Karlsruhe Institute of Technology

Property Types in Java: Combining Type Systems and Deductive Verification

Master's Thesis of

Florian Lanzinger

at the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Alexander Weigl, M. Sc.
Second advisor:	Dr. rer. nat. Mattias Ulbrich
Third advisor:	DI Dr. sc. ETH Werner Dietl

17 August 2020 – 16 February 2021

I declare that I have developed and written the enclosed thesis completely by myself and have not used sources or means without declaration in the text, and that I have observed the KIT's rules for good scientific practice (*KIT-Satzung zur Sicherung guter wissenschaftlicher Praxis*).

Karlsruhe, 16 February 2021

. . . . . .

(Florian Lanzinger)

Abstract

While static type checking ensures that type errors do not occur at run time, many other kinds of run-time errors cannot be prevented by conventional type systems.

To close this gap, there exist tools for the creation of optional type systems, i.e., type systems that can be used in addition to a language's existing type system. However, these type systems are often over-approximative, i.e., they reject programs even if no error can actually occur at run time. If one wants a general static analysis that leads to fewer false positives, one can use formal verification tools. On the other hand, using these tools requires one to become familiar with them, which often includes learning a separate specification language.

This thesis presents a framework that allows programmers to define optional type systems for Java with as little specification overhead as possible, and to check the correctness of a program with regard to these type systems, while avoiding false positives.

To achieve this, we implement a type-checking pipeline that consists of a type checker developed in the Checker Framework – a framework for the creation of optional Java type systems – and KeY – a formal verification tool for Java. Whenever the type checker cannot prove that some part of a program is correct, it translates the relevant type properties to clauses in the Java Modeling Language (JML). Using this translation, KeY can then be used to prove the program's well-typedness.

We evaluate this pipeline in a case study by using it to specify and prove some correctness properties of a small program. We see that the size of the specification is smaller than it would have been if we had only used JML and that the verification overhead is quite small as well.

Thus, we show that formal verification tools can be used in conjunction with type checkers to build a verification pipeline that is easier to use than a traditional formal verification tool while also not being as over-approximative as a traditional type system.

Zusammenfassung

Eine statische Typprüfung stellt zwar sicher, dass zur Laufzeit keine Typfehler auftreten, aber viele andere Arten von Laufzeitfehlern können von herkömmlichen Typsystemen nicht verhindert werden.

Um diese Lücke zu schließen, gibt es Werkzeuge für die Erstellung optionaler Typsysteme, die zusätzlich zum existierenden Typsystem einer Sprache genutzt werden können. Allerdings sind diese Typsysteme oft überapproximativ, d.h. sie lehnen Programme ab, selbst wenn zur Laufzeit überhaupt keine Fehler auftreten können. Wenn man eine allgemeine statische Analyse will, die weniger falsch positive Ergebnisse mit sich bringt, kann man auf Werkzeuge für formale Verifikation zurückgreifen. Auf der anderen Seite muss man sich, um diese Werkzeuge verwenden zu können, in sie einarbeiten, wobei man oft auch eine separate Spezifikationssprache lernen muss.

Diese Arbeit präsentiert ein Gerüst, das es Programmierern erlaubt, mit möglichst wenig Spezifikationsaufwand optionale Typsysteme für Java zu definieren und die Korrektheit von Programmen ins Bezug auf diese Typsysteme zu überprüfen, und das dabei falsch positive Ergebnisse vermeidet.

Um das erreichen, implementieren wir eine Typprüfungs-Pipeline, die aus einem im Checker Framework – einem Gerüst für die Erstellung von optionalen Typsystemen für Java – entwickelten Typprüfer und aus KeY – einem Werkzeug für formale Verifikation für Java – besteht. Wann immer der Typprüfer nicht beweisen kann, dass ein Teil eines Programms korrekt ist, übersetzt er die relevanten Typeigenschaften in Klauseln der Java Modeling Language (JML). Mit dieser Übersetzung kann dann KeY genutzt werden, um die Wohlgetyptheit des Programms zu beweisen.

Wir evaluieren diese Pipeline in einer Fallstudie, indem wir sie nutzen, um einige Korrektheitseigenschaften eines kleinen Programms zu spezifizieren und zu beweisen. Wir sehen, dass die Größe der Spezifikation kleiner ist als wenn wir nur JML benutzt hätten und dass der Verifikationsaufwand ebenfalls recht gering ist.

So zeigen wir, dass Werkzeuge für formale Verifikation zusammen mit Typprüfern genutzt werden können, um eine Verifikations-Pipeline zu bauen, die einfacher zu benutzen ist als ein traditionelles Werkzeug für formale Verifikation und gleichzeitig nicht so überapproximativ ist wie ein traditionellen Typsystem.

Contents

1. Introduction

Many programming errors, like type errors, can be detected by the compiler. Others, however, can usually only be found at run time. This includes, among others, errors where unstated assumptions about a variable's properties are violated. For example, an integer variable which should only contain non-negative numbers may be assigned a negative number, or a string which should only contain dates of the form "1815-12-10" may be assigned a date of the form "10.12.1815".

One way to deal with this problem is to make the type system of the language stronger so that it can detect more errors at compile time.

In fact, there already exist frameworks that allow the definition of new type systems for existing programming languages. The *Checker Framework* [Die+11] allows users to write custom compile-time checkers for pluggable Java type systems. A pluggable type system is an optional type system that has no effect on a program's run-time semantics, and that can be used in conjunction with other pluggable type systems [Bra04, 2, 3]. *Bean Validation* [Mor19] offers an alternative approach by allowing users to define constraints like the date format constraint above within the program and to verify them at run time.

However, run-time verification brings with it a performance cost, and it can by its nature not guarantee that no error occurs at a program's run time. Type checkers can offer such a guarantee, but on the other hand they are limited to those properties encoded in their types, and their static analysis often rejects programs that never actually behave incorrectly. By using a formal verification tool instead of a type checker, one can avoid most such false positives, but that then requires learning a specification language and the intricacies of a verification tool. That introduces a significant overhead that a compile-time type checker does not.

The first goal of this thesis is to develop a framework that allows the user to define a pluggable property type system with as little specification and verification overhead as possible. By property type system, we mean a type system in which every type is associated with a property – i.e., a boolean expression –, such that every instance of that type satisfies the property.

The second goal is to allow the user to check the well-typedness of a program with regard to such a type system while avoiding false positives.

To achieve this, we implement a type-checking pipeline that consists of a checker developed in the Checker Framework and the formal verification tool KeY [Ahr+16]. Whenever the type checker cannot prove that some part of a program is correct, it

translates the relevant property types to clauses in the Java Modeling Language (JML) [Lea+13], a behavioral specification language for Java. This JML specification can then be read and proven by KeY.

In this way, we are able to combine the ease-of-use of pluggable type systems with the power and flexibility of formal verification.

1.1. Contributions

The main contributions of this thesis are the following:

1. We define a theoretical framework for pluggable property type systems in Java. This includes the following components:
   a) The property types themselves. Property types are immutable types that are associated with two boolean expressions; the well-formedness condition determines if the type itself is well-formed, the property determines which instances belong to the type. For example, consider the property type `@Interval(min="2", max="3") int`. The user may have defined the well-formedness condition `0 <= min && min <= max` for `@Interval` types, making this type well-formed. The property may have been defined as `min <= subject && subject <= max`, meaning that every integer between `min` and `max` is an instance of that type.
   b) Type hierarchies between property types which are consistent with the properties. For example, if we have two property types `@Interval(min="2", max="3") int` and `@Interval(min="1", max="4") int`, the former can be a sub-type of the latter.
   c) An algorithm that translates the property types in a Java program to JML specifications, while taking into account the knowledge gained from the type checker.
   d) Different theoretical notions of program correctness based on the type rules and JML. A program may not respect the type rules given by the property type hierarchy, but still respect all JML specifications when the property types are translated to JML. We explain how this can be used to avoid false positives.
2. The implementation of this framework consists of the following components:
   a) A domain-specific language that allows the user to define a property type hierarchy and associate every property with a Java annotation type.

   b) A type checker in the Checker Framework which parses and uses multiple such hierarchy definitions to check whether all variables in the program observe the properties they are annotated with.

   c) An implementation of the aforementioned translation algorithm, which takes the program to be checked and the output of the type checker to create a JML specification. This specification can then be proven in KeY.

3. During the implementation of this thesis, some additions were made to the Checker Framework and to KeY:

   a) To accommodate the fact that the user might want to use multiple property type hierarchies simultaneously, the Checker Framework now allows a checker to have multiple instantiations of the same checker type as sub-checkers.

   b) KeY lacked support for JML `assume` and `assert` statements, which were also implemented during the development of this thesis.

1.2. Structure

Chapter 2 introduces some of the preliminaries necessary to define property types. It gives an overview over the Checker Framework, JML, and KeY, and it introduces some existing pluggable Java type systems which the property type system is based upon.

In Chapter 3, we define property types, property type hierarchies, and their translation to JML.

Chapter 4 explains all of the components in the type-checking pipeline and how they work together using some small example programs, while also discussing how this pipeline was implemented.

In Chapter 5, we evaluate this implementation in a small case study whose full source code can be found in Appendix A.

Finally, Chapter 6 concludes the thesis, summarizing the results and giving an outlook on how the approach introduced here may be refined in the future.

1.3. Related Work

As mentioned in the previous section, there already exist some frameworks that allow programmers to define and verify some properties via Java annotations, like *Bean Validation* and the *Checker Framework*.

In this section, we give an overview over those two tools, as well as other tools and type systems which this thesis is based on.

1.3.1. Run-time verification

Bean Validation [Mor19] is an example of run-time validation. It allows the programmer to define annotations via *Validator* objects. These Validators provide a boolean method which, when supplied with an object, must return `true` if and only if the object conforms to the annotation's constraints. Annotations defined this way can be used to annotate types, fields, methods, constructors, parameters, and container elements of *JavaBeans* [Ham97] only. [Mor19, 3]

Listing 1.1 shows an example. The `@NotNull` and `@Email` annotations are predefined, so no declaration and Validator needs to be provided. They ensure that a Person's `name` and `email` fields cannot be set to null and that their `email` field can only be set to well-formed email addresses.

The Validators are called at pre-defined times during run-time, e.g, whenever a setter method is called. They can also be called manually. [Mor19, 6.4]

```
1 public class Person {
2     private @NotNull String name;
3     private @NotNull @Email String email;
4
5     public String getName() { return name; }
6     public void setName(String name) { this.name = name; }
7     public String getEmail() { return email; }
8     public void setEmail(String email) { this.email = email; }
9 }
```

Listing 1.1: A Java program with annotations

1.3.2. Pluggable type system frameworks

```
1 public @Nullable String getNameFromDB(boolean useDefaultIfEmpty) {
2     String s = db.getName();
3     if (s == null && useDefaultIfEmpty) return "John Doe";
4     else return s;
5 }
6
7 public void foo(Person person) {
8     person.setName(getNameFromDB(true));
9 }
```

Listing 1.2: Possible NullPointerException

The Checker Framework [Die+11] and the property types introduced in this thesis are frameworks that allow for the addition of *optional type systems* to Java. In [Bra04, 2], optional type systems are defined as optional extensions of a language's existing type system that have no effect on the run-time semantics. I.e., any Java program that uses some checker from the Checker Framework and is correct with respect to that checker's type system is also a valid Java program with the same run-time semantics it would have without the checker.

Any type system implemented in Checker Framework also constitutes a *pluggable type system* as defined in [Bra04, 3] because the Checker Framework – as its name implies – provides a standardized framework through which multiple optional type systems may be added to the Java compiler.

Rather than defining the annotations that form these type systems via Java methods, as is the case for Bean Validation, the Checker Framework provides multiple different checker programs for different sets of related annotations, along with an API to allow the user to write their own checkers [Che20, 31].

The semantics of an annotation and in which program states the constraints have to hold depends on the individual checker.

One example for a pluggable type system implemented in the Checker Framework which is relatively easy to understand is the *nullness checker* [Che20, 3], which we use as an example in this section. An example for a more complex type system is *PUnit* [XLD20], which implements a type system for units of measurements.

If we look again at the example in Listing 1.1, we could show the correctness of that program in the Checker Framework by using the nullness checker, along with a custom e-mail checker. The nullness checker's type system would ensure at compile time that a variable of type `@NotNull Object` – in the Checker Framework it is called `@NonNull Object` instead – is never null after it has been initialized. Our custom e-mail type system would similarly ensure at compile time that a string of type `@Email String` always contains a valid e-mail address after it has been initialized.

The advantage of this when compared to something like Bean Validation is that we verify the correctness of the program at compile time, and can thus eschew run-time checks. The Checker Framework has no impact on a program's run-time performance.

The biggest disadvantage is that most type systems can only ever be a conservative approximation of the property we want to show. For example, it is very easy to write a program that never causes a `NullPointerException` but is still not well-typed according to the nullness checker. Take Listing 1.2 as an example. To prove that that program never throws a `NullPointerException`, we must show that the method `getNameFromDB()` never returns `null` if its parameter is set to `true` – not just for this class but for any subclass as well. While the nullness checker does have some capabilities to deal with such method post-conditions, those do not cover all conceivable post-conditions, meaning

that in some situations we have to live with false positives. With run-time checks such false positives can obviously not occur.

This disadvantage can be mitigated by using formal verification systems, which bring with them their own sets of advantages and disadvantages.

1.3.3. Formal verification

Formal verification systems formally prove or disprove that a given program respects a given specification. These specifications are often written in dedicated *specification languages* like the Java Modeling Language.

The *Java Modeling Language* (*JML*) [Lea+13] is a behavioral interface specification language for Java. It applies the *design-by-contract* [Mey92] methodology to Java by allowing programmers to define a method's behavior using a pre-condition – which describes the program states in which the method is allowed to be called – and a post-condition – which describes the valid program states after the method has terminated. [Lea+13, 1.1]

There exist many formal verification tools that work with JML specifications.

OpenJML [Cok11] encodes the JML-annotated Java code into a set of SMT formulas in *SMT-LIB* [BFT17] format, which can then be passed to any SMT-LIB-compatible solver.

KeY [Ahr+16] verifies the correctness of JML contracts by translating them into sequents of JavaDL formulas, JavaDL being a logic whose formulas can contain Java programs. [Ahr+16, 1.4]

While formal verification tools like KeY cannot prove everything, they lead to fewer false positives than most type systems. For example, we could annotate the method `getNameFromDB()` in Listing 1.2 with the post-condition `ensures useDefaultIfEmpty ==> \result != null;` and check that post-condition using OpenJML or KeY.

On the other hand, writing a specification and learning the intricacies of a verification system is much more difficult than using a type checker.

1.3.4. Dependent types and refinement types

The property types introduced in this thesis are very similar to *refinement types* as seen for example in *LiquidHaskell* [Vaz+14].

LiquidHaskell's refinement types allow the programmer to decorate types with logical predicates. Depending on where such a decorated type occurs, its predicate can be interpreted as a function pre-condition, a function post-condition, or an invariant.

Refinement type systems are a type of *dependent type systems*. In a language with a dependent type system, references to programs and variables can appear inside of types [Chl13, 1.2.2].

The type system of LiquidHaskell limits itself to SMT-decidable predicates. The well-typedness of a LiquidHaskell program can then always be decided by translating all refinement predicates to SMT and giving them to an SMT solver. [Vaz+14, 1, 2.1]

Full dependent type systems go a step further and make no restrictions about which values can appear in types. This of course comes at the cost of the well-typedness of a program no longer being decidable. An example for a language with full dependent types is Idris [Bra13].

2. Preliminaries

In this chapter, we give an overview over the type systems and tools presented in Section 1.3, as well as some other foundations that are necessary to understand this thesis.

Before we begin, some notes on notation.

1. For any $n : \mathbb{N}$, we define $[n] := \{1, \ldots, n\}$ to be the set containing all positive natural numbers less than or equal to n.
2. For any $n : \mathbb{N}$, we define $(e_i)_{[n]}$ to be a tuple containing n elements e_i, indexed by $i : [n]$. We also write just (e_i) if the index set is obvious from the context.
3. We sometimes use Java booleans as operators of logical symbols. For example, we write $a \longrightarrow b$ to mean that if a evaluates to `true`, then so does b (i.e., $a \longrightarrow b$ is true if and only if `!a || b` evaluates to true).
4. We use a colon (:) to denote both type membership and set membership. I.e., $x : X$ means either that x is a variable or expression of type X, or that x is an element of the set X.

2.1. Partial orders & lattices

Lattices are a kind of partial order important in type theory. In particular, the Checker Framework expects all annotation hierarchies to be bounded lattices.

> **Definition 2.1** (Partial order)**.** A *partial order* is a pair $(S, \preceq)$ consisting of a set S and a relation $\preceq$ on S such that
>
> 1. $\preceq$ is reflexive, i.e., $\forall s : S.\ s \preceq s$.
> 2. $\preceq$ is transitive, i.e., $\forall a, b, c : S.\ a \preceq b \land b \preceq c \longrightarrow a \preceq c$.
> 3. $\preceq$ is anti-symmetric, i.e., $\forall a, b : S.\ a \preceq b \land b \preceq a \longrightarrow a = b$.

Definition 2.2 (Lattice)**.** A *lattice* is a partial order $(S, \preceq)$ such that

1. $\forall a, b : S.\ \exists j : S.\ a \preceq j \land b \preceq j \land \forall c : S.\ a \preceq c \land b \preceq c \rightarrow j \preceq c$. We denote j as $a \vee b$ and call it the *least upper bound* or *join* of a and b.
2. $\forall a, b : S.\ \exists m : S.\ m \preceq a \land m \preceq b \land \forall c : S.\ c \preceq a \land c \preceq b \rightarrow c \preceq m$. We denote m as $a \wedge b$ and call it the *greatest lower bound* or *meet* of a and b.

Definition 2.3 (Bounded lattice)**.** A *bounded lattice* is a lattice $(S, \preceq)$ such that

1. $\exists \bot : S.\ \forall s : S.\ \bot \preceq s$. We call $\bot$ the *bottom element*, or *bottom* for short.
2. $\exists \top : S.\ \forall s : S.\ s \preceq \top$. We call $\top$ the *top element*, or *top* for short.

We now define *direct products* of partial order, which later allow us to combine two type systems into a single type system.

Definition 2.4 (Direct product)**.** Given two partial orders $(S_1, \preceq_1)$, $(S_2, \preceq_2)$, we define the *direct product* of those two partial orders as

$$(S_1, \preceq_1) \times (S_2, \preceq_2) := (S_1 \times S_2, \preceq_1 \times \preceq_2)$$

such that $S_1 \times S_2$ is the Cartesian product of S_1 and S_2 and $\preceq_1 \times \preceq_2$ is the relation defined by

$$(a_1, a_2)\ (\preceq_1 \times \preceq_2)\ (b_1, b_2) \iff a_1 \preceq_1 b_1 \land a_2 \preceq_2 b_2$$

2.2. The Checker Framework

In this section, we give an overview over Java annotations and the Checker Framework.

