the framework's Constant Value Checker such that any type check that fails with the included Checker invokes our SMT type-checking instead. Furthermore, we extend the framework's non-dependent value types with our dependent expression types.

4.1 Overview

The SMT value checking extension implemented as part of this thesis is integrated into the existing Value Checker, but disabled by default. To enable SMT value checking, the `-AsmtValueChecking` flag has to be passed on the command line when running the Checker Framework, alongside the `-processor org.checkerframework.common.value.ValueChecker` option for the framework to enable value checking as a whole. Further options are available to configure the SMT type checking process:

- `-AsmtSolver=<solvers>`: specifies the SMT solver(s) to use for deciding the satisfiability of the type checking formulas, separated by commas.
- `-AsmtTimeout=<timeout>`: a positive integer value specifying the amount of seconds to wait for a result from the SMT solver before giving up and returning a failed type check result.
- `-AsmtUseSmtIntegers`: an experimental option that uses integer theories instead of bitvector theories when type-checking Java integer variables. This option is prone to crashing as it cannot deal with some operation types, such as bitwise operations, and does not consider overflow behaviour.
- `-AsmtForceSmtChecking`: a debugging option that forces the SMT value checking subsystem to run even for type checks which where already successfully verified with the default value Checker.

The Value Checker's type checking process now roughly follows these steps if the SMT extension is enabled:

1. The Value Checker generates a control flow graph for the given part of the code.

2. The Checker walks through the control flow graph from start to finish using a *forward transfer function* that assigns each control flow graph node a value type based on the types of its children (e.g. a binary operation is assigned a type based on the type of its two operands) or a type matching its value if it is a constant, using a set of manual inference and transfer rules.
3. Once the transfer function has finished, the Checker then revisits the Abstract Syntax Tree (AST) of the analysed code and, when discovering locations at which a type check is necessary (such as an assignment or a passed parameter), checks whether the required type is a supertype of the type found via the transfer function.
4. If any such type checks fail, the Checker instead invokes the SMT value checking subsystem, which then proceeds to create a corresponding type check formula and passes it on to one or multiple SMT solvers to attempt to decide the satisfiability.
5. If the SMT solver does not terminate within the specified timeout limit, a timeout error is returned. Otherwise, if any solver decides that the type check formula is satisfiable, a failed type check result is returned alongside further information about values of expressions in the code with which a type error occurs, which are gather from the interpretation that the SMT solver has found. If any of the solvers consider the formula satisfiable, the Checker deems the type check successful.

4.2 Applying SMT value checking to Java

To denote its various additional types within program source, the Checker Framework makes use of type annotations. Within the Value Checker, there are already a few non-dependent value annotations related to integer and boolean types:

- `@IntVal`: This annotation specifies a set of allowed values for the underlying integer type using an array of `long` constants.
- `@IntRange`: This annotation specifies a range of allowed values for an underlying integer type using a constant lower and a constant upper bound of type `long`.
- `@BoolVal`: Similarly to the `@IntVal` annotation, this type specifies the allowed constant `boolean` values for the underlying `boolean` type.

With our SMT value checking extension, we introduce three additional *dependent* value types:

- `@IntValExpr`: This annotation allows the specification of an array of Java integer expressions of which at least one, in a given program execution, should evaluate to the value of the annotated integer type.
- `@IntRangeExpr`: This annotation allows the specification of an array of integer expressions for both the lower and the upper bounds of an annotated integer type. In any given method program execution, the values of such a type must be greater or equal

to the value of *all* specified lower bound expressions and less or equal to *all* upper bound expressions.

- `@BoolValExpr`: Similarly to the `@IntValExpr` type, this type specifies an array of `boolean` expressions, of which at least one expression has to evaluate to the same value as the annotated type in any given program execution.

The expressions specified on these dependent types, as is the case in the formalization, can only refer to final fields and variables. With this restriction, we do not need to consider the possibility that a dependent value type for a given variable might become invalid due to the change of another value, the tracking of which would be particularly difficult for fields due to the possibility of aliasing.

In addition to transferring the theoretical work to our implementation, several considerations had to be made around Java language features that are not part of our Field Assignment Language. This involves dealing with loop structures, in which a single assignment to a variable or field could be executed arbitrarily often instead of just once, which cannot be directly modeled in first-order logic formulas where formula variables can only have a single value. To get around this restriction, we detect loops in the reaching definitions of a variable use by calculating the reaching definitions of the variable at previous locations in the program and determining whether some previous reaching definition refers to the assignment of a current reaching definition. For these variable uses, we do not include any assignment context and instead only include the most specific constant value type known for the variable (which is either a declared constant value type or a type inferred by the manual typing rules of the Value Checker) into the context for our type check formula.