As mentioned in the introduction, the Checker Framework [Che20] is a framework for the creation of pluggable type systems for Java.

2.2.1. Java annotations

Java annotation types are a special kind of interface type [Gos+15, 9.6]. Listing 2.1 shows an example for an annotation type definition.

```java
public @interface Regex {
    String value();
}
```

Listing 2.1: Regex.java

```java
public void @Regex(value="(ab)*") String foo(int n) {
    String result = "";
    for (int i = 0; i < n; ++i) {
        result += "ab";
    }
    return result;
}
```

Listing 2.2: Regex usage

The main difference between annotation types and other interface types is that annotation types are subject to some restrictions. [Gos+15, 9.6]

1. They cannot be generic.
2. They cannot explicitly extend another type.
3. Their methods can have neither parameters nor type parameters.
4. Their methods cannot have `throws` clauses.

The main feature of annotation types is that they instantiate *annotations*, which can (usually) appear next to any declaration.

Listing 2.2 shows an example. The annotation `@Regex(value="(ab)*")` is defined by writing the `@` sign, followed by the annotation type's name, followed by a value for every method belonging to the annotation type.

Annotations are used by so-called *annotation processors* [Che20, 2.2.2], compiler plug-ins that evaluate some annotations. Some annotation processors, like those developed in the Checker Framework, use this to expand Java's type system. Some, like those that evaluate the `SuppressWarnings` annotation type[1], use it to suppress compiler warnings or change a program's semantics in some other way.

For our example annotation type `Regex`, we can imagine an annotation processor which verifies that every `String` annotated with `@Regex(value="`x`")` matches the regular expression x.

[1] `https://docs.oracle.com/javase/7/docs/api/java/lang/SuppressWarnings.html`

```
1  public @NonNull Object foo(@Nullable Object x) {
2      return x;
3  }
4
5  public @NonNull Object bar(@NonNull Object x) {
6      return x;
7  }
8
9  public @Nullable Object baz() {
10     return null;
11 }
```

Listing 2.3: Nullness example I

2.2.2. The nullness checker

Checkers developed in the Checker Framework are annotation processors that evaluate annotations on type definitions to expand Java's type system [Che20, 2].

We now go through a small example using the Checker Framework's nullness checker to illustrate the most important features of the Checker Framework. The documentation for this checker can be found under [Che20, 3].

Listing 2.3 shows three methods, which demonstrate two of the annotation types supported by the nullness checker: A variable annotated with `@NonNull` must not be null, whereas a variable annotated with `@Nullable` can be null. The `@NonNull` annotations are actually unnecessary because `@NonNull` is the default annotation, but we have written them out in this example for the sake of clarity.

Like all annotation hierarchies in the Checker Framework, these two annotations form a bounded lattice, with `@NonNull` as bottom and `@Nullable` as top. By taking the direct product of this lattice and the Java program's type hierarchy (see Definition 3.11), we obtain this checker's type hierarchy.

The checker checks that the type system defined by this hierarchy is respected. For example, it checks that `bar()` is only called with non-null arguments. If any method tries calling `bar()` with a null argument, the checker emits a type error. The checker also sees that the method `foo` is not well-typed, since it may return `null` despite being annotated with `@NonNull`. `bar` and `baz` on the other hand, are well-typed.

The Checker Framework also supports the implementation of type refinement rules [Che20, 29.7], as demonstrated by the example in Listing 2.4. Inside the `if` branch, `x` is implicitly cast to the type `@NonNull Object`, making the method `refined()` well-typed.

However, the nullness checker's refinement rules are only a conservative approximation. The method `notRefined()` also never returns `null` but the checker still emits an error.

```
1  public @NonNull Object refined(@Nullable Object x) {
2      if (x != null) {
3          return x;
4      } else {
5          return new Object();
6      }
7  }
8
9  public @NonNull Object notRefined(@Nullable Object x) {
10     boolean b = x == null;
11     if (!b) {
12         return x;
13     } else {
14         return new Object();
15     }
16 }
```

Listing 2.4: Nullness example II

Despite these issues, the Checker Framework and the nullness checker are obviously useful and powerful tools and remain widely used in practice. In fact, the typing rules could be expanded to be able to handle methods like `notRefined()`, but expanding them to be able to handle all or even most correct programs requires some sort of formal verification.

2.3. JML and KeY

In this section, we give an overview over JML and the JML verification tool KeY.

2.3.1. JML

The *Java Modeling Language* (JML) [Lea+13] allows programmers to specify the behavior of their program. The most common way to do this is to apply the *design-by-contract* [Mey92] methodology and specify a *method contract* for every method. Such a contract consists, among other things, of a pre-condition and a post-condition. When a method, which we call the *callee*, is called by a caller, the caller must prove the callee's pre-condition. It can then assume that after the callee returns, its post-condition will hold. The callee on the other hand can always assume that its pre-condition holds when it is called. It must then prove that when it returns, its post-condition holds. [Lea+13, 1.1]

Method contracts serve as a way to not only specify how a program should behave, but also as a way to divide the proof that a program is correct into smaller sub-proofs.

This is because the correctness of each method contract can be proven separately, and there is no need for the caller to know the callee's implementation as long as it knows that the callee's contract holds.

2.3.2. Method contracts

In this section, we explain how to specify a method contract in JML using a simple example.

The full documentation for JML method contracts is found under [Lea+13, 9].

The contract starts with the keywords `public behavior`.

`public` means that this contract is public and can be used by all callers.

`behavior` is the default behavior keyword and does nothing. There are other behaviors like `normal_behavior`, which would specify that this method always terminates normally, i.e., without an exception. All contracts in this thesis use the default behavior.

Next, `diverges` gives us the condition under which the method may not terminate. By writing `diverges true`, we are saying that the method may not terminate in any case. Obviously, the method in our example does always terminate, but since this thesis does not concern itself with termination proofs, we always write `diverges true` on our contracts to make the proofs easier.

Next, `requires true` is the method's pre-condition. Normally JML would also implicitly add the pre-condition `requires x != null` but we prevented that by writing `/*@nullable@*/` next to `x`'s type. `y`, however, is not nullable, so we get the additional pre-condition `requires y != null`.

Lastly, `ensures \result == x || result == y` is the post-condition. Since the return type is not `/*@nullable@*/`, we also have the additional post-condition `ensures \result != null`.

Thus, this method's contract states that when this method is called in any state in which its second argument `y` is not `null`, the method either returns a result which is identical to one of the arguments, or does not terminate.

```
/*@ public behavior
  @ diverges true;
  @ requires true;
  @ ensures \result == x || \result == y;
  @*/
public Object neverReturnsNull(/*@nullable@*/ Object x, Object y) {
    boolean b = x == null;
    if (!b) {
        return x;
    } else {
        return y;
    }
}
```

Listing 2.5: Method contract

2.3.3. Block contracts

KeY also supports block contracts. Block contracts, which were introduced in [Wac12] and further developed in [Lan18], allow programmers to specify a contract for any code block.

Listing 2.6 shows an example, which is very similar to the method contract in Listing 2.5.

A block contract works much like a method contract. In our example, we must prove first that, if the block is entered in a state in which `y != null`, either it does not terminate (since we again have the clause `diverges true`), or it sets the variable `result` to either `x` or `y`. Then, we must prove that whenever the block is entered, `y != null` holds. We can then skip the block and assume that `result == x || result == y` holds.

```
/*@ behavior
  @ diverges true;
  @ requires y != null;
  @ ensures result == x || result == y;
  @*/
{
    boolean b = x == null;
    if (!b) {
        result = x;
    } else {
        result = y;
    }
}
```

Listing 2.6: Block contract

2.3.4. Assertions and assumptions

The last JML specification element we will talk about are assertions and assumptions, which can also be read about in [Lea+13, 13.3, 13.4].

Observe the program in Listing 2.7.

It starts with an `assume` statement containing the expression `y != null`. This tells KeY that it should assume without checking that `y` is never `null` at this point in the program.

The `assert` statement in the last line tells KeY to prove that, at this point in the program, `x` can never be `null`.

Assertions and assumptions were implemented in KeY during the development of this thesis. They are transformed into equivalent block contracts, as shown in Listing 2.8.

The block contract generated by the assumption contains a clause of the kind `ensures_free`, which we have not talked about yet. This is a so-called *free post-condition*. It can be assumed by the caller (or, in the case of a block contract, by the surrounding method) to hold, but it does not have to be shown by the callee (or, in this case, the block).

```
//@ assume y != null;
x = y;
//@ assert x != null;

```

Listing 2.7: Assertions and assumptions

```
1  /*@ behavior
2    @ ensures_free y != null;
3    @*/
4  { }
5
6  x = y;
7
8  /*@ behavior
9    @ ensures x != null;
10   @*/
11 { }
12
```

Listing 2.8: Transformation into block contracts

There are also *free pre-conditions*, specified using `requires_free`, which can be assumed to hold by the callee without having to be shown by the caller.

Thus, the JML specifications in Listing 2.7 and Listing 2.8 are equivalent; they both state that, if we assume `y != null` to hold before the assignment `x = y`, then `x != null` will hold after the assignment.

2.3.5. KeY

KeY [Ahr+16] works by translating every JML method contract into a sequent of JavaDL formulas. JavaDL is a logic whose formulas can contain and reason about Java programs. [Ahr+16, 1.4]

Definition 2.5 (Sequent)**.** [Ahr+16, 2.2.2] A *sequent* is a pair of sets of formulas denoted as

$$\Phi \Longrightarrow \Psi$$

Φ is called the *antecedent* and Ψ the *succedent* of the sequent.
The sequent $\Phi \Longrightarrow \Psi$ is valid if and only if the formula

$$\bigwedge_{\phi:\Phi} \phi \rightarrow \bigvee_{\psi:\Psi} \psi$$

is valid.

It then applies rules from a sequent calculus to show the contracts' correctness.

One of the most important concepts in KeY's calculus is that of *symbolic execution*. Symbolic execution, like its name implies, executes a program step by step without ever substituting specific values for the program's variables.

We demonstrate this concept by the following (simplified) example.

Example 2.1 (Symbolic execution)**.** We start with the sequent

$$\Longrightarrow x > 0 \rightarrow [\ \texttt{if (x > 0) \{ y = 0; \} else \{ y = 1; \}}\ ]\ y = 0$$

This sequent contains only a single formula, which states that, if $x > 0$, then after executing the given program, $y = 0$ will hold.

After applying one rule, we get the equivalent sequent

$$x > 0 \Longrightarrow [\ \texttt{if (x > 0) \{ y = 0; \} else \{ y = 1; \}}\ ]\ y = 0$$

Because we know that $x > 0$, symbolically executing the `if` construct yields

$$x > 0 \Longrightarrow [\ \texttt{y = 0;}\ ]\ y = 0$$

Now, symbolically executing the assignment to `y` yields

$$x > 0, y = 0 \Longrightarrow y = 0$$

which is obviously valid.

This example should make it clear how we can use KeY as a sort of super-type-refinement tool. If we translated the `NonNull` and `Nullness` annotations from Section 2.2.2 to JML, we *could* verify that the method `notRefined` from Listing 2.4 never returns `null`.

2.4. A restricted Java fragment

We refrain from presenting property types using either a fully formalized theoretical language or an abstract language-independent presentation. Instead, we use a restricted subset of Java 8 [Gos+15], which we outline in this section.

Definition 2.6 (Well-typedness (Java))**.** We define the set of all *well-typed Java programs* $\mathfrak{WT}_{\mathfrak{Java}}$ as the set of all Java programs $\mathfrak{pr}$ such that

1. $\mathfrak{pr}$ conforms to the Java Language Specification [Gos+15],
2. generic types do not occur in $\mathfrak{pr}$,
3. assignments only occur in $\mathfrak{pr}$ as top-level statements, i.e., a statement like `(x = 5) + foo(x)` is illegal. A statement like `foo(x++)` is also illegal because the operator. `++` contains an implicit assignment.

We now define the set of types for a given Java program.

Definition 2.7 (Set of types $\mathscr{J}$)**.** For a given program $\mathfrak{pr} : \mathfrak{WT}_{\mathfrak{Java}}$, let $Cl(\mathfrak{pr})$ be the set of all classes and interfaces (not counting annotation types) in $\mathfrak{pr}$. Furthermore, let P be the set of primitive Java types.

Then $\mathscr{J}(\mathfrak{pr}) := Cl(\mathfrak{pr}) \dot\cup P \dot\cup \{\top_{\mathscr{J}}, \bot_{\mathscr{J}}, \texttt{nulltype}\}$.

We also give some informal definitions for *type contexts* and *program states*.

Definition 2.8 (Context)**.** A *context* Γ for a program $\mathfrak{pr} : \mathfrak{WT}_{\mathfrak{Java}}$ maps variables to types in $\mathscr{J}(\mathfrak{pr})$.

We say that a variable is *accessible* in Γ if it is contained in Γ's domain.

We say that an expression is *legal* in Γ if all variables used in the expression are accessible and the expression is a well-formed Java expression.

We write $\Gamma(id) = T$ if, in the context Γ, the identifier *id* refers to a variable of type T.

We write $\Gamma(id) = (T, (T_i)_{[n]})$ if, in the context Γ, the identifier *id* refers either to a method with return type T and parameter types T_i, or to a constructor of class T with parameter types T_i. In non-static methods, $p_1 : T_1$ is the method receiver `this`.

We write $\Gamma \models e : T$ if, in the context Γ, the expression e is legal, always terminates normally, and has type T.

We write $\Gamma \models e : T^c$ if, in the context Γ, the expression e is legal, always terminates normally, has type T, and is constant.

We write $\Gamma \models e : T^p$ if, in the context Γ, the expression e is legal, always terminates normally, has type T, and is pure.

We write $id : \texttt{Expr}_T^{\text{init}}(\Gamma)$ if, in the context Γ

1. the identifier *id* refers to a variable of type T that is initialized, and
2. if *id* is a field access $o.f$, then o is initialized.

A definition of *initialized* variables is given in Section 2.6.

Definition 2.9 (Program state)**.** Given a program $\mathfrak{pr} : \mathfrak{WT}_{\mathfrak{Java}}$ and its set of types $\mathscr{J}(\mathfrak{pr})$, a *program state* ς is a run-time state of $\mathfrak{pr}$.

We consider such a state to be a function which maps valid expressions to their type and value, that value being either a primitive literal or an object.

Definition 2.10 (States in context)**.** Let Γ be a context in a program $\mathfrak{pr} : \mathfrak{WT}_{\mathfrak{Java}}$.

We define $S(\Gamma)$ to be the set of all program states in which all variables that are accessible in Γ are also accessible.

Definition 2.8 references the concepts of "constant expressions" and "pure expressions", which we define below.

Intuitively, a constant expression is one whose value is known at compile time, regardless of the context in which it occurs. A pure expression's value does not have to be known at compile time, but the expression must be deterministic and side-effect free.

Definition 2.11 (Constant expression)**.** Given a context Γ, a *constant expression* is a Java expression denoting a value of primitive type or a String that – when executed in a state $\varsigma : S(\Gamma)$ – terminates normally, and is composed using only the following: [Gos+15, 15.28]

1. literals of primitive type and literals of type `String`,
2. casts to primitive types and casts to type `String`,
3. the operators `+`, `-`, `~`, `!`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||`, `?:`,
4. parenthesized expressions whose contained expression is a constant expression,
5. simple names that refer to constant variables, i.e., variables of primitive type or type `String` that are initialized with a constant expression [Gos+15, 4,12,4], and

6. qualified names of the form `TypeName.identifier` that refer to constant variables, i.e. final variables of primitive type or type String that are initialized with a constant expression.

$\texttt{Expr}^c_T$ is the set of all constant Java expressions e such that $\Gamma^G \models e : T$, where Γ^G is the global type context in which only public static members of public classes are accessible.

$\texttt{String}^c_T$ is the set of all string representations of such expressions. I.e., for every expression $e : \texttt{Expr}^c_T$, we have `"e"` $: \texttt{String}^c_T$.

$\texttt{Expr}^c_T(v_1 : V_1, \ldots, v_n : V_n)$ is the set of all constant Java expressions e such that $\Gamma \models e : T$ for the context Γ in which

1. all members accessible in the global context are accessible,
2. local variables v_i are accessible and initialized to constant expressions, and
3. nothing else is accessible.

$\texttt{String}^c_T(v_1 : V_1, \ldots, v_n : V_n)$ is the set of all string representations of such expressions.

$\texttt{Expr}^c_T(\Gamma)$ is the set of all constant Java expressions of type T in the context Γ. $\texttt{String}^c_T(\Gamma)$ is the set of all string representations of such expressions.

Definition 2.12 (Pure method)**.** A method m is *pure* if and only if

1. it is side-effect free, i.e., it does not modify any heap locations that already existed before it was called. It is still allowed to create and modify new objects.
2. it is deterministic, i.e., when called with the same parameters and in the same program state, it returns the same result.
3. all methods that override m are also side-effect free and deterministic.

Definition 2.13 (Pure expression)**.** Given a context Γ, a *pure expression* is a Java expression that – when executed in any state $\varsigma : S(\Gamma)$ – terminates normally and is composed only of

1. any constituents allowed in constant expressions.
2. calls to pure methods visible in Γ.
3. immutable, effectively final local variables, parameters, and fields that are initialized with a pure expression. Immutability is defined in Section 2.5.
4. the `this` reference.

$\texttt{Expr}_T^p$ is the set of all pure Java expressions e such that $\Gamma^G \models e : T$, where Γ^G is the global type context in which only public static members of public classes are accessible.
$\texttt{String}_T^p$ is the set of all string representations of such expressions.
$\texttt{Expr}_T^p(v_1 : V_1, \ldots, v_n : V_n)$ is the set of all pure Java expressions e such that $\Gamma \models e : T$ for the context Γ in which

1. all members accessible in the global context are accessible,
2. local variables v_i are accessible and initialized to pure expressions, and
3. nothing else is accessible.

$\texttt{String}_T^p(v_1 : V_1, \ldots, v_n : V_n)$ is the set of all string representations of such expressions.
$\texttt{Expr}_T^p(\Gamma)$ is the set of all pure Java expressions of type T in the context Γ. $\texttt{String}_T^p(\Gamma)$ is the set of all string representations of such expressions.

Definition 2.14 (Evaluation of constant expressions)**.** Let $e : \texttt{Expr}_T^c(\Gamma)$. Then we define $v_\Gamma(e)$ to be the *evaluation of e*.
For $e : \texttt{String}_T^c(\Gamma)$, we define $v_\Gamma(s)$ analogously.

Definition 2.15 (Evaluation of pure expressions)**.** Let $e : \texttt{Expr}_T^p(\Gamma)$. Then, given a program state $\varsigma : S(\Gamma)$, we define $v_{\Gamma,\varsigma}(e)$ to be the *evaluation of e in* ς.
For $e : \texttt{String}_T^p(\Gamma)$, we define $v_{\Gamma,\varsigma}(s)$ analogously.

```
1 public @Immutable class C {
2     private int field;
3     public C() { field = 42; }
4     public void foo() { field = 43; }
5 }
```

Listing 2.9: Violation of immutability property

2.5. Immutability

This section gives an overview over the immutability type system from Glacier [Cob+17], an immutability checker for Java, which is implemented in the Checker Framework [Cob+17, 3.C].

Glacier allows the programmer to mark classes as either *immutable* or *maybe-mutable* [Cob+17, 3.A]. It then guarantees *transitive class immutability*, i.e., it ensures that no heap location that is reachable from an instance of an immutable class can be modified.

For example, Glacier would emit an error for the program in Listing 2.9 because even though the class `C` is supposed to be immutable, its field is modified outside of its constructor.

We start with the definitions for immutable classes and interfaces.

Definition 2.16 (Immutable types)**.** A class is *immutable* if and only if

1. it is annotated with the annotation `@Immutable`,
2. all of its fields are of a primitive type or an immutable class type, and
3. all of its fields are effectively final, i.e., they cannot be assigned outside the class's constructors.

An interface is *immutable* if and only if it is annotated with the annotation `@Immutable`.

A class or interface that is not immutable is referred to as *maybe-mutable*.

All sub-classes of an immutable class must also be immutable. All implementing classes of an immutable interface must also be immutable.

Sub-classes of a maybe-mutable class can be both immutable and maybe-mutable. Implementing classes of a maybe-mutable interface can be both immutable and maybe-mutable.

Given a program $\mathfrak{pr} : \mathfrak{WT}_{\mathit{Immutability}}$, we define $\mathit{Immutable}(\mathfrak{pr})$ to be the set containing all primitive types and all immutable types in $\mathfrak{pr}$.

We now define what it means for variables and objects to be immutable.

These definitions differ from those in Glacier. While Glacier only considers the immutability of objects, we also consider the immutability of variables.

Definition 2.17 (Immutable object)**.** An object is immutable if and only if its class is immutable.

For simplicity's sake, we do not use Glacier's rules for array types. We instead consider arrays to never be immutable.

Definition 2.18 (Immutable variable)**.** A variable is immutable if and only if its type is either primitive or immutable.

Note that an immutable variable does not have to be final!

Finally, we define the set $\mathfrak{WT}_{Immutability}$ of programs that are well typed in this system.

Definition 2.19 (Well-typedness (immutability))**.** We define $\mathfrak{WT}_{Immutability} \subseteq \mathfrak{WT}_{\mathfrak{Java}}$ as the set of all well-typed Java programs such that all classes and interfaces that are annotated with `@Immutable` are actually immutable.

2.6. Initialization

```
1 public class C {
2     private @NonNull Object field;
3     public C() { foo(field); }
4     public static void foo(@NonNull Object arg) { }
5 }
```

Listing 2.10: Access to uninitialized field

This section presents an initialization type system based on the one from the Checker Framework [Che20, 3.8], which is itself based on [SM11].

This system allows programmers to type objects as *under initialization* or *initialized*. An object under initialization becomes initialized when its *commitment point* is reached, this being a point in the program at which the object and any heap locations reachable from the object have been initialized. [SM11, 3.1]

When coupled with another type system – e.g., the nullness checker or property types – this initialization type system can ensure that a variable's type is only taken into account after it has been initialized. For example, the program in Listing 2.10 should not be considered well-typed because even though the actual parameter `field` in the call to `foo` has the same type as the formal parameter, it has not been initialized, and thus, contains the value `null`.

[SM11] presents an initialization type system as an extension to a non-null type system. However, as stated in [SM11, 5.8], the same approach can be applied to invariant type systems like the property types from this thesis.

Definition 2.20 (Initialized)**.** An object o is *initialized* if o itself and any heap locations reachable from o have been initialized.

A variable v is *initialized* if

1. it refers to a primitive value or an initialized object, and
2. its definition is annotated with `@Initialized`.

An object or variable that is not initialized is *under initialization*.

Definition 2.21 (Type system)**.** The initialization type system consists of types $(A_{Initialization}, T_{\mathfrak{Java}})$ where $T_{\mathfrak{Java}}$ is an (unannotated) Java type and $A_{Initialization}$ is one of the following three type annotations:

1. `@Initialized`, for initialized variables.
2. `@UnderInitialization`, for variables under initialization.
3. `@UnknownInitialization`, the super-qualifier of both `@Initialized` and `@UnderInitialization`.

If the programmer does not specify an annotation, `@Initialized` is used as the default.

In this thesis, we use the initialization type system to ensure that a variable's property type is only used if that variable has been initialized.

We further restrict the use of uninitialized variables by allowing them only in constructors and so-called *helper methods*.

Definition 2.22 (Helper method)**.** A helper method is a method whose receiver parameter is annotated with `@UnderInitialization` or `@UnknownInitialization`.

We can now define the set $\mathfrak{WT}_{\mathit{Initialization}}$ of programs that observe these rules.

Definition 2.23 (Well-typedness (initialization))**.** We define $\mathfrak{WT}_{\mathit{Initialization}} \subseteq \mathfrak{WT}_{\mathfrak{Java}}$ as the set of all well-typed Java programs $\mathfrak{pr}$ such that

1. $\mathfrak{pr}$ is well-typed in the initialization type system.
2. the type modifiers `@UnderInitialization` or `@UnknownInitialization` are only applied to method receiver parameters.

3. Property annotations and type hierarchies

In this chapter, we introduce *property annotations* and *property types*, and how they can be used in a Java program. Property annotations are annotations that are associated with a boolean expression. Property types are types that are annotated with such an annotation.

For example, if we have the property annotation `@Interval(min="2", max="3")` with the associated expression `2 <= subject && subject <= 3`, then the property type `@Interval(min="2", max="3") int` contains all integers for which the associated expression evaluates to `true`. Property types are defined in Section 3.1 and Section 3.2.

We further state which hierarchies may exist between property annotations, and, by extension, which hierarchies may exist between property types by defining *property annotation lattices* in Section 3.3.

As stated in the introduction, our type-checking pipeline consists of a type checker developed in the Checker Framework, as well as the JML verification tool KeY. This means that we have two different approximations of program correctness, one for the type checker and one for JML. Section 3.4 explains how these approximations can be used together to check a program's correctness.

All Java programs that appear in this chapter and the chapter beyond are implicitly *normal* as defined below. Normal programs are subject to some further restrictions that had to be instated to make the introduction of property types feasible.

> **Definition 3.1** (Normal program)**.** We define the set of *normal* programs $\mathfrak{Normal}$ as containing all programs $\mathfrak{pr}$ such that
>
> 1. $\mathfrak{pr} : \mathfrak{WT}_{Immutability} \cap \mathfrak{WT}_{Initialization}$.
> 2. $PA(\mathfrak{pr}) = PA_{valid}(\mathfrak{pr}) \wedge PAT(\mathfrak{pr}) = PAT_{valid}(\mathfrak{pr})$.
> 3. property annotations only appear on field, parameter, and local variable declarations, on method return types, and on constructor declarations. An-

notations on `new` expressions and on casts are not supported by the type system.

4. static fields are only annotated with trivial annotations, and only trivial annotations appear in static initializers. We call an annotation $a : A$ trivial if its associated property $Prop_A(a)$ is equal to `"true"`. This is a constraint of the initialization type system because "it is in general not possible to determine modularly when static class initializers execute" [SM11, 5.3].

5. any field or method that is referred to in a property, a well-formedness condition, or an annotation parameter is public. Otherwise, the implementation of the type checker would not be able to access it when it wants to evaluate an annotation.

6. variables in an inner scope never shadow variables from an outer scope. The JML translation assumes that this is the case.

3.1. Property annotations

In this section, we formalize our notion of property annotations.

Firstly, a *property annotation type* is an annotation type that is associated with a boolean expression which we call its *property* and whose value may only depend on the annotation type's parameters and the variable being annotated, which is called the *subject*. In addition, every property annotation type has a *well-formedness condition* that is essentially a constraint on its parameters' values.

For example, if we imagine a property annotation type `Length` with two parameters `min`, `max` and the associated property `min <= subject.size() && subject.size() <= max`, a sensible well-formedness condition would be `0 <= min && min <= max`, since lists with negative lengths do not exist, and a closed interval whose upper bound is smaller than its lower bound is empty.

This intuition is captured by the following definitions:

Definition 3.2 (Property annotation type)**.** A *property annotation type* A is a Java annotation type that is associated with

1. a *subject type* $T_A : \mathscr{J}(\mathtt{pr})$,

2. a *well-formedness condition* Wf_A : `String`,

3. a *property* $Prop_A$: `String`, and

4. a list of parameters $(p_i)_{[n]}$. Every parameter p_i must be associated with an *evaluation type* $T_{p_i} : \mathscr{J}(\mathfrak{pr})$. Every parameter p_i must be of type `String`.

We define $PAT(\mathfrak{pr})$ to be the set of all property annotation types that occur in a given program $\mathfrak{pr} : \mathfrak{Normal}$.

The reason every parameter p_i of a property annotation type must be a string associated with an evaluation type $T_{p_i} : \mathscr{J}(\mathfrak{pr})$ is that this allows p_i to contain not only literals of type T_{p_i}, but also expressions.

The above definition is not complete by itself. For example, both $Prop_A$ and Wf_A are represented by a string, but the definition does not require that those strings contain well-formed Java expressions of type `boolean`.

In addition, as stated in the introduction, we only concern ourselves with annotations of immutable types. To that end, we want to require that the subject type as well as all evaluation types be immutable. In fact, we will be even stricter and require that the evaluation types be primitive types or `String`.

As an exception to this, we allow non-immutable types as long as the property is equal to `"true"`. This is because, later in this chapter, we want every variable in a program to be annotated somehow.

All of these constraints are captured by the following definition:

Definition 3.3 (Validity of property annotation types)**.** Given a program $\mathfrak{pr} : \mathfrak{Normal}$, a property annotation type A is *valid* if and only if

1. $Wf_A : \texttt{String}^{p}_{\texttt{boolean}}(p_1 : T_{p_1}, \ldots, p_n : T_{p_n})$,
2. $Prop_A : \texttt{String}^{p}_{\texttt{boolean}}(\texttt{subject} : T_A, p_1 : T_{p_1}, \ldots, p_n : T_{p_n})$,
3. $Prop_A = \texttt{"true"} \vee T_A : Immutable(\mathfrak{pr})$, and
4. $\forall i : [n],\ T_{p_i} : \mathscr{P} \cup \texttt{String}$.

A property annotation type that is not valid is called *invalid*.

We define $PAT_{valid}(\mathfrak{pr})$ to be the set of all *valid* property annotation types that occur in $\mathfrak{pr}$.

Before moving on to defining property annotations, let us look at an example of a property annotation type.

Example 3.1 (Property annotation types)**.** We define the property annotation type `Length`
with

1. $T_{\texttt{Length}} = \texttt{List}$,
2. $p_0 = min$, $p_1 = max$,
3. $T_{min} = T_{max} = \texttt{int}$,
4. $Wf_{\texttt{Length}} =$ `"min >= 0 && min <= max"`,
5. $Prop_{\texttt{Length}} =$ `"subject.size() >= min && subject.size() <= max"`.

Is this property annotation type valid?
The first condition, $Wf_{\texttt{Length}} : \texttt{String}^{p}_{\texttt{boolean}}(\texttt{min} : \texttt{int}, \texttt{max} : \texttt{int})$, holds because if the parameters `min` and `max` are initialized to pure expressions, the expression `min >= 0 && min <= max` is also pure.
The second condition, $Prop_{\texttt{Length}} : \texttt{String}^{p}_{\texttt{boolean}}(\texttt{subject} : T_A, \texttt{min} : \texttt{int}, \texttt{max} : \texttt{int})$, holds if and only if `List.size()` is a pure method that terminates normally.
The third condition, $Prop_A =$ `"true"` $\vee\, T_A : Immutable(\mathfrak{pr})$, depends on whether `List` is immutable.
The fourth condition, $\forall i : [n],\ T_{p_i} : \mathscr{P} \cup \texttt{String}$, is fulfilled because both parameters have the type `int`.
All in all, this property annotation type is valid if and only if `List` is immutable and `List.size()` is pure and terminates normally.

Now, we can define *property annotations* as instances of property annotation types.

Definition 3.4 (Property annotation)**.** A *property annotation* a is a tuple consisting of a property annotation type A and a list of actual parameters $(a_i)_{[n]}$ such that $(a_i)_{[n]}$ and the list of A's parameters have the same length.
We define $PA(\mathfrak{pr})$ to be the set of all property annotations that occur in a given program $\mathfrak{pr} : \mathfrak{Normal}$.

Like Definition 3.2, this definition is too permissible because it does not require that the property annotation's type be valid, nor that the parameter a_i contain an expression of type T_{p_i}.
These constraints are captured by the following definition:

Definition 3.5 (Validity of property annotations)**.** A property annotation a defined in a context Γ is *valid* if and only if its type A is valid and

$$\forall i : [n],\ a : \texttt{String}^{p}_{T_{p_i}}(\Gamma)$$

i.e., if every actual parameter is a pure expression of the correct type.

A property annotation that is not valid is called *invalid*.

Given a valid property annotation a, we define a_i : `String` to be a's actual parameters.

We further define $Wf_A(a) := Wf_A(a_1, \ldots, a_n)$ and $Prop_A(s, a) := Prop_A(s, a_1, \ldots, a_n)$ for any subject s.

$PA_{valid}(\mathfrak{pr})$ is the set of all *valid* property annotations that occur in a given program $\mathfrak{pr} : \mathfrak{Normal}$.

For a given property annotation type $A : PAT_{valid}(\mathfrak{pr})$, we also define $A_{valid} := A \cap PA_{valid}(\mathfrak{pr})$.

Note that we do not require that a valid property annotation a respect its type's well-formedness conditions. This is because, unless all parameters of a are constant expressions, we may not be able to determine if the well-formedness condition will hold in every program state.

Look at the following for an example of this:

Example 3.2 (Property annotations)**.** We define the following property annotations using the type `Length` from Example 3.1:

```
1 @Length(min="1", max="0") List l0 = l1;
```

`@Length(min="1", max="0")` is valid because both `1` and `0` evaluate to a result of type `int`. However, the well-formedness condition `min >= 0 && min <= max` is violated, which means that the declaration of `l0` cannot be well-typed. For details on the well-typedness of assignments and programs, see Section 3.4.

Now, we look at another example using some more complex annotations.

```
1 public @Length(min="a+c", max="b+d") List
2 concat(int a, int b, int c, int d,
3        @Length(min="a", max="b") List l0,
4        @Length(min="c", max="d") List l1) {
5     ...
6 }
```

All property annotations in this second program fragment are valid. Essentially, for all values of a, b, c, d which make the well-formedness conditions of the three property annotations true, this function maps two lists of lengths in $[a, b]$ and $[c, d]$ respectively to a list of length in $[a + c, b + d]$.

Even though both code fragments in the previous example are valid, there is still an important difference between the property annotations `@Length(min="1", max="0")` and `@Length(min="a+c", max="b+d")`.

For annotations like `@Length(min="1", max="0")` or `@Length(min="0", max="1")` or even `@Length(min="0", max="1 + 1")`, we can tell at compile-time whether or not they respect their well-formedness conditions.

But for `@Length(min="a+c", max="b+d")`, the actual parameters of the annotation depend on the method parameters. While these method parameters must be effectively final for the annotation to be valid (i.e., they must not be modified in the method body), their values cannot be known at compile time. Thus, we also cannot tell at compile-time whether this annotation is well-formed.

This difference is formalized by the following definition:

Definition 3.6 (Constant property annotation)**.** A property annotation a is called *constant in* Γ if and only if

1. $a : PA_{valid}(\mathfrak{pr})$,
2. $\forall i : [n].\ a_i : \texttt{String}^{c}_{T_{p_i}}(\Gamma)$, i.e., all of its actual parameters are constant expressions in Γ,
3. $Wf_A : \texttt{String}^{c}_{\texttt{boolean}}(p_1 : T_{p_1}, \ldots, p_n : T_{p_n})$, i.e., the well-formedness condition is a constant expression, and
4. $Prop_A : \texttt{String}^{c}_{\texttt{boolean}}(\texttt{subject} : T_A, p_1 : T_{p_1}, \ldots, p_n : T_{p_n})$, i.e., the property is a constant expression.

For a given property annotation type $A : PAT(\mathfrak{pr})$, we define $A^c(\Gamma)$ to be the set of all property annotations of type A that are constant in Γ.

Non-constant property annotations allow us to specify what are essentially dependent types. However, in this thesis, we mostly focus on constant annotations – i.e., non-dependent types.

3.2. Evaluating property annotations

In the previous section, we defined the parameters of property annotations as strings containing Java expressions of a specific type. In this section, we define an evaluation function to evaluate these expressions.

First, we define *evaluated property annotations*, which are essentially property annotations that contain values instead of expressions.

Definition 3.7 (Evaluated property annotation)**.** An *evaluated property annotation* is a pair consisting of

1. a valid property annotation type $A : PAT_{valid}(\mathfrak{pr})$.
2. a list p of values where the type of p_i is T_{p_i}, the evaluated type of A's ith parameter.

We write this pair as $A(p)$.

We further define $Wf_A(A(p)) := Wf_A(p)$ and $Prop_A(s, A(p)) := Prop_A(s, p)$ for any subject s.

Now, we can define the evaluation function as a function that maps a valid property annotation to an evaluated property annotation.

Definition 3.8 (Evaluation of a property annotation)**.** For a valid property annotation $a : A_{valid}$ with actual parameters a_i, we define the *evaluation of a in the context Γ and the program state ς* as

$$v_{\Gamma,\varsigma}(a) := A((v_{\Gamma,\varsigma}(a_i)))$$

If a is constant, the evaluation does not depend on the program state. We thus write it simply as $v_\Gamma(a)$.

This now allows us to define the concept of *well-formedness*. Whereas a valid property annotation may or may not always respect its well-formedness condition, a well-formed property annotation must respect it.

Definition 3.9 (Well-formed evaluated property annotation)**.** An evaluated property annotation $A(p)$ is *well-formed* if and only if $Wf_A(p)$.

An evaluated property annotation that is not well-formed is called *mal-formed*.