Other constructs that are not part of FAL are exceptions and exception handling. In our implementation, we only consider exceptions that are caught by an explicit `catch` block in the current method, since in any program state in which a statement that is included in the assignment context for our type check throws an uncaught exception, the execution is terminated abruptly and thus never reaches the expression being type-checked. For any caught exception that can be thrown at a given statement, we create a corresponding boolean formula variable that we do not define any further in the formula (since we generally do not have information about when the exception is thrown). This variable is then set as part of the branching conditions of the corresponding `catch` block, while all other `catch` blocks and the regular non-exceptional successors of the statement have the negation of this variable set as an additional branching condition for their execution, effectively treating a `catch` block as an additional regular conditional block for which we do not have any further information about the condition.

4.3 Adapting the Checker Framework

In addition to the newly-introduced SMT value checking subsystem, several adjustments had to be made to the Checker Framework for SMT value checking. As mentioned in Section 3.2.1, for the purposes of adding assignment contexts to the type checking formulas,

an analysis of the reaching definitions of each variable at different points in the program is required. The Checker Framework already contains a reaching definition analysis in the `org.checkerframework.dataflow.reachingdef` package that, apart from a simple demonstration, was unused by the rest of the framework. For the purposes of the implementation, it proved sufficient to adapt this analysis, although several changes had to be made:

1. The `ReachingDefinitionStore` class used for storing the reaching definitions at a given `Node` in the control flow graph as well as the `ReachingDefinitionNode` class representing a single definition were missing getters for retrieving the values gathered during analysis, which had to be added.
2. The initial reaching definition analysis did not have a way of representing the definition coming from a passed parameter to a method. While this might seem valid initially, it results in a problem with reassignable parameters: since the analysis did not consider the parameter definition a reaching definition, it would have been difficult for clients of this analysis to infer whether a parameter could still hold its original value or not, because a parameter that has been reassigned in all branches to the current use cannot easily be differentiated from a parameter that has only been reassigned in some branches. We therefore opted to also create `ReachingDefinitionNodes` for the initial parameter value, in which case only the target variable (the parameter) and no assignment definition is stored.
3. The original `equals` and `hashCode` implementations consider two `ReachingDefinitionNodes` equal if the control flow graph `Nodes` for the assignments are equal. However, the equivalence of two `Nodes` is solely based on their syntax, meaning two reaching definitions based on assignments at different points of the program that are syntactically the same are also considered equal, despite the fact that the values of expressions occurring on the right hand side of the assignments could differ in the same execution of a method. We therefore adapted the implementation to compare the assignment in the reaching definition based on reference equality instead.

Unlike in Section 3.2.1, the reaching definition analysis here does not include the conditions for each reaching definition. In the implementation, the conditions for a reaching definition d of a variable use are instead calculated from the branching conditions directly by taking the condition for the assignment of d (or `true` if d is a parameter definition) together with the negation of the assignment conditions of all other reaching definitions for the current variable use that are *predecessors* to d, i.e. they were also a reaching definition before the assignment of d.

Another component of the Checker Framework that required modifications was the handling of `JavaExpressions`. `JavaExpressions` represent a subset of all possible expressions in Java and are used by the framework to parse and handle expressions written as part of dependent types. During the type checking process, the framework automatically modifies expressions for various reasons, such as standardizing expressions (e.g. making all implicit `this` references explicit), adapting expressions to different sites (e.g. converting references to `this` to an expression referring to the same object at a method call) and checking their validity. Such modifications are performed by parsing the expressions in the type annotations from

`Strings` into `JavaExpressions` and reserializing them to strings after their modifications to replace the previous annotations with an adapted type. However, during testing of the new dependent value types, it became apparent that the serialization of `JavaExpressions` with unary and binary operations did not properly preserve the order of operations as it did not create necessary brackets around expressions, incorrectly changing expressions like `(x + 1) * (x - 1)` to `x + 1 * x - 1`. To prevent this without more complex analysis of which brackets are necessary, we adapted the serialization of unary and binary operations to be always wrapped in brackets. This modification also required changes in the Index Checker, whose separate parsing of expressions from dependent types did not support brackets.

4.4 Integration with SMT solvers

The SMT-enabled Value Checker makes use of JavaSMT [2], a library providing a common Java interface to create formulas for and call into various supported SMT solvers to attempt to decide their satisfiability. This library abstracts away the specifics of interacting with each SMT solver and allows us to use various different solvers without having to provide support for each of them separately. For SMT value checking, we can therefore use all solvers that are compatible with JavaSMT, as long as they support the theories used in our type check formulas.

The SMT value checking subsystem implementation currently only supports integration of Java expressions of the integer types `byte`, `short`, `int`, `long` and `char` as well as the `boolean` type. While boolean expressions directly map to formulas in the *uninterpreted functions with equality theory* (a theory in which the only defined predicate is the equivalence operator), which is decidable [1], there are two common theories that could be used for integer expressions:

- *Integer theory*: a theory which makes use of the semantics of regular mathematical integers. These do not directly match the semantics of the finite Java integers, as arithmetic operations on them do not cause any overflows since the integer domain is infinite, and bitwise operations cannot be easily translated. Furthermore, for non-linear arithmetic, the integer theory is non-decidable [4].
- *Bitvector theory*: a theory using bitvectors of fixed sizes, which closely match Java integer semantics including overflows, and can also be used to model bitwise operations. As bitvectors can be viewed as an array of boolean variables, we can reduce formulas making use of bitvectors to simple propositional logic formulas, making this theory decidable[1].