Definition 3.10 (Well-formed property annotation)**.** A property annotation $a : A$ is *well-formed* in the context Γ and the program state ς if and only if $v_{\Gamma,\varsigma}(a)$ is well-formed.

An property annotation that is not well-formed is called *mal-formed*.

Let us look at some examples, again using the `Length` property annotation type from Example 3.1.

Example 3.3 (Well-formedness)**.** We begin with a simple example.

```
1 public static final int ONE = 1;
2
3 public static void foo() {
4     @Length(min="1", max="0") List l0;     // a0
5     @Length(min="0", max="ONE") List l1;   // a1
6     @Length(min="0", max="1 + 1") List l2;// a2
7 }
```

Let Γ be the type context for `foo()`.

Then we have

$$v_\Gamma(a_0) = \texttt{Length}(1, 0)$$
$$v_\Gamma(a_1) = \texttt{Length}(0, 1)$$
$$v_\Gamma(a_2) = \texttt{Length}(0, 2)$$

with a_0 being mal-formed (because `min` > `max`), and a_1 and a_2 being well-formed.

Now for a more complicated example.

```
1 public @Length(min="n", max="n") List repeat(int n, Object obj) {
2     // Return a list that contains obj n times.
3 }
```

The property annotation a in this example is not constant. Therefore, we need a concrete program state to be able to evaluate it.

Let Γ be the type context for `repeat()`. Let ς_0 be the state in which `repeat` has just been called with the actual parameter 0. Let ς_1 be the state in which `repeat` has just been called with the actual parameter 1. Finally, let ς_{-1} be the state in which `repeat` has just been called with the actual parameter -1.

Then

$$v_{\Gamma,\varsigma_0}(a) = \texttt{Length}(0,0)$$
$$v_{\Gamma,\varsigma_1}(a) = \texttt{Length}(1,1)$$
$$v_{\Gamma,\varsigma_{-1}}(a) = \texttt{Length}(-1,-1)$$

As we can see, a is well-formed in the states ς_0, ς_1, and mal-formed in the state ς_{-1} (because $\texttt{min} < 0$).

3.3. Type hierarchies

In this section, we define the concept of a *property annotation lattice* and combine it with a hierarchy of unannotated Java types to get a *property type hierarchy*, or *type hierarchy* for short.

These type hierarchies are used later in the definition for well-typedness (Definition 3.18). Intuitively, a program is well-typed with respect to a type hierarchy if it respects the sub-type relationships defined by that hierarchy.

We first define a program $\mathfrak{pr}$'s *program hierarchy* as the partial order whose set contains all types in $\mathfrak{pr}$ and whose relation models the sub-class relationships between these types.

Definition 3.11 (Program hierarchy)**.** For a given program $\mathfrak{pr} : \mathfrak{Normal}$, $\mathfrak{pr}$'s *program hierarchy* $(\mathscr{J}(\mathfrak{pr}), \preceq_{\mathscr{J}(\mathfrak{pr})})$ is the partial order where

1. $a \preceq_{\mathscr{J}(\mathfrak{pr})} b$ if and only if
   a) $a, b : Cl(\mathfrak{pr})$ and a is a sub-class of b, or
   b) $b = \top_{\mathscr{J}}$, or
   c) $a = \bot_{\mathscr{J}}$, or
   d) $a = \texttt{nulltype} \wedge b : Cl(\mathfrak{pr})$.

Next, we define *property annotation lattices* as hierarchies which contain evaluated property annotations.

In addition, we require that the hierarchy be consistent with the annotations' properties. By this we mean that $A(p)$ can only be a sub-annotation of $B(q)$ if $Prop_A(p) \rightarrow Prop_B(q)$.

Note, however, that from $Prop_A(p) \rightarrow Prop_B(q)$ it does not necessarily follow that $A(p) \preceq B(q)$! These hierarchies, and the definition of well-typedness later, are only supposed to give an approximation of program correctness.

Definition 3.12 (Property annotation lattice)**.** For a given program $\mathfrak{pr} : \mathfrak{Normal}$ and a set of valid property annotations types $PAT \subseteq PAT(\mathfrak{pr})$ which contains elements `Top`, `Bottom` such that

1. $Wf_{\texttt{Top}} = Wf_{\texttt{Bottom}} = \texttt{"true"}$.
2. $Prop_{\texttt{Top}} = \texttt{"true"}, Prop_{\texttt{Bottom}} = \texttt{"false"}$.
3. $T_{\texttt{Top}} = T_{\texttt{Bottom}} = \top_{\mathscr{J}}$.

a *property annotation lattice* is a bounded lattice $(\mathscr{A} := v(PAT), \preceq_{\mathscr{A}})$ where

1. $v(PAT) := \bigcup_{A:PAT}\{A(p) | Wf_A(p)\}$ is the set of all well-formed evaluated property annotations with types from PAT (We define $\top := v(\texttt{Top})$, $\bot := v(\texttt{Bottom})$).
2. for all pairs $A(p), B(q) : \mathscr{A}$ such that $A(p) \preceq_{\mathscr{A}} B(q)$,
   a) $T_A \preceq_{\mathscr{J}(\mathfrak{pr})} T_B$ holds.
   b) $\forall s : T_B.\ Prop_A(s, p) \longrightarrow Prop_B(t, q)$ holds.
3. $\forall A(p) : \mathscr{A}.\ A(p) \preceq_{\mathscr{A}} \top \wedge \bot \preceq_{\mathscr{A}} A(p)$.

Given a program hierarchy and a property annotation lattice, we can combine the two into a *property type hierarchy* (*type hierarchy* for short).

This hierarchy models the sub-type relationships between annotated types that are used in the definition of well-typedness (Definition 3.18).

Definition 3.13 (Property type)**.** Given a program $\mathfrak{pr} : \mathfrak{Normal}$, a *property type* is a pair $(A(p), T)$ such that $A : PAT(\mathfrak{pr})$ and $T : \mathscr{J}(\mathfrak{pr})$

Definition 3.14 (Property type hierarchy)**.** A *property type hierarchy* $(\mathscr{T}, \preceq)$ is the direct product of a property annotation lattice $(\mathscr{A} = v(PAT), \preceq_{\mathscr{A}})$ and a program hierarchy $(\mathscr{J}(\mathfrak{pr}), \preceq_{\mathscr{J}(\mathfrak{pr})})$.

For a given program $\mathfrak{pr} : \mathfrak{Normal}$, we define the set $\mathscr{H}(\mathfrak{pr})$ of that program's type hierarchies as the set containing all type hierarchies $(\mathscr{A} = v(PAT), \preceq_{\mathscr{A}}) \times (\mathscr{J}(p), \preceq_{\mathscr{J}(p)})$ such that

1. $PAT \subseteq PAT(\mathfrak{pr})$, and

2. every local variable, field, parameter, method return type, and constructor in $\mathfrak{pr}$ is annotated with exactly one annotation $a : A$ such that $A : PAT$.

For a given type hierarchy $H = (\mathscr{A} = v(PAT), \preceq_{\mathscr{A}}) \times (\mathscr{J}(\mathfrak{pr}), \preceq_{\mathscr{J}(\mathfrak{pr})})$, we define $PAT_H := PAT$.

We also sometimes write $A : H$ when we mean $A : PAT_H$.

Because the type hierarchy is the direct product of a property annotation lattice $\mathscr{A}$ and a program hierarchy $\mathscr{J}(\mathfrak{pr})$, we have $(a : A, T) \preceq (b : B, S)$ if and only if

1. $a \preceq_{\mathscr{A}} b$, and
2. $T \preceq_{\mathscr{J}(\mathfrak{pr})} S$.

Example 3.4 (Property type hierarchies)**.** For example, let $\mathscr{A} = PAT$ be a property annotation lattice such that PAT contains the `Length` annotation type from Example 3.1, and such that $\texttt{Length}(a, b) \preceq_{\mathscr{A}} \texttt{Length}(c, d)$ if and only if the interval $[a, b]$ is contained in $[c, d]$. Let $\mathscr{J}(\mathfrak{pr})$ be a program hierarchy containing two classes `List` and `SubList` such that `SubList` extends `List`.

Then, in the type hierarchy $\mathscr{A} \times \mathscr{J}(\mathfrak{pr}) : \mathscr{H}(\mathfrak{pr})$, we get the following sub-type relationships, among others:

1. $(\texttt{Length}(2, 3), \texttt{List}) \preceq (\texttt{Length}(1, 4), \texttt{List})$
2. $(\texttt{Length}(2, 3), \texttt{SubList}) \preceq (\texttt{Length}(2, 3), \texttt{List})$
3. $(\texttt{Length}(2, 3), \texttt{SubList}) \preceq (\texttt{Length}(1, 4), \texttt{List})$

Lastly, we define the concept of a *cover*. This is a set of property annotations hierarchies such that every property annotation that appears in the program is present in exactly one lattice in the cover.

Definition 3.15 (Cover)**.** For a given program $\mathfrak{pr} : \mathfrak{Normal}$, we define the set $\mathscr{C}(\mathfrak{pr})$ of *covers of* $\mathfrak{pr}$ as the set of all tuples $(H_i)_{[n]}$ of type hierarchies such that

1. $\forall i : [n],\ H_i : \mathscr{H}(\mathfrak{pr})$,
2. $\forall i, j : [n],\ i \neq j \longrightarrow PAT_{H_i} \cap PAT_{H_j} = \emptyset$, and

3. $\bigcup_{i:[n]} PAT_{H_i} = PAT(\mathfrak{pr})$.

Together with Definition 3.14, this means that for a set C of type hierarchies to be a cover for $\mathfrak{pr}$, every local variable, field, parameter, method return type, and constructor in $\mathfrak{pr}$ must be annotated with exactly one annotation from every hierarchy in C.

Using a cover of type hierarchies instead of just a single hierarchy allows us to annotate a variable with multiple property annotations (as long as each one comes from a different hierarchy). It also allows us to have each hierarchy only contain related property annotations instead of having one giant type hierarchy containing every single property annotation.

3.4. Well-typed & correct programs

We begin this section by defining what it means for a program to be *correct*. Intuitively, this is the case if every variable always respects the properties it is annotated with.

Definition 3.16 (Correctness)**.** Let $\mathfrak{pr} : \mathfrak{Normal}$ and $H : \mathscr{H}(\mathfrak{pr})$.

Let Γ be a context.

We write $\Gamma_H(id) = (a : A, T)$ if, in the context Γ, the identifier id refers to a variable of type T that is annotated with a property annotation $a : A : H$.

We write $\Gamma_H(id) = ((a : A, T), ((a_i : A_i, T_i))_{[n]})$ if, in the context Γ, the identifier id refers to a method whose return type T is annotated with $a : A : H$ and whose ith parameter type T_i is annotated with $a_i : A_i : H$. In non-static methods, $p_1 : T_1$ is the method receiver `this`.

Furthermore, we write Γ^{id} to refer to the context in which the method or variable id refers to is defined.

We then define the set $\mathfrak{Correct}$ of *correct programs* as the set of all programs such that

1. in every program state ς, every defined local variable, field, or parameter $var : T$ which is annotated with an annotation $a : A$ is either uninitialized or satisfies
$$T \preceq T_A \wedge Wf_A(v_{\Gamma^{var},\varsigma}(a)) \wedge Prop_A(var, v_{\Gamma^{var},\varsigma}(a))$$
2. in every program state ς, every method and constructor m whose return type T is annotated with $a : A$ and whose formal parameters $p_i : T_i$ are annotated with $a_i : A_i$ returns a result $r : T$ such that
$$T \preceq T_A \wedge Wf_A(v_{\Gamma^m,\varsigma}(a)) \wedge Prop_A(r, v_{\Gamma^m,\varsigma}(a))$$

if it is called with actual parameters $p_i : T_i$ that satisfy

$$T_i \preceq T_{A_i} \wedge Wf_{A_i}(v_{\Gamma^m,\varsigma}(a_i)) \wedge Prop_{A_i}(p_i, v_{\Gamma^m,\varsigma}(a_i))$$

The type hierarchies introduced in Section 3.3 allow us to develop an approximation for program correctness in Section 3.4.1: A program is *well-typed* with respect to a type hierarchy if it respects the sub-type rules defined by that hierarchy.

We also define a translation from property annotations to JML specifications in Section 3.4.2, yielding another approximation for program correctness: A program is *JML-correct* if it respects all of its JML specifications.

Lastly, we explain in Section 3.4.4 how these two approximations together can form a better approximation for correctness than using either of them by itself.

3.4.1. Well-typedness

In this section we define typing rules for a Java program $\mathfrak{pr}$ and a hierarchy $H : \mathscr{H}(\mathfrak{pr})$. After that, we give some well-typedness rules to define when $\mathfrak{pr}$ is well-typed with respect to H.

Definition 3.17 (Property typing rules)**.** We now define the following type rules for a given property type hierarchy H:

First, every expression has a property type of the form $(\top, T)$ for some Java type T.

$$\frac{\Gamma \models e : T}{\Gamma_H \models e : (\top, T)} \text{ type-top}$$

If an expression has the type $(A(p), T)$, it also has every super-type of $(A(p), T)$.

$$\frac{\Gamma_H \models e : (A(p), T) \qquad (A(p), T) \preceq_H (B(q), S)}{\Gamma_H \models e : (B(q), S)} \text{ type-super}$$

A constant expression has the type $(A(p), T)$ if it fulfills $A(p)$'s property.

$$\frac{\begin{array}{c} e : \mathrm{Expr}^c_T(\Gamma) \\ A : H \\ T \preceq_{\mathscr{J}} T_A \wedge Wf_A(p) \wedge Prop_A(v_\Gamma(e), p) \end{array}}{\Gamma_H \models e : (A(p), T)} \text{ type-constexpr}$$

A variable *var* has the type $(A(p), T)$ if it is annotated with a constant annotation a such that $v_{\Gamma^{var}}(a) = A(p)$, and it is initialized.

$$\frac{\begin{array}{c}\Gamma_H(var) = (a : A^c(\Gamma), T) \\ var : \mathtt{Expr}_T^{\mathrm{init}}(\Gamma) \\ T \preceq_{\mathscr{J}} T_A \wedge Wf_A(v_{\Gamma^{var}}(a))\end{array}}{\Gamma_H \models var : (v_{\Gamma^{var}}(a), T)}\ \text{type-var}$$

A method call has the type $(A(p), T)$ if the method's result type is annotated with a constant annotation a such that $v_{\Gamma^m}(a) = A(p)$.

$$\frac{\begin{array}{c}\Gamma_H(m) = ((a : A^c(\Gamma^m), T), ((a_i : A_i^c(\Gamma^m), T_i))_{[n]}) \\ T \preceq_{\mathscr{J}} T_A \wedge Wf_A(v_{\Gamma^m}(a))\end{array}}{\Gamma_H \models \mathtt{m(\ldots)} : (v_{\Gamma^m}(a), T)}\ \text{type-method}$$

These typing rules are evidently not exhaustive. The checker implements some more typing rules, which have been left out because they are not germane to this theoretical treatment.

Also, most of these typing rules are only applicable for constant property annotations. While non-constant annotations are covered by the translation to JML, they are not covered by the type system. A variable that is annotated with a non-constant translation is never well-typed.

With the typing rules out of the way, we can define the well-typedness rules.

Definition 3.18 (Well-typedness)**.** Let $\mathfrak{pr} : \mathfrak{Normal}$ and a $H : \mathscr{H}(\mathfrak{pr})$.

Then $\mathfrak{pr}$ is *well-typed with respect to H* if every program construct in $\mathfrak{pr}$ is well-typed with respect to H, as defined by the following rules:

❊

An assignment is well-typed if the right-hand side and left-hand side have the same type.

$$\frac{\begin{array}{c}\Gamma_H(var) = (a : A^c(\Gamma)), T) \\ \Gamma_H \models e : (v_{\Gamma^{var}}(a), T)\end{array}}{\Gamma_H \models \mathtt{var = e;}\ \checkmark}\ \text{wt-assignment}$$

A variable declaration is well-typed if the annotation on the declaration is well-formed and constant.

$$\frac{\Gamma_H(v) = (a : A^c(\Gamma), T) \quad T \preceq_{\mathscr{J}} T_A \wedge Wf_A(v_\Gamma(a))}{\Gamma_H \models \texttt{@a}\ v\texttt{;}\ \checkmark}\ \text{wt-declaration}$$

A variable definition is well-typed if the corresponding declaration and assignment are well-typed.

$$\frac{\Gamma_H \models \texttt{@a}\ var\texttt{;}\ \checkmark \quad \Gamma_H^{var} \models var\ \texttt{=}\ e\texttt{;}\ \checkmark}{\Gamma_H \models \texttt{@a}\ var\ \texttt{=}\ e\texttt{;}\ \checkmark}\ \text{wt-definition}$$

A return statement is well-typed if the returned expression has the method's return type.

$$\frac{\Gamma_H(m) = ((a : A^c(\Gamma), T), ((a_i : A_i^c(\Gamma), T_i))_{[n]}) \quad \Gamma_H \models e : (v_{\Gamma^m}(a), T)}{\Gamma_H \models m\texttt{(...) \{ ... return } e\texttt{; ... \}}\ \checkmark}\ \text{wt-return}$$

A method or constructor call is well-typed if all actual parameters have the required types.

$$\frac{\Gamma_H(m) = ((a : A^c(\Gamma), T), ((a_i : A_i^c(\Gamma), T_i))_{[n]}) \quad \forall i : [n].\ \Gamma_H \models p_i : (v_{\Gamma^m}(a_i), T_i)}{\Gamma_H \models m(p_1, \ldots, p_n)\ \checkmark}\ \text{wt-method-call}$$

A method signature of a method m is well-typed if all annotations in the signature are constant and well-formed. In addition, if m overrides another method m', its return type must be co-variant with that of m', and its parameter types must be contra-variant with those of m' (except, of course, for the receiver parameter `this`).

$$\frac{\Gamma_H(m) = ((a : A^c(\Gamma)), T), ((a_i : A_i^c(\Gamma), T_i))_{[n]}) \quad T \preceq_{\mathscr{J}} T_A \wedge Wf_A(v_\Gamma(a)) \quad \forall i : [n].\ T_i \preceq_{\mathscr{J}} T_{A_i} \wedge Wf_{A_i}(v_\Gamma(a_i))}{\Gamma_H \models \texttt{@}a\ T\ m\texttt{(@}a_1\ T_1\ p_1\texttt{,} \ldots\texttt{,@}a_n\ T_n\ p_n\texttt{)}\ (\checkmark)}\ \text{wt-method-def-1}$$

$$\frac{\Gamma \models \texttt{@}a\ T\ m\texttt{(@}a_1\ T_1\ p_1\texttt{,} \ldots\texttt{,@}a_n\ T_n\ p_n\texttt{)}\ (\checkmark) \quad m \text{ does not override another method}}{\Gamma_H \models \texttt{@}a\ T\ m\texttt{(@}a_1\ T_1\ p_1\texttt{,} \ldots\texttt{,@}a_n\ T_n\ p_n\texttt{)}\ \checkmark}\ \text{wt-method-def-2}$$

$$\frac{\begin{array}{c}\Gamma_H \models @a\ T\ m(@a_1\ T_1\ p_1, \ldots, @a_n\ T_n\ p_n)\ (\checkmark) \\ m \text{ overrides } m' \\ \Gamma_H(m') = ((b : B^c(\Gamma), S), ((b_i : B_i^c(\Gamma), S_i))_{[n]}) \\ \Gamma'_H \models @b\ S\ m'\ (@b_1\ S_1\ q_1, \ldots, @b_n\ S_n\ q_n)\ \checkmark \\ (v_{\Gamma^m}(a), T) \preceq_H (v_{\Gamma'^{m'}}(b), S) \\ \forall i : [n] \setminus \{1\}.\ (v_{\Gamma'^{m'}}(b_i), S_i) \preceq_H (v_{\Gamma^m}(a_i), T_i)\end{array}}{\Gamma_H \models @a\ T\ m(@a_1\ T_1\ p_1, \ldots, @a_n\ T_n\ p_n)\ \checkmark}\ \text{wt-method-def-3}$$

A constructor signature is well-typed if all annotations in the signature are constant and well-formed, and if there is no non-top annotation on the constructor itself.

$$\frac{\begin{array}{c}\Gamma_H(m) = ((a : A^c(\Gamma), T), ((a_i : A_i^c(\Gamma), T_i))_{[n]}) \\ \forall i : [n].\ T_i \preceq_{\mathscr{J}} T_{A_i} \wedge Wf_{A_i}(v_\Gamma(a_i))\end{array}}{\Gamma_H \models @\top\ T(@a_1\ T_1\ p_1, \ldots, @a_n\ T_n\ p_n)\ \checkmark}\ \text{wt-constr-def}$$

Any program construct that is not an assignment, a variable declaration, a variable definition, a return statement, a method/constructor call, or a method/-constructor definition, is always well-typed.