To support all Java integer operations, we use the bitvector theory by default. An experimental option `-AsmtUseSmtIntegers` is available that, when used, instead translates integer

[1]The satisfiability problem for propositional logic is decidable since, for any propositional logic formula, it is possible to simply iterate over all possible interpretations of its atoms and test whether any one of them satisfy the given formula.

expressions to formulas in the integer theory, however, it is experimental and not well-tested and does not support bitwise operations or modeling of overflows.

By default, the SMT value checking subsystem makes use of the Z3 SMT solver [9]. Other SMT solvers can be selected for type-checking using the `-AsmtSolver` command-line flag, to which a list of solver names separated by commas can be provided. The list of supported names matches the names of the enum constants in `SolverContextFactory.Solvers` in JavaSMT, although the SMT Checker does not differentiate between lower- and uppercase letters in the arguments to this option. Note that our build of the Checker Framework does not ship any solvers to avoid bloating the classpath and framework archive, therefore, the chosen solvers need to be provided manually, using `-classpath` to include solvers written in a JVM-compatible language as well as bindings for native solvers, and `-J-Djava.library.path` to specify paths at which binaries for native solvers can be found. More information about the necessary native libraries and Java bindings for the solvers can be found in the JavaSMT documentation [5].

Once the type check formulas have been created, all solvers selected using the `-AsmtSolver` option are started in parallel. The SMT Checker will then wait for any one of the solver instances to return whether the formula is satisfiable or not. If a formula is satisfiable, the SMT Checker will then query a model from the solver to find an interpretation of the formula variables that lead to the satisfiability of the given formula. Using this interpretation, the Checker outputs values for each program expression corresponding to a formula variable to give the user information about which values (could) lead to a type error in the program. If none of the solvers return a result within the timeout specified in seconds using `-AsmtTimeout` (or 120 seconds if no timeout is provided), the Checker terminates all solvers and returns a negative type check result together with an error signaling a timeout.

5 Evaluation

We evaluate our implementation on the basis of a set of sample methods with Value Checker type specifications alongside existing Value Checker test cases in the Checker Framework. We first compare the results for these test cases produced by the default Value Checker implementation with ones resulting from enabling the SMT value checking extension and analyse the reasons behind different results. For this purpose, we will use the `-AsmtSolver` option to select the *Z3* [9] solver when enabling the SMT-solving subsystem. Afterwards, we compare the runtime for default and SMT-enabled value checking and assess the overhead incurred by the SMT solving layer.

5.1 Decrease in false negative type check results

5.1.1 Checker Framework-internal tests

Running the SMT value checking extension on the Checker Framework's integrated Value Checker tests yields 7 errors about expected diagnostics not being found. These are type-checked statements for which the original Value Checker would issue a type error, but which the SMT value checking subsystem no longer considers ill-typed. 4 of these statements no longer invoke type errors due to the extension's condition evaluation for branches. As an example, consider the code snippet from the Checker Framework in Figure 5.1. The non-SMT Value Checker does not consider the fact that the condition of the `if` statement at line 8 always evaluates to `true` and the program therefore always executes the statement at line 9. Thus, it considers it possible that `a` at line 13 still holds the value from line 6, which is 3 or 4, and cannot infer that `test4` at line 13 has a value between 15 and 30. In contrast, the SMT solving extension knows that the reaching definition from line 6 has the condition `false` and is therefore never the source of the current value of `a` at line 13. Thus, it concludes that its value must come from the parameter `y`, which is between 20 and 30.

In one case, the SMT extension's result differs from the one of the regular Value Checker due to a previous ill-typed statement. Consider the code snippet in Figure 5.2. Since the regular Value Checker only uses the types of the subexpressions in the expression it type-checks to infer information, it considers `j` at line 6 to be of value 2, whereas the SMT extension uses the assignments in the code to infer that `j` has value 4 and thus does not issue an error at line 6. However, as this is only a change for the semantics of cases where a previous statement already is ill-typed, it is neither a strengthening nor a weakening of the type checking capabilities.

```
public void IntegerTest(
            @IntRange(from = 3, to = 4) Integer x,
            @IntRange(from = 20, to = 30) Integer y) {
  Integer a;
  ...
  a = x;
  @IntVal({3, 4}) Integer test3 = a;
  if (true) {
      a = y;
  }
  @IntRange(from = 15, to = 30)
  // :: error: (assignment.type.incompatible)
  Integer test4 = a;
  ...
}
```

Figure 5.1: Partial code of a Value Checker test case from the Checker Framework.

```
@IntVal(2) int j = 2;
// :: error: (compound.assignment.type.incompatible)
j += 2;

// :: error: (assignment.type.incompatible)
@IntVal(4) int four = j;
```

Figure 5.2: Another code snippet from the Checker Framework's Value Checker tests.