We now define what it means for this type system to be sound. We demand that every expression that has a certain property type actually respect that type's property.

From this, it follows directly that a program $\mathfrak{pr}$ that is well-typed with respect to every hierarchy in a cover $C : \mathscr{C}(\mathfrak{pr})$ is correct.

Theorem 3.1 (Soundness of the type system)**.** Let $\mathfrak{pr} : \mathfrak{WT}(H), H : \mathscr{H}(\mathfrak{pr})$. Then for every context Γ and every expression e, it follows from $\Gamma \models e : (A(p), T)$ that $\forall \varsigma : S(\Gamma).\ Prop_A(v_{\Gamma,\varsigma}(e), p)$.

Proof:

This could be proved by rule induction over the rules of an operational semantics. Since defining an operational semantics for a Java-like language is out of the scope of this thesis, and such a proof would not be very helpful in understanding the properties of this type system, we will forgo it.

3.4.2. Translating property types to JML

In this section, we define a translation from programs that contain property annotations to programs that contain JML specifications. We then use this translation to define the

notion of *JML-correctness*. A program is JML-correct if its translation respects all of its JML specifications.

For now, we ignore any information that could be gleaned from the type rules presented in Section 3.4.1. We explain in Section 3.4.4 how we can use this information to make the translated JML specifications easier to prove.

But before moving on to the translation itself, we define *assert sequences*. An assert sequence for an expression is a sequence of JML assertions which contains one assertion for every type hierarchy in a cover. We will use this sequences as a part of the translation to avoid having to write them out every time we use them.

Definition 3.19 (Assert sequences)**.** Let $\mathfrak{pr} : \mathfrak{Normal}$, $C = (H_i)_{[n]} : \mathscr{C}(\mathfrak{pr})$. Let $t = ((a_i : A_i)_{[n]}, T)$ such that $\forall i : [n].A_i : H_i$. Let e be an expression such that $\forall i : [n].\Gamma_{H_i} \models e : (a_i : A_i, T)$.

We then define the *assert sequence* $[\![t, e]\!]_{\text{assert}}$ to be the following program fragment,

```
1 //@ assert T <: T_{A_1} && Wf_{A_1}(a_1) && Prop_{A_1}(e, a_1);
2 ⋮
3 //@ assert T <: T_{A_n} && Wf_{A_n}(a_n) && Prop_{A_n}(e, a_n);
```

and the *assume sequence* $[\![t, e]\!]_{\text{assume}}$ to be the following program fragment:

```
1 //@ assume T <: T_{A_1} && Wf_{A_1}(a_1) && Prop_{A_1}(e, a_1);
2 ⋮
3 //@ assume T <: T_{A_n} && Wf_{A_n}(a_n) && Prop_{A_n}(e, a_n);
```

We also define the *requires sequence* $[\![t, e]\!]_{\text{requires}}$ to be the following program fragment,

```
@ requires T <: T_{A_1} && Wf_{A_1}(a_1) && Prop_{A_1}(e, a_1);
⋮
@ requires T <: T_{A_n} && Wf_{A_n}(a_n) && Prop_{A_n}(e, a_n);
```

and the *requires-free sequence* $[\![t, e]\!]_{\text{requires\_free}}$ to be the following program fragment:

```
@ requires_free T <: T_{A_1} && Wf_{A_1}(a_1) && Prop_{A_1}(e, a_1);
⋮
```

```
@ requires_free T <: T_{A_n} && Wf_{A_n}(a_n) && Prop_{A_n}(e, a_n);
```

Furthermore, we define the *ensures sequence* $[\![t, e]\!]_{\text{ensures}}$ and the *ensures-free sequence* $[\![t, e]\!]_{\text{ensures\_free}}$ analogously.

The assertions contained in an assert sequence $[\![t, e]\!]_{\text{assert}}$ fully characterize the property types of the expression e, as illustrated by the following example:

Example 3.5 (Assert sequences)**.** Consider the following variable definition from a program $\mathfrak{pr} : \mathfrak{Normal}$ which has a cover $C = (H_1, H_2) : \mathscr{C}(\mathfrak{pr})$ such that

1. H_1 is a property type hierarchy containing the property annotation types `@Even` and `@Odd`, such that the type `@Even int` contains all even integers and the type `@Odd int` contains all odd integers.
2. H_2 is a property type hierarchy containing the property annotation type `@Interval`, such that the type `@Interval(min="n", max="m") int` contains all integers in the interval $[n, m]$.

```
1 @Interval(min="1", max="5") @Odd int var = 3;
```

The assert sequence $[\![(\texttt{Odd}, \texttt{Interval}(1, 5)), \texttt{var}]\!]_{\text{assert}}$ contains the two assertions

```
1 //@ assert \typeof(var) <: T_{Odd} && Wf_{Odd} && Prop_{Odd}(var);
2 //@ assert \typeof(var) <: T_{Interval}
3 //@     && Wf_{Interval}(1, 5) && Prop_{Interval}(var, 1, 5);
```

which (with the right definitions for `Odd` and `Interval`) are equivalent to

```
1 //@ assert \typeof(var) <: int && true && var % 2 != 0;
2 //@ assert \typeof(var) <: int && 5 >= 1 && 1 <= var && var <= 5;
```

Using these assert sequences, we can now define the translation to JML.

Algorithm 3.1 (JML translation). Given a program $\mathfrak{pr} : \mathfrak{Normal}$ and a cover $C = (H_i)_{[n]} : \mathscr{C}(p)$, we construct the program $\mathcal{JML}(\mathfrak{pr}, C)$ as follows.

For brevity's sake, we say that a variable has the type $((a_i : A_i)_{[n]}, T)$ if it has the type T and is annotated with $(a_i : A_i)_{[n]}$.

Assignments

Input: An assignment to a local variable, parameter, or field of the form v `=` e`;` where v has the type $((a_i : A_i)_{[n]}, T)$.

Output: If the assignment occurs in the scope in which v was defined, and v is not a field of another object:

```
1 T temp = e;
2 [[((a_i : A_i)_[n], T), temp]]_assert
3 v = temp;
```

If this is not the case, some identifiers in the annotation arguments may refer to different objects, and the objects they originally referred to may not be accessible. In this case, we cannot check whether the assignment is correct, so we translate it to:

```
1 T temp = e;
2 //@ assert false;
3 v = temp;
```

The `assert false` ensures that the correctness check for this program does not succeed.

Definitions

Input: A definition of a local variable of the form $(((a_i : A_i)_{[n]}, T)$ v `=` e`;`. Field definitions do not occur in our Java fragment because fields must not be initialized outside of constructors.

Output:

```
1 T temp = e;
2 [[((a_i : A_i)_[n], T), temp]]_assert
3 T v = temp;
```

Declarations

Input: A declaration of a local variable or field of the form $(((a_i : A_i)_{[n]}, T)\ v;$
Output: $T\ v;$ for a local variable; `/*@nullable@*/` $T\ v;$ for a field.

Return statements in methods

Input: A return statement returning the expression e in a method m with return type $((a_i : A_i)_{[n]}, T)$.
Output:

```
1 T temp = e;
2 [[((a_i : A_i)_[n], T), temp]]_assert
3 return temp;
```

Return statements in constructors

Input: A return statement in a constructor.
Output: The same return statement, unmodified.

For return statements in methods, Section 3.4.4 shows that we can turn some assertions into assumptions if we know that the return statement is well-typed, thus making the JML proof obligation easier.

However, since our type system does not support annotations on constructors, we deal with them completely on the JML side. It is thus easier to write the constructor's post-condition once, in an `ensures` clause, instead of repeating it for every return statement.

Methods

Input: A method m in a class C with return type $((a_i : A_i)_{[n]}, T)$ and parameter types $((a_i^j : A_i^j)_{i:[n]}, T^j)$ for parameters $(p_j)_{[m]}$.

If m is neither static, a helper method, or a constructor, let $(f_k)_{k:[\mu]}$ be all non-static fields in C, and let $((b_i^k : B_i^k)_{i:[n]}, S^k)$ be their types. Otherwise, let $(f_k)_{k:[\mu]}$ be the empty tuple ().

Output: We add a JML contract with the properties of the fields as free pre-conditions, the properties of the parameters as non-free pre-conditions, and the properties of the return type as free post-conditions.

The properties of the fields can be free because they are established whenever a field is assigned. The properties of the return type can be free because they are established for every return statement, as seen above. We also add a *trampoline method* whose purpose becomes clear in Section 3.4.4.

```
1  /*@ public behavior
2    @ diverges true;
3    @ [[(b_i^k : B_i^k)_{i:[n]}, S_k), f_k]]_requires_free  for every k : [μ]
4    @ [[(a_i^j : A_i^j)_{i:[n]}, T_j), p_j]]_requires  for every j : [m]
5    @ [[(a_i : A_i)_[n]], T), \result]]_ensures_free
6    @*/
7  T m (/*@nullable@*/ T_2 p_2, ..., /*@nullable@*/ T_m p_m) {
8  // or, if m is static:
9  static T m (/*@nullable@*/ T_1 p_1, ..., /*@nullable@*/ T_m p_m) {
10     Translation of method body
11 }
12
13 /*@ public behavior
14   @ diverges true;
15   @ requires b_i^j || (T_j <: T_{A_i^j} && Wf_{A_i^j}(a_i^j) && Prop_{A_i^j}(p_j, a_i^j));
16   @          for every (i, j) : [n] × [m]
17   @ requires_free !b_i^j || (T_j <: T_{A_i^j} && Wf_{A_i^j}(a_i^j) && Prop_{A_i^j}(p_j, a_i^j));
18   @          for every (i, j) : [n] × [m]
19   @ ensures_free T <: T_{A_i} && Wf_{A_i}(a_i) && Prop_{A_i}(\result, a_i);
20   @          for every i : [n]
21   @*/
22 T _m_trampoline(/*@nullable@*/ T_2 p_2, ..., /*@nullable@*/ T_m p_m,
23 // or, if m is static:
24 static T _m_trampoline(/*@nullable@*/ T_1 p_1, ..., /*@nullable@*/ T_m p_m,
25         boolean b_0^0, ..., boolean b_n^m) {
26     return m(p_1, ..., p_m);
27 }
```

Constructors

Input: A constructor in a class C which is annotated with $(a_i : A_i)_{[n]}$ and has parameter types $((a_i^j : A_i^j)_{i:[n]}, T^j)$ for parameters $(p_j)_{[m]}$.

Output:

```
/*@ public behavior
  @ diverges true;
  @ [[((a_i^j : A_i^j)_{i:[n]}, T_j), p_j]]_requires  for every j : [m]
  @ [[((a_i : A_i)_[n], C), this]]_ensures
  @*/
static C (/*@nullable@*/ T_1 p_1, ..., /*@nullable@*/ T_m p_m) {
    Translation of constructor body
}

/*@ public behavior
  @ diverges true;
  @ requires b_i^j || (T_j <: T_{A_i^j} && Wf_{A_i^j}(a_i^j) && Prop_{A_i^j}(p_j, a_i^j));
  @          for every (i, j) : [n] × [m]
  @ requires_free !b_i^j || (T_j <: T_{A_i^j} && Wf_{A_i^j}(a_i^j) && Prop_{A_i^j}(p_j, a_i^j));
  @          for every (i, j) : [n] × [m]
  @ ensures C <: T_{A_i} && Wf_{A_i}(a_i) && Prop_{A_i}(this, a_i);
  @          for every i : [n]
  @*/
static C _C_trampoline(/*@nullable@*/ T_1 p_1, ..., /*@nullable@*/ T_m p_m,
        boolean b_0^0, ..., boolean b_n^m) {
    return new C(p_1, ..., p_m);
}
```

Method calls

Input: A method call to m.

Output: A call to `_m_trampoline` with the additional boolean parameters all set to `false`.

Constructor calls

Input: A constructor call for class C.

Output: A call to `_C_trampoline` with the additional boolean parameters all set to `false`.

Using this translation, we define a program to be JML-correct if the translated program respect all of its JML specifications.

Definition 3.20 (JML-correctness)**.** We define the set $\mathfrak{JMLCorrect}$ of *JML-correct programs* as the set of all programs $\mathfrak{pr}$ such that

1. $\mathfrak{pr} : \mathfrak{Normal}$.
2. for every cover $C : \mathscr{C}(\mathfrak{pr})$, every method in $\mathcal{JML}(\mathfrak{pr}, C)$ is JML-correct.

A method m is JML-correct if and only if when executing m in any arbitrary program state $\varsigma : S(\Gamma(m))$ such that all of m's precondition's evaluate to `true`, either

1. all assertions in m succeed,
2. all non-free pre-conditions of methods called by m hold, and
3. if m terminates, its non-free post-conditions hold

or, if there is a failing assertion, a failing pre-condition of a called method, or a failing post-condition, there is a failing assumption before it.

One problem with this definition is that it seems like we must iterate through every possible cover $C : \mathscr{C}(\mathfrak{pr})$ to verify that a program $\mathfrak{pr}$ is JML-correct. The following theorem explains why this is not actually case, and why it suffices to show JML-correctness using a single cover $C : \mathscr{C}(\mathfrak{pr})$.

Having proven this, we are justified in using the notation $\mathcal{JML}(\mathfrak{pr})$ to refer to an arbitrary translation of the form $\mathcal{JML}(\mathfrak{pr}, C)$.

Theorem 3.2 (Independence of the translation from the cover)**.** Let $\mathfrak{pr} : \mathfrak{Normal}$, $C : \mathscr{C}(p)$.

Then the following are equivalent:

1. there exists a $C : \mathscr{C}(p)$ such that in the program $\mathcal{JML}(\mathfrak{pr}, C)$, every method is correct.
2. for all $C : \mathscr{C}(p)$, every method in the program $\mathcal{JML}(\mathfrak{pr}, C)$ is correct.

Proof:

Observe that with all boolean parameters b_i^j being false, the definition of a trampoline method is equivalent to (i.e., always leads to the same program state as) the following:

```
1  /*@ public behavior
2    @ diverges true;
3    [[((a_i^j : A_i^j)_{i:[n]}, T_j), p_j]]_requires  for every j : [m]
4    [[((a_i : A_i)_[n], T), \result]]_ensures_free
5    @*/
6  T _m_trampoline(/*@nullable@*/ T_2 p_2, ..., /*@nullable@*/ T_m p_m,
7  // or, if m is static:
8  static T _m_trampoline(/*@nullable@*/ T_1 p_1, ..., /*@nullable@*/ T_m p_m,
9           boolean b_0^0, ..., boolean b_n^m) {
10     return m(p_1,...,p_m);
11 }
```

We can thus assume that every JML clause (except for the `diverges` clauses) is part of a [[]] sequence.

Per definition, every such sequence contains exactly one statement for every hierarchy in the cover. Also, every local variable, field, parameter, method return type, and constructor must be annotated with exactly one annotation from every hierarchy in the cover.

This means that – ignoring the order of the sequence – the algorithm's output is independent of the chosen cover.

Since JML clauses have no side-effects, changing the order within a sequence does not change the correctness of the program.

Thus, the correctness of the program output by the algorithm is independent of the chosen cover.

■

Lastly, we prove a soundness property for this translation: Every JML-correct program is correct according to Definition 3.16.

Theorem 3.3 (Soundness of the translation). $\mathfrak{JMLCorrect} \subseteq \mathfrak{Correct}$.

Proof

Let $a : A$ be a property annotation.

Unless $Prop_A$ = `"true"`, every variable *var* annotated with *a* must be immutable. This means that var can only be modified by assigning it and – if v is a method parameter – calling the respective method.

Furthermore, the assertions which Algorithm 3.1 adds to all assignments have the same evaluation in Γ that they do in Γ^{var} – because if Γ and Γ^{var} differ, Algorithm 3.1 just adds trivially false assertions. JML method pre-conditions are always evaluated in the context in which the method was defined.

This means that the assertions which Algorithm 3.1 adds to all assignments and the pre-conditions it adds to all methods can only succeed when

$$T \preceq T_A \wedge Wf_A(v_{\Gamma^{var},\varsigma}(a)) \wedge Prop_A(var, v_{\Gamma^{var},\varsigma}(a))$$

holds for all Γ, ς.

For methods, the assertions which Algorithm 3.1 adds to all return statements similarly ensure that

$$T \preceq T_A \wedge Wf_A(v_{\Gamma^{m},\varsigma}(a)) \wedge Prop_A(r, v_{\Gamma^{m},\varsigma}(a))$$

always holds.

For constructors, the `ensures` statements do the same.