Two type checks failed with the previous implementation of the Value Checker due to incorrect handling in the Value Checker's transfer function, which is unable to properly evaluate bitwise operations on `boolean` types, leading to trivial expressions like `false ^ true` or `false | true` not receiving the type `@BoolVal(true)` as expected. Due to the separate evaluation of failed type checks with formulas and SMT solvers, these two type checks no longer fail with the extension, although the non-SMT part of the Value Checker could be easily adapted to handle these expressions correctly as well.

5.1.2 Additional test cases with non-dependent value types

As expected, most of the type checking results do not change for the integrated Value Checker tests, as these merely test cases which the Value Checker can already handle. However, as part of this thesis, new test cases using the original set of value types were developed to demonstrate the additional strength gained due to the SMT value checking subsystem, which the regular implementation is unable to prove well-typedness for.

Correlation tracking Recall our simple example in 1.1 from the introduction. As stated, the manual typing rules for the Value Checker do not perform any tracking of correlations

```java
// The return type is the tightest correct range for this
// expression, since it evaluates to 318 for x = -27 and
// to -77 for x = -16.
// Without the SMT extension, the tightest bound known to
// the Checker is @IntRange(from=-54449, to=77785).
@IntRange(from = -77, to = 318) long degree7PolynomialPos(
        @IntRange(from = -30, to = 21) long x) {
    return ((x + 30) * (x + 20) * (x + 10)
            * x * (x - 10) * (x - 20) * (x - 30))
        / 3000000;
}
```

Figure 5.3: A 7th-degree polynomial for which the SMT extension is successfully able to prove the return value type.

```java
@IntRange(from = 5, to = 10) int precisionLossTest(
        @IntRange(from = -10, to = -5) int x,
        @IntRange(from = 5, to = 10) int y) {
    // This is the most precise type we can specify - and the most precise
    // type that the non-SMT Value Checker can store about this expression.
    @IntRange(from = -10, to = 10) int someValue = someCondition() ? x : y;

    // The default Value Checker only knows the result of this expression
    // to be within @IntRange(from = 0, to = 10), whereas the SMT system
    // has more context and can verify that this expression has the more
    // precise type @IntRange(from = 5, to = 10).
    return someValue > 0 ? someValue : -someValue;
}
```

Figure 5.4: A sample method in which the Value Checker does not persist enough information about a statement due to the limited expressiveness of the value type system.

between expressions, which often leads to larger losses of precision in the inferred types. This becomes particularly apparent with statements like `@IntVal(0) int res = x - x;`, for which the Value Checker is unable to prove correctness w.r.t the constant value type since it only considers the types for both occurrences of `x`, but not the fact that they are both equal. These problems are largely alleviated with the SMT type checking extension, as it provides context about the relations between expressions with the variable assignments to the SMT solvers, which can then prove tighter bounds for the results of calculations like the polynomial evaluation seen in Figure 5.3.

Precision losses due to limits of types In other cases, the Value Checker loses precision not due to loss of correlation, but due to the limited expressiveness of the value types. Consider the test case in Figure 5.4. We know that the variable `someValue` holds a value $x \in [-10, -5] \cup [5, 10]$. However, since we can only specify a single value type annotation,

the most precise range we can specify is $[-10, 10]$ [1], meaning we lose the fact that `someValue` never holds a value in the range $[-4, 4]$. As the default Value Checker can only use type information for a variable, it therefore can no longer prove the fact that the returned value is within the range $[5, 10]$. The SMT extension, in contrast, does not merely rely on the specified and inferred types, but also uses the assignment context, from which it can conclude that `someValue` is never assigned a value within $[-4, 4]$.

Bitwise operations The Value Checker's transfer function handles bitwise operations on range types rather imprecisely. It often only calculates the resulting range in special cases while simply returning the range containing all values of the underlying base type in all other cases:

- For signed shift operations (<< and >>), an estimate is only given if the range of the right hand side is a subset of the range $[0, 31]$.
- For unsigned right shifts (>>>), the analysis is only performed if the values of the range of the left hand side are non-negative.
- For bitwise AND (&) operations, a tighter range bound is only given when the range of the right hand side operand only encompasses a single value.
- No analysis is performed at all for bitwise OR (|) and bitwise XOR (^) operations.

Since it is rather impractical to specify results of bitwise operations with ranges in many cases, these limitations are not substantial in practice. Regardless, the SMT Value Checker enables more precise analysis of bitwise operations as demonstrated with the example in Figure 5.5.