■

3.4.3. Examples

In this interlude sub-section, we take a look at some examples to clarify the difference between JML-correct and well-typed programs.

```
1 public @NonNull Object neverReturnsNull(@Nullable Object x) {
2     boolean b = x == null;
3     if (!b) {
4         return x;
5     } else {
6         return new Object();
7     }
8 }
```

Listing 3.1: Not well-typed but JML-correct

Firstly, it is not surprising that some JML-correct programs are not well-typed. That is, after all, the whole reason why we want to combine our type checker with a JML verification tool like KeY.

```
/*@ public behavior
  @ diverges true;
  @ ensures_free typeof(\result) <: Object
  @           && true && \result != null;
  @*/
public Object neverReturnsNull(/*@nullable@*/ Object x) {
    boolean b = x == null;
    if (!b) {
        Object temp0 = x;
        //@ assert \typeof(temp0) <: Object
        //@     && true && temp0 != null;
        return temp0;
    } else {
        Object temp1 = new Object;
        //@ assert \typeof(temp1) <: Object
        //@     && true && temp1 != null;
        return temp1;
    }
}
```

Listing 3.2: Not well-typed but JML-correct, translation

Listing 3.1 repeats an example from Section 2.2.2. The function `neverReturnsNull()` does indeed never return `null`, but it is not well-typed.

However, if we take a look at its JML translation, seen in Listing 3.2, we can easily see that it is JML-correct, since both of the assertions always hold.

What is not so easily seen is that there are well-typed programs that are not JML-correct.

The program in Listing 3.3 is evidently well-typed. However, looking at the translation of `foo()` in Listing 3.4, we see that the assertion cannot be proven because there is no pre-condition that states that `c.field != null`.

Theoretically, we could add pre-conditions or assumptions for all properties of all visible variables whenever they are necessary, but in a real program that would be completely impractical.

To make this program JML-correct, we could either choose to access `c.field` via a getter of type `@NonNull Object` – in that case, the getter's post-condition would contain the property `c.field != null` – or we could manually add a clause `requires c.field != null` to `foo`'s contract.

In Section 4.4, we present a way to enrich the JML specifications generated by the translation algorithm with our own specifications.

```
1  public class C {
2      public @NonNull Object field;
3      public C() { field = new Object(); }
4  }
5
6  public class D {
7      public void foo(@NonNull C c) {
8          @NonNull Object x = c.field;
9      }
10 }
```

Listing 3.3: Well-typed but not JML-correct

```
1  /*@ public behavior
2    @ diverges true;
3    @ requires typeof(c) <: Object
4    @       && true && c != null;
5    @*/
6  public void foo(C c) {
7      Object x;
8      Object temp0 = c.field;
9      //@ assert typeof(temp0) <: Object
10     //@     && true && temp0 != null;
11     x = temp;
12 }
```

Listing 3.4: Well-typed but not JML-correct, translation

3.4.4. Combining well-typedness and JML-correctness

The goal of this thesis is to combine a type checker with a JML verification tool, but so far we have only defined two ways to check a program's correctness – one using a type checker, and one using a JML verification tool – without any connection between the two.

In this section, we define a modified JML translation algorithm which uses knowledge gained from a well-typedness check to simplify the JML specification.

> **Algorithm 3.2** (Modified JML translation)**. Input:** A program $\mathfrak{pr} : \mathfrak{Normal}$ and a cover $C = (H_i)_{[n]} : \mathscr{C}(\mathfrak{pr})$.

Output: A valid JML-annotated Java program $\mathcal{JML}^*(\mathfrak{pr})$ without any property annotations.

Algorithm:
Run Algorithm 3.1 with the following modifications:

1. When translating a variable assignment or definition that is well-typed for H_i in the context in which it appears, replace the ith `assert` in the sequence by `assume`, unless the ith `assert` clause is `assert false`, in which case leave it as is.

2. When translating a method return statement that is well-typed for H_i in the context in which it appears, replace the ith `assert` in the sequence by `assume`.

3. When translating a method or constructor call, for every actual parameter p_j that for H_i has the same property type as the corresponding formal parameter, make b_j^i be `true` instead of `false`.

How this modified algorithm works is made clearer by two examples.

Example 3.6 (Translation of an assignment)**.** Consider the following program fragment:

```
1 public static void foo(
2         @Length(min="2", max="2") List arg0) {
3     @Length(min="1", max="3") List l0 = arg0;
4 }
```

If we assume that $\texttt{Length}(a, b) \preceq \texttt{Length}(c, d)$ if and only if the interval $[a, b]$ is contained in the interval $[c, d]$, then the definition of `l0` is well-typed because $[2, 2]$ is contained in $[1, 3]$.

Algorithm 3.1 would translate this definition as follows:

```
1 List temp = arg0;
2 //@ assert List <: List && 0 <= 1 && 1 <= 3
3 //@      && 1 <= temp.size() && temp.size() <= 3;
4 List l0 = temp;
```

However, Algorithm 3.2, knowing that the definition is well-typed, would translate it as follows instead:

```
List temp = arg0;
//@ assume List <: List && 0 <= 1 && 1 <= 3
//@      && 1 <= temp.size() && temp.size() <= 3;
List l0 = temp;
```

This reduces the size of the proofs KeY has to find, and also allows KeY to use knowledge gained by the type checker in its proof.

Example 3.7 (Translation of a method call)**.** Consider also the following program fragment:

```
public static void foo(
        @Length(min="1", max="3") List arg0) { }

public static void bar(
        @Length(min="2", max="2") List arg0) {
    foo(arg0);
}
```

If we again assume that $\texttt{Length}(a, b) \preceq \texttt{Length}(c, d)$ if and only if the interval $[a, b]$ is contained in the interval $[c, d]$, the call to `bar` is well-typed.

Algorithm 3.1 would translate this method call as follows:

```
_foo_trampoline(arg0, false);
```

with `_foo_trampoline` defined like this

```
/*@ public behavior
  @ diverges true;
  @ requires arg0WellTyped || List <: List && 0 <= 1 && 1 <= 3 && 1 <=
    temp.size() && temp.size() <= 3;
  @ requires_free !arg0WellTyped || List <: List && 0 <= 1 && 1 <= 3 &&
    1 <= temp.size() && temp.size() <= 3;
  @*/
public static void _foo_trampoline(List arg0, boolean arg0WellTyped) {
```

```
7       return foo(arg0);
8   }
```

Thus, with `arg0WellTyped` set to false, we would need to prove that `List <: List && 0 <= 1 && 1 <= 3 && 1 <= temp.size() && temp.size() <= 3` holds in `bar` when `foo` is being called.

Algorithm 3.2, knowing that the actual parameter has the correct type, would translate the method call as follows:

```
1   _foo_trampoline(arg0, true);
```

Now, with `arg0WellTyped` set to true, the `requires` clause in the trampoline's contract becomes trivially true.

Since the `requires_free` clause now ensures that `foo`'s pre-condition holds, the call to `foo` in `_foo_trampoline` is still legal, meaning that the correctness of the JML specification has not been compromised, but we no longer need to prove `foo`'s pre-condition in `bar`.

Last but not least, we take a look at an example that demonstrates how non-constant property annotations are translated to JML.

Example 3.8 (Translation of non-constant annotations)**.** Non-constant property annotations are translated just like constant property annotations. The only difference is that – because they are not supported by the type system – their properties can never appear in assumptions (or free pre-/post-conditions), but only in assertions (or non-free pre-/post-conditions).

Furthermore, if a field annotation appears in a context other than the one they were defined in, the assertion is turned to `assert false` to ensure that no expression is evaluated in the wrong context. For annotations on method parameters or method return types, this is not necessary, since JML method contracts are always evaluated in the context of the method.

```
1   public @Length(min="a+c", max="b+d") List
2   concat(int a, int b, int c, int d,
3          @Length(min="a", max="b") List l1,
4          @Length(min="c", max="d") List l2) {
5       ...
6       return result;
7   }
```

```
8
9  public void foo(
10         @Length(min="1", max="1") List l1,
11         @Length(min="2", max="2") List l2) {
12     @Length(min="3", max="3") List l3 = concat(l0, l1);
13 }
```

Listing 3.5: Untranslated non-constant annotation

To illustrate this, we use the program shown in Listing 3.5, which contains a function `concat` that concatenates two lists of any length. We assume that the parameters `a`, `b`, `c`, `d` are implicitly annotated with the top annotation.

In the Checker Framework, one would usually avoid using parameter names in annotations and instead identify a parameter by its number. For example, `#1` would refer to `a`. This is because, in a compiled program, there is no way to know which name refers to which parameter. Since we need to have access to the source code of the program we want to check to be able to use JML and KeY, we assume that we do. So, we always use parameter names instead of the numerical identifiers that would be preferred by the Checker Framework.

```
1  /*@ public behavior
2    @ diverges true;
3    @ requires typeof(l1) <: List
4    @      && 0 <= a && a <= b
5    @      && a <= l1.size() && l1.size() <= b;
6    @ requires typeof(l2) <: List
7    @      && 0 <= c && c <= d
8    @      && c <= l2.size() && l2.size() <= d;
9    @ ensures_free typeof(\result) <: List
10   @      && 0 <= a + c && a + c <= b + d
11   @      && a + c <= \result.size() && \result.size() <= b + d;
12   @*/
13 public List concat(
14         int a, int b, int c, int d,
15         /*@nullable@*/ List l1, /*@nullable@*/ List l2) {
16     ...
17     List temp = result;
18     //@ assert typeof(temp) <: List
19     //@      && 0 <= a + c && a + c <= b + d
20     //@      && a + c <= temp.size() && temp.size() <= b + d;
21     return temp;
22 }
```

Listing 3.6: Translated non-constant annotation, I

The function `concat` is translated as expected (see Listing 3.6) with one `requires` clause for every parameter, one `ensures_free` clause for the result, and one `assert` clause before the `return` statement.

```
/*@ public behavior
  @ diverges true;
  @ requires l1WellTyped || (typeof(l1) <: List
  @     && 0 <= a && a <= b
  @     && a <= l1.size() && l1.size() <= b);
  @ requires_free !l1WellTyped || (typeof(l1) <: List
  @     && 0 <= a && a <= b
  @     && a <= l1.size() && l1.size() <= b);
  @ requires l2WellTyped || (typeof(l2) <: List
  @     && 0 <= c && c <= d
  @     && c <= l2.size() && l2.size() <= d);
  @ requires_free !l2WellTyped || (typeof(l2) <: List
  @     && 0 <= c && c <= d
  @     && c <= l2.size() && l2.size() <= d);
  @ ensures_free typeof(\result) <: List
  @     && 0 <= a + c && a + c <= b + d
  @     && a + c <= \result.size() && \result.size() <= b + d;
  @*/
public List _concat_trampoline(
        int a, int b, int c, int d,
        /*@nullable@*/ List l1, /*@nullable@*/ List l2,
        boolean l1WellTyped, boolean l2WellTyped) {
    return concat(a, b, c, d, l1, l2);
}
```

Listing 3.7: Translated non-constant annotation, II

The trampoline `_concat_trampoline` is shown in Listing 3.7. Theoretically, this trampoline would also need parameters `aWellTyped`, `bWellTyped`, etc., and appropriate pre-conditions, but as the parameters `a`, `b`, `c`, `d` are implicitly annotated with the top annotation, all of the pre-conditions would be trivial, and we thus leave them out.

```
/*@ public behavior
  @ diverges true;
  @ requires typeof(l1) <: List
  @     && 0 <= 1 && 1 <= 1
```

```
5    @     && 1 <= l1.size() && l1.size() <= 1;
6    @ requires typeof(l2) <: List
7    @     && 0 <= 2 && 2 <= 2
8    @     && 2 <= l2.size() && l2.size() <= 2;
9    @*/
10 public void foo(/*@nullable@*/ List l1, /*@nullable@*/ List l2) {
11     List temp = _concat_trampoline(
12         1, 1, 2, 2,
13         l0, l1,
14         false, false);
15     //@ assert typeof(temp) <: List
16     //@     && 0 <= 3 && 3 <= 3
17     //@     && 3 <= temp.size() && temp.size() <= 3;
18     List l3 = temp;
19 }
```

Listing 3.8: Translated non-constant annotation, III

The function `foo` which calls `concat` is shown in Listing 3.8 and again looks as expected. An assertion is added for the assignment because the type system does not know that `concat`'s return type `@List(min="a+c", max="b+d") List` and the type `@List(min="3", max="3") List` are equivalent.

Using this modified translation, we define the notion of *JML*-correctness*.

Definition 3.21 (JML*-correctness)**.** We define the set $\mathfrak{JMLCorrect}^*$ of *JML*-correct programs* as the set of all programs $\mathfrak{pr}$ such that

1. $\mathfrak{pr} : \mathfrak{Normal}$.
2. for every cover $C : \mathscr{C}(\mathfrak{pr})$, every method in $\mathcal{JML}^*(\mathfrak{pr}, C)$ is JML-correct.

We now state our final theorem from which it follows that $\mathfrak{JMLCorrect}^* \subseteq \mathfrak{Correct}$. This means that if a program's translation according to Algorithm 3.2 respects all of its JML specifications, then that program is correct.

This is the central theorem of this thesis, because it tells us that combining a type checker with a JML verification tools – as Algorithm 3.2 does – never leads to an incorrect program being accepted.

Theorem 3.4 (Soundness of the modified translation)**.** $\mathfrak{JMLCorrect}^* \subseteq \mathfrak{JMLCorrect}$.

Proof

Firstly, we must show that Algorithm 3.2 only replaces assertions by assumptions (or `requires` clauses by `requires_free` clauses) if they always hold.

When a variable assignment or definition is well-typed for a hierarchy H in the context Γ in which it appears, then $\Gamma_H \models e : (v_{\Gamma^{var}}(a), T)$, where e is the expression being assigned and $a : A : H$ is the annotation the variable *var* is annotated with.

Because the type system is sound, this means that

$$T \preceq T_A \wedge Wf_A(v_{\Gamma^{var}}(a)) \wedge Prop_A(v_{\Gamma,\varsigma}(e), v_{\Gamma^{var}}(a))$$

holds for all $\varsigma : S(\Gamma)$.

It makes no difference whether we evaluate a in Γ^{var} or Γ. If it did, Algorithm 3.1 would have turned the assertion into `assert false`, and Algorithm 3.2 would not have replaced it. Thus

$$T \preceq T_A \wedge Wf_A(v_{\Gamma}(a)) \wedge Prop_A(v_{\Gamma,\varsigma}(e), v_{\Gamma}(a))$$

holds, from which it follows that the assertion in question always holds.

This can be proven in the same way for method return statements.

Secondly, we must show Algorithm 3.2 only makes a boolean trampoline parameter b^i_j `true` if the associated condition

$$T_j <: T_{A^j_i} \wedge Wf_{A^j_i}(a^j_i) \wedge Prop_{A^j_i}(p_j, a^j_i)$$

always holds.

This can be shown analogously to two case above: If $\Gamma_H \models p : (v_{\Gamma^m}(a), T)$ for an actual parameter $p : T$ whose corresponding formal parameter is annotated with $a : A : H$, it follows from the soundness of the type system that

$$T \preceq T_A \wedge Wf_A(v_{\Gamma^m}(a)) \wedge Prop_A(v_{\Gamma,\varsigma}(p), v_{\Gamma^m}(a))$$

■

4. The type-checking pipeline

In this chapter, we describe a pipeline consisting of a type checker developed using the Checker Framework, which checks whether a Java program is well-typed, and KeY, which is used to check if a program is JML*-correct.

Section 4.1 gives an overview over the pipeline, while the sections after it go into more detail about specific components.

4.1. The pipeline

The activity diagram in Figure 4.1 visualizes the pipeline's mode of operation.

In the following paragraphs, we briefly describe every part of this diagram before going into more detail about individual parts in the later sections.

Input: The user writes a **program** which is annotated with property annotations. These annotations form one or multiple **property annotation lattices**, which are defined in lattice files. The syntax and semantics of these lattice files is explained in Section 4.2. Furthermore, the program may contain some **JML specifications** in addition to the property annotations.

Prove lattice properties: This component proves that the partial orders defined by the user in the markup language described in Section 4.2 actually form bounded lattices. The proof is described in Section 4.2.2. The component also proves that all property annotation types that appear in the lattices are valid and that they form a cover for the program.

Prove program normality: This component proves that the program is normal, i.e., that it is well-typed in the immutability and the initialization type systems, and that every property annotation in the program is valid.

Type checker: This component proves that the program is well-typed. The well-typedness information is given to the translator, along with any existing JML specifications. We saw a theoretical overview over the type system in Section 3.4.1. The implementation of the type checker is explained in Section 4.3.

Property-annotation-to-JML translator: This component translates all property annotations to JML clauses, as described in Algorithm 3.2, keeping any JML specifications that were already in the program. This is described in more detail in Section 4.4.

KeY: KeY proves the correctness of all JML specifications. This includes the specifications output by the translator, as well as those that were in the program to begin with. We already saw an overview over KeY in Section 2.3. Section 4.5 explains how KeY is used as a part of this pipeline.

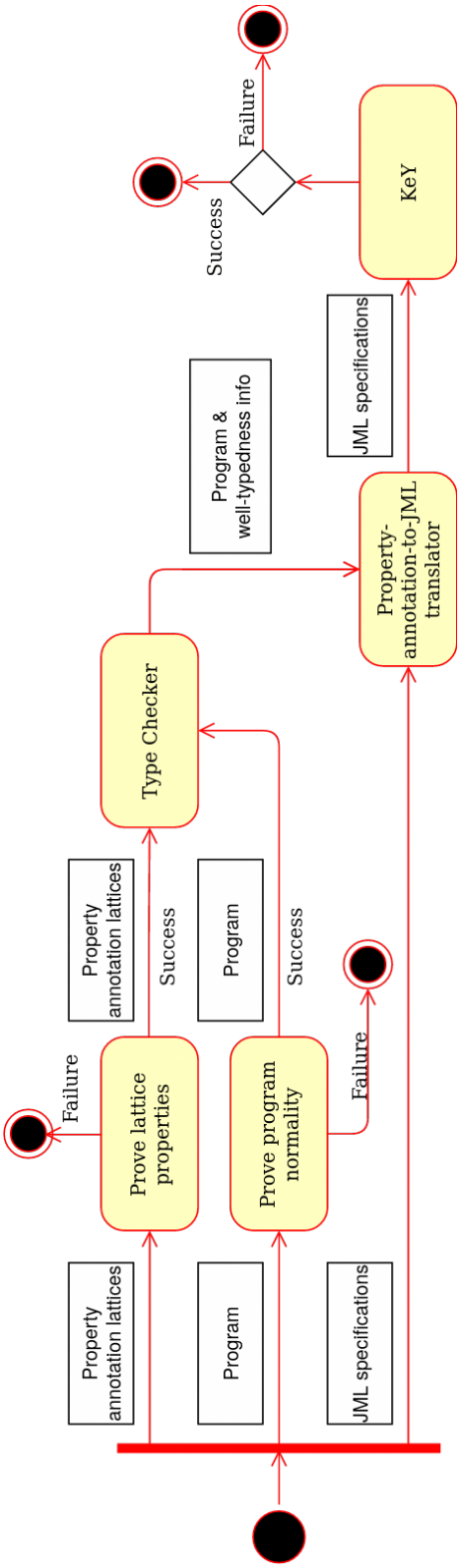

Figure 4.1.: Type-checking pipeline

4.2. A language for property annotation lattices

In this section, we define a language that allows programmers to, firstly, define the property annotation types in their program, and secondly, define a property annotation lattice using these types.