5.1.3 Dependent type evaluation

In addition to extending the type checking process with SMT solving, we introduced new dependent value types into the value type system. When specifying these types, SMT value checking *must* be enabled when running the Value Checker as the manual type inference cannot handle these expression types. Using the dependent value types, we can annotate various simpler mathematical utility functions and prove properties about their return values. Some samples that are provable with the SMT extension can be found in Figure 5.6.

In cases where a method simply calculates a single expression, the result can often also be expressed directly in the dependent value type. This could be helpful in cases where some calculations of a method are offloaded to another method for clarity, since it gives the SMT Value

[1] We could, in theory, *specify* the type as `@IntVal({-10, -9, -8, -7, -6, -5, 5, 6, 7, 8, 9, 10})` instead. However, the Value Checker replaces an `@IntVal` annotation with more than 10 values with an `@IntRange` annotation whose range encompasses all values of the array to prevent it from checking too many combinations of values when applying the transfer function [12]. Therefore, the Value Checker would end up using the same range as mentioned above.

```
void combinedBitwiseAndAndOrPos(boolean[] odds, boolean[] evens) {
    int oddBits = ((odds[3] ? 1 : 0) << 7)
                    | ((odds[2] ? 1 : 0) << 5)
                    | ((odds[1] ? 1 : 0) << 3)
                    | ((odds[0] ? 1 : 0) << 1);
    int evenBits = ((evens[3] ? 1 : 0) << 6)
                    | ((evens[2] ? 1 : 0) << 4)
                    | ((evens[1] ? 1 : 0) << 2)
                    | (evens[0] ? 1 : 0);
    @IntVal(0) int x = oddBits & evenBits;
    // :: error: (assignment.type.incompatible)
    @IntVal(0) int y1 = oddBits & (evenBits << 1);
    // :: error: (assignment.type.incompatible)
    @IntVal(0) int y2 = (oddBits >> 1) & evenBits;
    @IntRange(from = 0, to = 0b10101010) int z1 = oddBits & (evenBits << 1);
    @IntRange(from = 0, to = 0b01010101) int z2 = (oddBits >> 1) & evenBits;
}
```

Figure 5.5: Sample code demonstrating that SMT value checking performs more precise analysis on bitwise operations. Lines with a comment starting with `:: error: ...` are lines preceding a statement at which a type checking error is to be expected. All other statements are well-typed w.r.t. the constant value types and do not produce errors with the SMT extension.

Checker more information about the values returned by a called method. Consider the methods in Figure 5.7. The method `degree6PolynomialPos` specifies the return type using the exact expression used to calculate the result, while `degree6PolynomialExpressedDifferentlyPos` instead chooses a different representation of the result that is equivalent. The latter approach can, therefore, be used to prove equivalence of two different expressions.

5.2 Overhead evaluation

To gain an insight into how enabling the SMT extension for the Value Checker affects the type-checking runtime, we first run the internal Value Checker test cases, which encompass 114 classes directly testing the Value Checker as well as 349 tests common to all Checkers for testing handling of various different features of the Java language, using three different configurations:

1. In the first configuration, we will leave SMT value checking disabled.
2. In the second configuration, we will simply enable the SMT extension using the `-AsmtValueChecking` option. In this case, the SMT extension will only become active when the default Value Checker emits a negative type-check result, i.e. if it was unable to prove an expression to be well-typed.

```java
// For a function returning the smaller of two values, we can specify that
// the return value must be one of the two parameters.
@IntValExpr({"#1", "#2"})
int min(final int first, final int second) {
    return first >= second ? first : second;
}

// This method returns either the input value if it is between min and
// max, or the bound closer to it.
// Therefore, the return value is always within the interval [min, max], if
// min <= max.
@IntRangeExpr(from = "#2", to = "#3")
int clampPos(int input,
        // Only one of the following annotations is necessary to prove
        // correctness.
        final @IntRangeExpr(to = "#3") int min,
        final @IntRangeExpr(from = "#2") int max) {
    if (input < min) {
        return min;
    }
    if (input > max) {
        return max;
    }
    return input;
}
```

Figure 5.6: Sample methods using dependent value types to specify additional properties about their return value. The well-typedness for these is provable with the SMT extension.

```java
@IntValExpr("#1 * (#1 - 10) * (#1 - 20) * (#1 - 30) * (#1 - 40) * (#1 - 50)")
int degree6PolynomialPos(final int x) {
    return x * (x - 10) * (x - 20) * (x - 30) * (x - 40) * (x - 50);
}

@IntValExpr("#1 * #1 * #1 * #1 * #1 * #1 - 150 * #1 * #1 * #1 * #1 * #1"
            + "+ 8500 * #1 * #1 * #1 * #1 - 225000 * #1 * #1 * #1"
            + "+ 2740000 * #1 * #1 - 12000000 * #1")
int degree6PolynomialExpressedDifferentlyPos(final int x) {
    return x * (x - 10) * (x - 20) * (x - 30) * (x - 40) * (x - 50);
}
```

Figure 5.7: Degree-6 polynomial calculation methods.