4.2.1. Syntax and semantics

The syntax of this language is defined by the grammar in Listing 4.1. In addition to this grammar, any line starting with a # is a comment, and is ignored.

The parts between question marks are informal specifications. For example, `? Java identifier ?` is a non-terminal symbol that can be evaluated to any valid Java identifier.

For Java expressions, we require identifiers to be delimited by § to make it easier for the checker to substitute expressions for these identifiers without having to parse them.

The statement `ident latticeident` defines `latticeident` as the unique identifier of this lattice. This identifier is used in error messages by the Checker Framework and in variable names and comments in the JML translation (see Algorithm 3.1) to distinguish between multiple lattices.

An annotation type A is defined by the following line:

`annotation` $A(T_{p_1} p_1, \ldots, T_{p_n} p_n)$ T_A `:<==>` $Prop_A$ `for` Wf_A`;`

The most general subject type $\top_{\mathscr{J}}$ can be referred to by the key-word `any`.

The relation $\preceq$ belonging to the lattice is defined by lines like this:

`relation` $A(p_1, \ldots, p_n)$ `<:` $B(q_1, \ldots, q_m)$ `:<==>` $S_{A,B}(p, q)$`;`

This states that for two evaluated property annotations $A(p)$, $B(q)$, $A(p) \preceq B(q)$ holds if and only if the *relation condition* $S_{A,B}(p, q)$ holds.

For any two evaluated property annotations $A(p)$, $B(q)$, $S_{A,B} : \texttt{String}^{p}_{\texttt{boolean}}(p, q)$ must hold, i.e., the value of the relation condition must only depend on the actual parameters of the two annotations.

There must be exactly one `relation` statement for each pair of property annotation types A, B.

> **Definition 4.1** (<?)**.** We write $A(p) <? B(q)$ if there exists a relation statement for the types A, B or if $A(p) = B(q)$.

```
<lattice> ::= <lattice_ident><annotations> <relations> <bounds>
<lattice_ident> ::= "ident" ? Java identifier ? ;

<annotations> ::= <annotation> | <annotation> <annotations>
<annotation> ::= "annotation" <ident> "(" (<params> | "") ")"
                 (<qualified_ident> | "any")
                 ":<==>" <expression> "for" <expression> ";"

<relations> ::= <relation> | <relation> <relations>
<relation> ::= <annotation_inst> "<:" <annotation_inst>
               ":<==>" <expression> ";"

<bounds> ::= <bound> | <bound> <bounds>
<bound> ::= ( "join" | "meet" ) <annotation_inst> "," <annotation_inst>
            ":=" <annotation_expr_inst> [ "for" <expression> ] ";"

<annotation_inst> ::= <ident> "(" (<idents> | "") ")"
<annotation_expr_inst> ::= <ident> "(" (<expressions> | "") ")"

<params> ::= <param> | <param> "," <params>
<param> ::= <qualified_ident> <ident>

<expressions> ::= <expression> | <expression> "," <expressions>
<idents> ::= <ident> | <ident> "," <idents>

<expression> ::= '"' ? Java expression with identifiers delimited by § ? '"'
<qualified_ident> ::= ? qualified Java identifier ?
<ident> ::= ? unqualified Java identifier ?
```

Listing 4.1: Language for lattices

Theoretically, the annotations and relations suffice to define a property annotation lattice. However, to avoid having to compute all possible joins and meets programatically, we force the user to give the joins and meets for each pair of property annotations that are not in a sub-annotation relationship (if $a \preceq b$, then obviously $a \vee b = b$ and $a \wedge b = a$).

$$\texttt{join } A(p_1,\ldots,p_n) \texttt{ , } B(q_1,\ldots,q_m) \texttt{ := } C_j(c_1^j,\ldots,c_m^j) \quad \texttt{for } J^j_{A,B,C_j}(p,q);$$

Here, the c_i^j are not identifiers, but expressions in $\texttt{String}^p(p,q)$.

There can be multiple `join` and `meet` statements for each pair of property annotation types A, B.

The join/meet of two evaluated property annotations $A(p), B(q)$ is decided by iterating through all `join`/`meet` statements for A, B in the order they appear and taking the $C_j(c)$ of the first statement where the *join/meet condition* $J^j_{A,B,C_j}(p,q)$ holds to be the join/meet.

This means that the *overall join/meet condition* for a statement is stronger than $J^j_{A,B,C_j}(p,q)$ because it also includes the negated conditions for all previous statements.

> **Definition 4.2** (Overall join condition)**.** We define the *overall join condition* for the above join statement as
>
> $$J^{j*}_{A,B,C}(p,q) := J^j_{A,B,C_j}(p,q) \wedge \bigwedge_{i=0}^{j-1} \neg J^i_{A,B,C_i}(p,q)$$

For any two evaluated property annotations $A(p), B(q)$, the join/meet condition $J^j_{A,B,C_j} : \texttt{String}^s_{\texttt{boolean}}(p,q)$ must hold.

4.2.2. Proving the lattice properties

Here, we give an overview over how it can be proven that a partial order R given by a description as defined in the previous section is actually a property annotation lattice.

The pipeline component that does this proof has not been implemented, so for now, it must be done manually.

1. All property annotation types appearing in R are valid. For this, it must be proven that all well-formedness conditions are constant expressions, that all properties are pure expressions, and that all subject types are immutable.
2. R is a lattice.
   a) R is reflexive: For every pair $A(p) <? A(q)$:

   $$Wf_A(p) \rightarrow S_{A,A}(p,p)$$

   b) R is transitive: For all triples $A(p) <? B(q) <? C(r)$:

   $$Wf_A(p) \wedge Wf_B(q) \wedge Wf_C(r) \wedge S_{A,B}(p,q) \wedge S_{B,C}(q,r) \rightarrow S_{A,C}(p,r)$$

c) R is anti-symmetric: For all pairs $A(p) <? B(q),\ B(q) <? A(p)$ where $A(p) \neq B(q)$:

$$\begin{aligned} &Wf_A(p) \wedge Wf_B(q) \\ &\quad \rightarrow (S_{A,B}(p,q) \rightarrow \neg S_{B,A}(q,p)) \\ &\quad \wedge (S_{B,A}(q,p) \rightarrow \neg S_{A,B}(p,q)) \end{aligned}$$

d) The joins given in the description are correct: For all join statements with overall join condition $J^*_{A,B,C}(p,q)$ and all $D(u)$ such that $A(p) <? D(u)$ and $B(q) <? D(u)$:

$$\begin{aligned} &Wf_A(p) \wedge Wf_B(q) \wedge Wf_D(u) \wedge J^*_{A,B,C}(p,q) \wedge S_{A,D}(p,u) \wedge S_{B,D}(q,u) \\ &\quad \rightarrow Wf_C(r) \wedge S_{C,D}(r,u) \end{aligned}$$

e) The meets given in the description are correct: This can be proven in the same way as the correctness of the joins.

3. There is a top and a bottom, i.e., annotation types `Top` and `Bottom` such that
   a) $Wf_{\texttt{Top}} = Wf_{\texttt{Bottom}} = \texttt{"true"}$.
   b) $Prop_{\texttt{Top}} = \texttt{"true"}, Prop_{\texttt{Bottom}} = \texttt{"false"}$.
   c) $T_{\texttt{Top}} = T_{\texttt{Bottom}} = \top_{\mathscr{J}}$.
   d) $\forall A(p) : \mathscr{A},\ A(p) \preceq_{\mathscr{A}} \top \wedge \bot \preceq_{\mathscr{A}} A(p)$ where $\top = v(\texttt{Top}), \bot = v(\texttt{Bottom})$.

4. R conforms to the property annotations' properties: For all pairs $A(p) <? B(q)$:

$$\begin{aligned} T_A \preceq T_B \wedge \forall s : T_B,\ &Wf_A(p) \wedge Wf_B(q) \wedge S_{A,B}(p,q) \\ &\rightarrow (Prop_A(s,a) \rightarrow Prop_B(s,b)) \end{aligned}$$

If all of these conditions hold, the partial order R defined by such a description is a property annotation lattice. We can then get a corresponding type hierarchy H by taking the direct product of R and a program $\mathfrak{pr}$'s program hierarchy.

Then we can use the definitions in Section 3.4 to check if $\mathfrak{pr}$ is well-typed with respect to H.

4.2.3. An example lattice

In this section, we explain the partial order shown in Listing 4.2, and prove that it is a property annotation lattice.

Listing 4.2 defines a partial order containing the annotations `UnknownLength`, `BottomLength` and `Length`(min, max) for all $min, max : \mathbb{Z}_{Java}$ where $min \geq 0, min \leq max$.

`UnknownLength()` is a super-annotation of every other annotation, while `BottomLength()` is a sub-annotation of every other annotation.

The last `relation` line defines that

$$\texttt{Length}(a_0, b_0) \leq \texttt{Length}(a_1, b_1) \iff a_0 \geq a_1 \wedge b_0 \leq b_1$$

i.e., if and only if the interval $[a_0, b_0]$ is a sub-interval of $[a_1, b_1]$.

For example, `Length`$(2, 3)$ is a sub-annotation of `Length`$(1, 4)$.

The lines starting with `join` and `meet` give the joins and meets, which are explained in more detail below.

We now prove that this partial order R is a property type lattice:

1. All property annotation types appearing in R are valid. The only non-trivial annotation type in R is `Length`, and we have already seen in Example 3.1 that it is valid – assuming that `List` is immutable and `List.size()` is pure and terminates normally.

2. R is reflexive. The reflexivity of `a0 >= a1 && b0 <= b1` follows directly from the reflexivity of $\leq$ and $\geq$.

3. R is anti-symmetric. This follows directly from the anti-symmetry of $\leq$ and $\geq$.

4. R is transitive. Again, this follows directly from the transitivity of $\leq$ and $\geq$.

5. The joins given in R's description are correct. Let $A_0 := \texttt{Length}(a_0, b_0)$ and $A_1 := \texttt{Length}(a_1, b_1)$ be two well-formed evaluated property annotations. Let $A^* := \texttt{Length}(\alpha, \beta)$ be another well-formed evaluated property annotation such that $A_0 \leq_R A^*$ and $A_1 \leq_R A^*$.

   Then $a_0 \geq \alpha \wedge b_0 \leq \beta$ and $a_1 \geq \alpha \wedge b_1 \leq \beta$ must hold.

   This implies that $\min(a_0, a_1) \geq \alpha \wedge \max(b_0, b_1) \leq \beta$ and thus

   $$\texttt{Length}(\min(a_0, a_1), \max(b_0, b_1)) \leq_R \texttt{Length}(\alpha, \beta)$$

   The well-definedness of $\hat{A} := \texttt{Length}(\min(a_0, a_1), \max(b_0, b_1))$ follows directly from the well-definedness of A_0 and A_1:

   $$(\forall i : [1, 2],\ a_i \geq 0 \wedge a_i \leq b_i) \rightarrow \min(a_0, a_1) \geq 0 \wedge \min(a_0, a_1) \leq \max(b_0, b_1)$$

We have proven that every super-annotation of A_0 and A_1 is also a super-annotation of $\hat{A}$ and that $\hat{A}$ is well-defined, which means that

$$A_0 \vee A_1 = \texttt{Length}(\max(a_0, a_1), \min(b_0, b_1))$$

i.e., the definition of the join was correct.

6. The meets given in R's description are correct. Let $A_0 := \texttt{Length}(a_0, b_0)$ and $A_1 := \texttt{Length}(a_1, b_1)$ be two well-formed evaluated property annotations. Let $A^* := \texttt{Length}(\alpha, \beta)$ be another well-formed evaluated property annotation such that $A^* \preceq_R A_0$ and $A^* \preceq_R A_1$.

   Then $\alpha \geq \max(a_0, a_1) \wedge \beta \leq \min(b_0, b_1)$ must hold.

   In the case where $[a_0, b_0]$ and $[a_1, b_1]$ overlap, we can continue the proof in the same way as the proof for the joins.

   For the case where they do not overlap, i.e., $b_0 < a_1 \wedge b_1 < a_0$, it follows that $\min(b_0, b_1) < \max(a_0, a_1)$, and thus A^* is not well-formed.

   This means that the only well-formed lower bound for A_0 and A_1 is $\bot$.

7. There is a top and a bottom. `UnknownLength` and `BottomLength` obviously conform to the requirements for top and bottom, respectively.

8. R conforms to the property annotations' properties. Observe that

   $$\texttt{Length}(a_0, b_0) \preceq_R \texttt{Length}(a_1, b_1)$$

   if and only if the interval $[a_0, b_0]$ is contained in $[a_1, b_1]$. Thus, any `List` length which is in the interval $[a_0, b_0]$ must also be in the interval $[a_1, b_1]$.

```
ident length;

annotation UnknownLength() any :<==> "true" for "true";
annotation BottomLength() any :<==> "false" for "true";

annotation Length(int min, int max) List
    :<==> "§subject§.size >= §min§ && §subject§.size <= §max§"
    for "§min§ >= 0 && §min§ <= §max§";

relation Length(a0, b0) <: Length(a1, b1)
    :<==> "§a0§ >= §a1§ && §b0§ <= §b1§";

relation Length(a,b) <: UnknownLength() :<==> "true";
relation BottomLength() <: Length(a,b) :<==> "true";
relation BottomLength() <: UnknownLength() :<==> "true";

join Length(a0, b0), Length(a1, b1)
    := Length("java.lang.Math.min(§a0§, §a1§)",
              "java.lang.Math.max(§b0§, §b1§)");

# overlapping
meet Length(a0, b0), Length(a1, b1)
    := Length("java.lang.Math.max(§a0§, §a1§)",
              "java.lang.Math.min(§b0§, §b1§)")
    for "§b0§ >= §a1§ || §b1§ >= §a0§";

# non-overlapping
meet Length(a0, b0), Length(a1, b1) := BottomLength();
    # "!(b0 >= a1 || b1 >= a0)" is implicit!
```

Listing 4.2: Example lattice

4.3. The type checker

The type checker is implemented in the Checker Framework and can be used much like any other checker. [Che20, 2.2] describes how to use a checker.

The property type checker differs in that it requires some additional arguments to specify the lattices to use (which must be written in the language defined in Section 4.2), the package containing the annotation type definitions, and the root directory of the program to be checked.

One unfortunate restriction of the current implementation is that the Java package containing the annotation types must be part of the property type checker itself, and annotation types cannot be loaded from an external source.

Once the Checker Framework and the property type checker have been installed, the property type checker can be run using the command shown in Listing 4.3.

```
javac -processor property \
    -APropertyChecker_lattices=path/to/lattice/one;path/to/lattice/two \
    -APropertyChecker_classes path/to/project/root \
    -APropertyChecker_qual=annotation.type.package \
    FileToCheck.java
```

Listing 4.3: Type checker usage

The type checker implementation instantiates a so-called lattice checker for every lattice file. A lattice checker is responsible for checking whether the program is well-typed with respect to its lattice. All of these lattice checkers run independently of each other.

Lattice checkers override the default qualifier hierarchy – this is what the Checker Framework calls the annotation hierarchy – with a custom implementation that gets its sub-typing information from a parsed lattice file.

For example, to see if two `Length` annotations $\texttt{Length}(a_0, b_0)$ and $\mathit{Length}(a_1, b_1)$ are in a sub-type relationship, it looks for the appropriate `relation` line from the lattice file:

```
relation Length(a0, b0) <: Length(a1, b1)
    :<==> "§a0§ >= §a1§ && §b0§ <= §b1§";
```

Then the parameters of the two annotations are plugged into the condition `§a0§ >= §a1§ && §b0§ <= §b1§` and the resulting expression is evaluated.

Finding the join or meet of two annotations works the same way.

Every lattice checker collects all mal-typed assignments, definitions, etc. After every lattice checker has been run, the parent checker collates all of these results and gives them to the JML translator.

❊

As explained in the previous sections, property annotation lattices are defined in lattice files. However, because Java annotations are just a special kinds of interface, they must still be defined in Java files.

As this is unintuitive, we explain it using the following example.

```
1 @SubtypeOf({UnknownLength.class})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target({ElementType.TYPE_USE})
4 public @interface Length {
5     String min() default "0";
6     String max() default "2147483647";
7 }
```

Listing 4.4: Length.java

Let us use the example lattice from Listing 4.2 again. The annotation type `Length` is defined by the file shown in Listing 4.4 in addition to the information found in the lattice file. This additional definition is necessary because the checker cannot dynamically create new Java types at run-time. Also, most of the meta-annotations defined by the Java API and the Checker Framework are not supported by the lattice language.

It would obviously be both useful and not very difficult to add support for these meta-annotations to our lattice language and to generate the Java files for the annotation types programatically.

We now give an overview over some of the most important meta-annotations and how they should be used with property types.

1. `@SubtypeOf`. These annotations define an approximation of the property annotation lattice. Since `Length` is annotated with `@SubtypeOf({UnknownLength.class})`, every instance of `Length` must be a sub-annotation of every instance of `UnknownLength`.

2. `@Retention(RetentionPolicy.RUNTIME)` specifies that wherever a `Length` annotation is used, it is compiled into the class file and retained at run-time, so that it can be accessed using reflection. `@Retention(RetentionPolicy.CLASS)` – which would compile the annotations but not retain them at run-time – is sufficient for type-checking, but run-time retention has negligible costs and allows for future development of run-time tools. [Che20, 37.7.3]

3. `@Target({ElementType.TYPE_USE})` specifies that this annotation may appear on any type use [Che20, 34.5.1]. As mentioned before, this is further constrained by the property checker, which only supports annotations on variable definitions, method return types, and constructors.

4. `@RelevantJavaTypes({List.class})` [Che20, 34.5.5]. This meta-annotation does not appear because it is redundant with the definition of the subject type in the lattice file. Specifying it anyway does not lead to an error, as long as the subject type is the only parameter.