Configuration	Run 1	Run 2	Run 3	Ø
1	16 s 988 ms	16 s 667 ms	17 s 158 ms	16 s 938 ms
2	18 s 693 ms	19 s 146 ms	19 s 118 ms	18 s 986 ms
3	24 s 093 ms	24 s 534 ms	24 s 169 ms	24 s 265 ms

Figure 5.8: Runtimes for the Checker Framework's integrated Value Checker tests with the three different configurations specified in Section 5.2. For each configuration, the test cases were run three times on a system with an AMD Ryzen 5 5600X 6-core processor and 16 GB of memory.

3. In the third configuration, we will, in addition to enabling the SMT extension, force SMT type checking with the `-AsmtForceSmtChecking` option. With this option, the Checker always performs its SMT-based type checking for supported expressions, regardless of whether the regular type inference was already able to prove that the expression was well-typed.

As before, we will be using the Z3 SMT solver for the SMT-enabled configurations. Running the Value Checker on the integrated tests three times for each configuration on a Linux system with an AMD Ryzen 5 5600X 6-core processor and 16 GB of memory yielded the results in Figure 5.8. On average, configuration 2 (which is the recommended operation mode of SMT value checking) is about 2.048 seconds slower than the configuration without SMT checking (1) for these test cases, which amounts to an increase in runtime of about 12.1%. By contrast, when using configuration 3, the time relative to non-SMT value checking increases substantially by about 7.327 seconds or about 43.3%. In comparison to the previous result, it therefore shows that avoiding the SMT type checking process when a positive type check result has already been found is required to keep the overhead incurred by performing trivial type checks with SMT solvers within acceptable levels.

Figure 5.9 shows the time required for type-checking and compiling the additional test cases written for this thesis with SMT value checking using the Z3 solver, separated by test class. Since the regular Value Checker cannot (successfully) evaluate most of the type checks in these classes, we do not compare these results to runs where SMT solving is disabled. Overall, for type checks that successfully produce a result, the runtime remains under 2 seconds. However, whether a given type check ends up running into a timeout is not entirely obvious from the type-checked code. Consider the following method:

```
1 @IntRangeExpr(from = "0", to = "#2 - 1")
2 int moduloPos1(@IntRange(from = 0) int val, final @IntRange(from = 1) int x) {
3     return val % x;
4 }
```

Despite the rather simple nature of this statement, Z3 is unable to find a solution for the type check problem when using the bitvector theory for integers (which is the default for SMT value checking as mentioned in Chapter 4). However, when using `-AsmtUseSmtIntegers`, this statement is proven correct rather quickly, despite non-linear integer arithmetic being undecidable in general.

Test case class	Run 1	Run 2	Run 3	Ø
`Degree11PolynomialDep`	1 s 291 ms	1 s 235 ms	1 s 251 ms	1 s 259 ms
`...TimeoutCases`	1 min (⊥)	1 min (⊥)	1 min (⊥)	(1 min)
`Degree6PolynomialDep`	540 ms	473 ms	527 ms	513 ms
`IndexDep`	430 ms	375 ms	451 ms	419 ms
`MathUtilsDep`	588 ms	497 ms	524 ms	536 ms
`...TimeoutCases`	5 min (⊥)	5 min (⊥)	5 min (⊥)	(5 min)
`PolynomialDep`	294 ms	303 ms	290 ms	296 ms
`BigContext`	378 ms	403 ms	397 ms	393 ms
`BitwiseOperations`	317 ms	304 ms	308 ms	310 ms
`Loops`	228 ms	224 ms	226 ms	226 ms
`Polynomial`	1 s 625 ms	1 s 685 ms	1 s 653 ms	1 s 654 ms
`RangePrecisionLoss`	213 ms	198 ms	186 ms	199 ms
All non-timeout test cases	5 s 904 ms	5 sec 697 ms	5 sec 813 ms	5 sec 805 ms

Figure 5.9: Runtime for each test class of the additional Value Checker tests written for this thesis. These test cases were run with SMT value checking enabled (`-AsmtValueChecking`) using the Z3 SMT solver (`-AsmtSolver=Z3`) and a timeout value of 1 minute (`-AsmtTimeout=60`) on a Linux system with an AMD Ryzen 5 5600X 6-core processor with 16 GB of memory. All type checks that lead to a timeout in the SMT solver have been separated into other classes with the suffix `TimeoutCases` to show the runtime for the successful type checks.

5.3 Overall assessment

With the amount and sizes of the test cases used for this evaluation being rather limited, we can only give an approximate answer to RQ2. Nevertheless, when extrapolating from our results, we arrive at a less clear-cut answer than for RQ1. While we were able to find various areas in which the SMT solving layer now produces more precise results than the regular Value Checker analysis does, it remains difficult to assert the impact and applicability of these improvements in practical programs. Furthermore, while the introduction of dependent types does allow users to specify properties that were out of reach for the regular Value Checker's value types, we often still found it to be difficult to find more complex methods to which we could apply meaningful type specifications.

Regarding the overhead of the SMT value checking process, we found an increase of 12.1% in runtime for code that was mostly already type-checked correctly by the default Value Checking inference. This overhead is not entirely avoidable due to the design of our SMT value checking extension: without a complex analysis of the precision of the results produced by the inference rules of the Value Checker, we cannot know in advance whether invoking the SMT extension for a given type check is required or not. We thus believe this increase to be within a non-negligible, but acceptable range, although it is difficult to estimate how well this scales to large programs with a high density of value-typed code. For test cases that leverage the improved power of the type checker, we found the runtime for a handful of type-checked statements to lie between about 300-1700ms. As many of the longer-running

cases were engineered to be more difficult to solve, the overall runtime still appears to be within manageable bounds; however, we also found some simple test cases that the SMT solver was unable to find a result for within an acceptable time. Ultimately, more realistic case study programs are required to find a more precise answer to questions regarding the scalability and applicability of the SMT extension.

6 Limitations & future work

The following paragraphs will be dedicated to describing some of the limitations of the SMT value checking extension. These are either a result of the design decisions of the implementation or of deliberate simplifications made due to time constraints. Furthermore, we will discuss some areas in which improvements could be made to the SMT value checking system to extend its strength and usability.

Handling of loops The SMT extension's analysis of variables which are assigned to in loops is limited. If a variable's reaching definitions form a loop, the SMT extension gives up and only uses an inferred or declared type as information for that variable. Since only the default Value Checker's rules directly infer types for expressions that are not type-checked, this often requires the manual specification of a type to increase precision. As an example, consider the code in Figure 6.1. For the method call's parameter type checks to succeed, the SMT extension needs to know that `i >= 0 && i < array.length`. While the latter part of that condition is inferred from the branching conditions at that statement, it cannot infer the former since `i` is assigned to after each iteration based on its previous value and therefore only information about declared or inferred types is used. As the Value Checker's default inference does not infer any type for `i` and no type is declared manually, the type check at line 11 fails. In this case, the type check will succeed if `i` is instead manually annotated with `@IntRange(from=0)`. Nevertheless, improvements could be made here to

```
1  int safeAccess(
2          final int[] array,
3          @IntRangeExpr(from = "0", to = "#1.length - 1") int index) {
4      // this access is guaranteed to be safe
5      return array[index];
6  }
7
8  void sum(int[] array) {
9      int sum = 0;
10     for (int i = 0; i < array.length; i++) {
11         sum += safeAccess(array, i);
12     }
13 }
```

Figure 6.1: Sample code containing a loop in which not enough information is inferred automatically to successfully type-check the method call at line 11.

automatically infer more information about loop variables and conditions without requiring manual specification of types, especially in common loop patterns. Furthermore, in cases where a compile-time bound is known for the iteration count of the loop, one could perform *loop unrolling*, turning a loop into as many conditional statement blocks with the loop's body statements as the maximum iteration count of the loop, which could then be transformed into type check formulas as usual, thus giving the solver more information about the current state of loop variables in relation to the rest of the program.

Integration with Index Checker Within the Checker Framework, several other Checkers make use of the Value Checker as a dependency, the most notable of which is the Index Checker [13], which provides a type system that allows specifying and checking whether a variable holds a valid index for an array or `String` (when, for example, using `String.charAt(int)`). Using the Value Checker, it can gain more information about which expressions are still within the bounds of valid array access indices. However, despite this integration, the Index Checker does not directly benefit from the improvements made to the Value Checker even when enabling SMT solving because it only uses the types the Value Checker has stored for expressions or inferred during the transfer analysis. As the SMT extension is only invoked on-demand in the Value Checker's post-transfer analysis and does not store any new type information, the Index Checker does not profit from the increased strength of the Value Checker. While some efforts were made to improve this integration, it was decided that further adaptation of the Index Checker was out of the scope for this thesis. In addition to extending the usability of SMT checking within the Index Checker, one could also implement a translation of Index Checker types such as `@IndexFor("array")` (signifying that the annotated value must be a valid index for an array of name `array`) into corresponding dependent value types like `@IntRangeExpr(from = "0", to = "array.length - 1")` when the Index Checker's analysis fails to verify such a type, which could vastly improve the Index Checker's overall strength.

Theory selection for representing Java integers As mentioned in Section 4.4, the SMT value checking system makes use of the bitvector theory for representing Java's integer types in the type check formulas. While an option to use the integer theory for integer types is present, it is experimental and unsound, since it does not consider arithmetic overflows and is unable to represent some operation types such as bitwise operations. Nevertheless, in some cases (such as the sample mentioned in Section 5.2), the integer theory's solver is able to prove facts that the bitvector theory's solver does not find a solution for (in reasonable time). To leverage this, the SMT checking system could be extended in a way that first tries to translate the program expressions into formulas using the integer theory and falls back to the bitvector theory if it encounters a subexpression that it cannot translate or if the solver fails to find a solution within the duration of the current timeout value. Furthermore, overflows could be handled by translating Java's arithmetic operations to multiple operations that model overflow behaviour, as was done by Zuber [19], which also used the integer theory to represent Java integer expressions.

Expressiveness of dependent types Despite the addition of dependent types to increase the expressiveness of the value type system, there remain many rather simple cases where no significant properties can be specified using the value type system. This is often due to the fact that the value types can only specify the result as being *equal to*, *greater than* or *less than* a specified expression. To increase the expressiveness even further, additional dependent types could be introduced which, for example, specify boolean expression invariants that hold for the resulting value in relation to other variables. Alongside enabling the user to denote a type that expresses that a value is the square root of another one (using a syntax like `#val * #val == x`) or that it is even (`#val % 2 == 0`), such a type would also allow specifying multiple distinct ranges using the `&&` operator.

7 Conclusion

Within the scope of this thesis, we developed a theoretical value type checking process that decides the well-typedness of statements in a simple formal language based on the satisfiability of constructed type check formulas in first-order logic. We described the intricacies of building such formulas, including how to incorporate details about the program at hand by translating expressions and previous assignments as well as type constraints in order to maximize the power of this type system, and went on to prove the soundness of the aforementioned process, allowing us to conclusively give a positive answer to our first research question RQ1.

Based on the formal description, we then transferred the theoretical work into practice by developing an extension of the Value Checker within the Checker Framework for Java. Within this extension, we made use of SMT solvers to decide the satisfiability of our type check formulas in an effort to improve the strength and applicability of the Value Checker without manually introducing additional, more complex inference and analysis rules. Using the modularity of the JavaSMT library, we enabled the usage of various different SMT solvers in a pluggable fashion.

Finally, we analyzed the overhead incurred by our SMT extension by comparing the runtime for the Checker's existing tests for regular and SMT-enabled type checking, and evaluated its capability and strength in comparison to the existing Value Checker implementation using a limited set of additional test cases probing the limits of the regular Value Checker. We found and analyzed various scenarios for value-typed code in which the SMT extension was able to improve upon the result of the previous Checker implementation, and estimated the incurred overhead to be non-negligible, but acceptable. While we were unable to give a precise answer to our second research question RQ2 due to the limited scope and size of our evaluation test cases, we gave an answer based on extrapolation from the data at hand and found our answer to be more intricate and nuanced.

Bibliography

[1] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. 3rd ed. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1954. ISBN: 9780444533807.

[2] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. "JavaSMT 3: Interacting with SMT Solvers in Java". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 195–208. ISBN: 978-3-030-81688-9. DOI: 10.1007/978-3-030-81688-9_9.

[3] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[4] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I." In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 173–198. DOI: 10.1007/BF01700692. URL: https://doi.org/10.1007/BF01700692.

[5] *JavaSMT: Getting started*. URL: https://github.com/sosy-lab/java-smt/blob/master/doc/Getting-started.md.

[6] Florian Lanzinger et al. "Kukicha - Efficient Yet Powerful Refinement Types for Object-Oriented Languages". 2025.

[7] *LiquidHaskell Docs*. URL: https://ucsd-progsys.github.io/liquidhaskell/.

[8] Jonas Mittnacht. *Checker Framework Fork with SMT Value Checking*. Feb. 2026. DOI: 10.5281/zenodo.18788797. URL: https://doi.org/10.5281/zenodo.18788797.

[9] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[10] Frank Pfenning and André Platzer. *Lecture Notes on Dataflow Analysis*. URL: https://symbolaris.com/course/Compilers/05-dataflow.pdf.

[11] *The Checker Framework Manual: [...] - Called Methods Checker for the builder pattern and more*. URL: https://eisop.github.io/cf/manual/manual.html#called-methods-checker.

[12] *The Checker Framework Manual: [...] - Constant Value Checker*. URL: https://eisop.github.io/cf/manual/manual.html#constant-value-checker.

[13] *The Checker Framework Manual: [...] - Index Checker*. URL: https://eisop.github.io/cf/manual/manual.html#index-checker.

[14] *The Checker Framework Manual: [...] - Map Key Checker*. URL: https://eisop.github.io/cf/manual/manual.html#map-key-checker.

[15] *The Checker Framework Manual: [...] - Nullness Checker*. URL: https://eisop.github.io/cf/manual/manual.html#nullness-checker.

[16] *The EISOP Checker Framework*. URL: https://eisop.github.io/cf/.

[17] *The Java Modeling Language Home Page*. URL: https://www.cs.ucf.edu/~leavens/JML/index.shtml.

[18] *The KeY Project*. URL: https://www.key-project.org/.

[19] Johann Zuber. "More Efficient Property Types with SMT- Decidable Assertions". 46.23.01; LK 01. Abschlussarbeit - Bachelor. Karlsruher Institut für Technologie (KIT), 2025. 50 pp. DOI: 10.5445/IR/1000180975.