5. `@DefaultQualifierForHierarchy`, `@DefaultQualifierFor(List.class)`. None of these annotations appear here, but they could be used to make `Length` the default qualifier. `@DefaultQualifierFor(List.class)` would mean that any `List` variable which the user does not give an explicit annotation for would be annotated with `@Length(min="0", max="2147483647")` (using the default values for the parameters). `@DefaultQualifierForHierarchy` would mean that any variable at all which the user does not give an explicit annotation for would be annotated with `@Length(min="0", max="2147483647")`; this would of course lead to a type error, since only lists may be annotated with `Length`. [Che20, 29.5]

4.4. The property-annotation-to-JML translator

The translator is implemented as a part of the type checker. After the type checker has checked the well-typedness of the program separately for every lattice, the well-typedness results for every lattice are given to the translator.

As mentioned in Section 3.4.3 and Section 4.1, the user may specify some JML clauses for their program, which in this stage of the pipeline have to be combined with the JML clauses that arise from the translation of the property annotations.

Look at the example in Listing 4.5, which models part of a web shop.

The translation to JML is shown in Listing 4.6. For the assumption before the assignment to `customers`, we assume that the default constructor `List()` returns a result of type `@Length(min="0", max="0") List`; otherwise the assignment would not be well-typed and we would have an assertion instead of an assumption.

The assumption before the assignment to `order` is missing because `orders` is implicitly annotated with the top annotation `UnknownLength` whose property and well-formedness condition are both equal to `"true"` and whose subject type is $\top_{\mathcal{J}}$, making the assumption trivial and unnecessary.

We also see that the JML clauses which the user has specified for the constructor have been integrated into the contract. The `ensures` clause specifies that the reference returned by the constructor and all fields are fresh and do not refer to variables that already existed before the constructor was invoked. The `assignable` clause specifies

```
1  public class Shop {
2
3      private List orders;
4      private @Length(min="0", max="2147483647") List customers;
5
6      @JMLClause(values={
7          "ensures \fresh(this) && \fresh(this.*)",
8          "assignable \nothing",
9      })
10     public Shop() {
11         orders = new List();
12          customers = new List();
13     }
14 }
```

Listing 4.5: Untranslated example

that the constructor does not modify any heap locations – except for the two fields of the object it is creating.

To make the implementation of the parser easier, we specify JML clauses using the `@JMLClause` annotation instead of using a JML comment as in Listing 4.6. This keeps us from having to parse JML comments.

As shown in Section 3.4.3, such JML clauses may be necessary to be able to show the JML-correctness of a program.

```
1  public class Shop {
2
3      private /*@nullable@*/ List orders;
4
5      private /*@nullable@*/ List customers;
6
7      /*@ public behavior
8        @ diverges true;
9        @ ensures \fresh(this) && \fresh(this.*);
10       @ assignable \nothing;
11       @*/
12     public Shop() {
13         super();
14
15         List temp0 = new List();
16         orders = temp0;
17
18         List temp1 = new List();
19         //@ assume typeof(temp1) <: List
20         //@      && 0 >= 0 &&  0 <= 2147483647
21         //@      && 0 <= temp1.size() && temp1.size() <= 2147483647;
22         customers = temp1;
23     }
24 }
```

Listing 4.6: Translated example

4.5. KeY

Section 2.3, Section 3.4.2, and Section 4.4 already went into much detail about JML, so in this section all that is left is to give an overview over the proof settings used by KeY in this pipeline.

In the program output by the translator, there is one proof obligation for every method.

Since assertions and assumptions are syntactic sugar for block contracts, there is a separate proof obligation for every one of these as well, but since the correctness of most assertions depends on the context of the surrounding method, assertions and assumptions are evaluated during the proof for the surrounding method.

There is also a proof obligation for every trampoline, but since all trampoline post-conditions are either free (i.e., come from `ensures_free` clauses) or copied from the contract of the trampoline's target method, a trampoline's contract is valid by construction and need not be proven.

For now the user must manually load the program output by the translator into KeY and prove all remaining proof obligations manually. In the future, it would obviously be nice to have a fully automatic pipeline that calls the type checker and KeY sequentially and presents the result in a dedicated interface.

5. Case Study

In this chapter, we present a small program that was annotated with property annotations and then validated using the pipeline described in Chapter 4.

Section 5.1 offers a description and explanation of the most interesting parts of this program.

Section 5.2 evaluates the case study with regard to the goals stated in the introduction: allowing the user to define property types with as little specification and verification overhead as possible, combining some of the strengths of type checkers and formal verification tools.

The complete source code for this case study is printed in Appendix A.

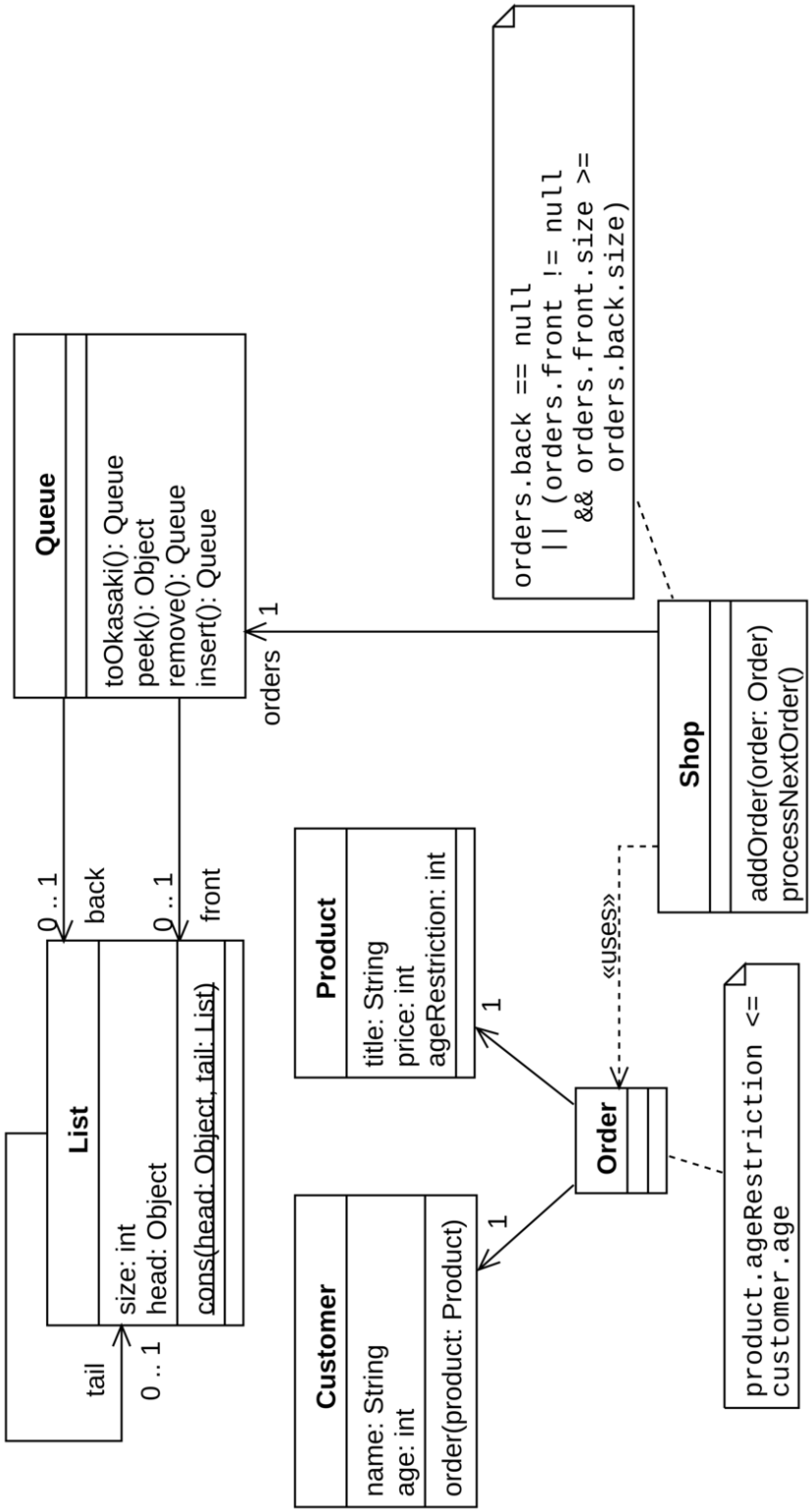

Figure 5.1.: Class diagram for the case study

5.1. Description

The class diagram in Figure 5.1 shows the classes in the case study and their most important fields and methods.

The `Main` class is not shown, as it contains only a `main()` method which calls some methods from the other classes for demonstration purposes.

The case study implements part of a web shop. The `Shop` class contains a FIFO queue of `Orders`, which are pairs consisting of a `Customer` and a `Product`. Here, we see the first property we want to prove: that no customer can order a product that is not approved for their age.

The method `Customer.order()` instantiates a new order, and then calls `Shop.addOrder()` to add the order to the shop's order queue. Here, we see the second property we want to prove: the queue containing the order should satisfy the *Okasaki invariant*.

Okasaki queues, as we will call them, are a kind of FIFO queue first introduced by Chris Okasaki in [Oka95]. Our implementation is based on that in [JSV17, 8].

An Okasaki queue is made up of two lists, called *front* and *back*, such that *front* is never shorter than *back*. To insert an item into the queue, we add it to *back*. To remove an item, we remove it from *front*.

Thus, for the queue containing the items (a, b, c, d), we might have $front = (a, b)$, $back = (d, c)$, or $front = (a, b, c)$, $back = (d)$, but not $front = (a)$, $back = (d, c, b)$ since then *front* would be shorter than *back*.

Re-establishing the Okasaki invariant after every insertion and removal guarantees that those two operations can be done in amortized constant time [Oka95, 4]. This does however assume that we are using lazy lists, which we do not do in this case study for simplicity's sake.

In the rest of this section, we show how property types can be used to verify that the two aforementioned properties are respected.

We also show some other, less interesting properties, e.g., that all attributes that correspond to associations with multiplicity 1 in Figure 5.1 are non-null.

5.1.1. Products and orders

We start with the `Product` class, which is one of the simpler classes in this program.

As we can see in Listing 5.1, every product has a title, a price, which must be a non-negative number, and an age restriction, which must be a number between 0 and 18.

In addition, the `title` is implicitly annotated with the default annotation `@NonNull`, meaning that is must not be equal to null.

```java
public class Product {
    public String title;
    public @Interval(min="0", max="2147483647") int price;
    public @Interval(min="0", max="18") int ageRestriction;

    @JMLClause(values={
        "assignable this.*",
        "ensures \fresh(this) && \fresh(this.*)",
        "ensures this.title == title"
          + " && this.price == price"
          + " && this.ageRestriction == ageRestriction"})
    public @AllowedFor(age="ageRestriction") Product(
            String title,
            @Interval(min="0", max="2147483647") int price,
            @Interval(min="0", max="18") int ageRestriction) {
        this.title = title;
        this.price = price;
        this.ageRestriction = ageRestriction;
    }

    @JMLClause(values={"assignable \nothing"})
    public String getTitle() { return title; }

    @JMLClause(values={"assignable \nothing"})
    public @Interval(min="0", max="2147483647") int getPrice() {
        return price;
    }

    @JMLClause(values={"assignable \nothing"})
    public @Interval(min="0", max="18") int getAgeRestriction() {
        return ageRestriction;
    }
}
```

Listing 5.1: Product.java

```
ident allowedfor;

annotation AllowedForEveryone() any :<==> "true" for "true";
annotation AllowedForNoone() any :<==> "false" for "true";

annotation AllowedFor(int age) case_study.Product
    :<==> "§subject§ != null && §subject§.ageRestriction <= §age§"
    for "§age§ >= 0";

relation AllowedForNoone() <: AllowedFor(y) :<==> "true";
relation AllowedFor(x) <: AllowedForEveryone() :<==> "true";
relation AllowedFor(x) <: AllowedFor(y) :<==> "§x§ < §y§";
```

Listing 5.2: lattice_allowedfor

The additional `@JMLClauses` are necessary to prove the correctness of other classes that use the `Product` class. We already saw an explanation for something similar to this in Section 3.4.3.

The restrictions of the values of `price` and `ageRestriction` are enforced using a property annotation of type `@Interval`. The hierarchy of `@Interval` annotations is defined in the most obvious way, with $[a, b]$ being a sub-annotation of $[c, d]$ if $[a, b]$ is contained in $[c, d]$.

In addition, we define a property annotation type `@AllowedFor`, which allows us to type products according to their age restriction: a product has the type `@AllowedFor(age="n") Product` if customers aged `n` years are allowed to buy it. The hierarchy of these annotations is also obvious, with $\texttt{AllowedFor}(a) \leq \texttt{AllowedFor}(b)$ if $a \geq b$. The full lattice file can be seen in Listing 5.2.

The `@Interval` annotations shown here, as well as the implicit `@NonNull` annotations, are all constant, so the checker can verify that the assignments in the constructor and the return statements in the getter methods are all well-typed.

The annotation `@AllowedFor(age="ageRestriction")` on the constructor has the non-constant parameter `price`. Also, constructors that are annotated with anything other than $\top$ cannot be well-typed anyway. This means that the fact that the constructor returns an object of type `@AllowedFor(age="ageRestriction") Product` has to be proven in KeY.

Listing 5.3 shows how the constructor is translated to JML. Since all assignments inside the constructor are well-typed, the only thing left to be proven are the post-conditions, those being the post-conditions specified via `@JMLClause`, and the post-condition belonging to `@AllowedFor(age="price")`.

Because we have chosen `@NonNull` as the implicit default annotation for all objects, we get the additional post-condition `this != null`. The fact that a constructor returns

```
/*@ public behavior
  @ diverges true;
  @ requires \typeof(price) <: int
  @            && (0 >= 0 && 0 <= 2147483647)
  @            && (price >= 0 && price <= 2147483647);
  @ requires \typeof(ageRestriction) <: int
  @            && (0 >= 0 && 0 <= 18)
  @            && (ageRestriction >= 0 && ageRestriction <= 18);
  @ requires \typeof(title) <: java.lang.Object
  @            && (true) && (title != null);
  @ assignable this.*;
  @ ensures \fresh(this) && \fresh(this.*);
  @ ensures this.title == title && this.price == price
  @      && this.ageRestriction == ageRestriction;
  @ ensures \typeof(this) <: case_study.Product
  @      && (ageRestriction >= 0)
  @      && (this != null && this.ageRestriction <= ageRestriction);
  @ ensures \typeof(this) <: java.lang.Object && (true) && (this != null);
  @*/
public Product(
        /*@nullable@*/ java.lang.String title,
        int price, int ageRestriction) {
    super();

    java.lang.String temp0 = title;
    //@ assume \typeof(temp0) <: java.lang.Object
    //@     && (true) && (temp0 != null);
    this.title = temp0;
    int temp1 = price;
    //@ assume \typeof(temp1) <: int
    //@     && (0 >= 0 && 0 <= 2147483647)
    //@     && (temp1 >= 0 && temp1 <= 2147483647);
    this.price = temp1;
    int temp2 = ageRestriction;
    //@ assume \typeof(temp2) <: int
    //@     && (0 >= 0 && 0 <= 18)
    //@     && (temp2 >= 0 && temp2 <= 18);
    this.ageRestriction = temp2;
}
```

Listing 5.3: Product constructor in JML

```
1  public class Order {
2      public Customer customer;
3      public @AllowedFor(age="customer.age") Product product;
4
5      @JMLClause(values={
6          "assignable this.*",
7          "ensures \fresh(this) && \fresh(this.*)",
8          "ensures this.customer == customer && this.product == product"})
9      public Order(Customer customer, @AllowedFor(age="customer.age") Product
       product) {
10         this.customer = customer;
11         this.product = product;
12     }
13
14     @JMLClause(values={"assignable \nothing"})
15     public Customer getCustomer() {
16         return customer;
17     }
18
19     @JMLClause(values={"assignable \nothing"})
20     public @AllowedFor(age="this.customer.age") Product getProduct() {
21         return product;
22     }
23 }
```

Listing 5.4: Order.java

a non-null object is obvious, but the system cannot reason about the property in this way. In the future, it would be nice to be able to specify and use lattice-specific rules like "a constructor's result is always `@NonNull`" without having to resort to KeY.

Listing 5.4 shows the `Order` class.

An order consists of a customer and a product, with the product's type `@AllowedFor(age="customer.age") Product` depending on the customer's age. Since this annotation's parameter is evidently not a constant, anything to do with the product's type must be verified in KeY. KeY rejects as invalid the creation of any order where the product's type indicates that the customer is too young to buy it.

How can we now use these types? Let us first define two products:

```
@AllowedFor(age="18") Product product18
    = new Product("Louisiana Buzzsaw Carnage", 10, 18);
@AllowedFor(age="6") Product product6
    = new Product("Tim & Jeffrey, All Episodes", 10, 6);
```

These two assignments are valid, but not well-typed. In the future, it would be nice to include a variable substitution mechanism in the checker, but for now an assignment from the type `@AllowedFor(age="this.age") Product` to `@AllowedFor(age="18") Product` is not well-typed even if `this.age == 18`.

This restriction could be worked around by including a method like the following:

```
@AllowedFor(age="18") static Product product18(
        String title,
        @Interval(min="0", max="2147483647") int price) {
    return new Product(title, price, 18);
}
```

Of course, this method itself is not well-typed, but any client code that uses it instead of calling the `Product` constructor directly is, so we would only have to verify this single method in KeY instead of every single method that creates a new `Product`.

Next, we define three customers:

```
Customer customer18 = new Customer("Alice", 18);
Customer customer14 = new Customer("Bob", 14);
Customer customer6  = new Customer("Charlie", 6);
```

The first two lines are well-typed. The third line is neither well-typed nor valid because a customer's age has the type `@Interval(min="14", max="150") int` to represent the fact that only people aged 14 or older are allowed to register an account with our web shop.

Now, let us try to order some products for the two valid customers:

```
customer18.order(product18);
customer18.order(product6);
customer14.order(product18);
customer14.order(product6);
```

The signature of `Customer.order` is `void order(@AllowedFor(age="this.age") Product product)`.

Because it includes a non-constant annotation, none of these lines are well-typed. They are translated to JML as follows:

```
customer18.__order_trampoline(product18, false);
customer18.__order_trampoline(product6, false);
customer14.__order_trampoline(product18, false);
customer14.__order_trampoline(product6, false);
```

with `Customer.order(...)` and its trampoline method defined as follows (for simplicity's sake, we only include the properties for annotations in the `@AllowedFor` lattice):

```
/*@ public behavior
  @ diverges true;
  @ requires \typeof(product) <: case_study.Product
  @       && (this.age >= 0)
  @       && (product != null && product.ageRestriction <= this.age);
  @ assignable Shop.instance.orders;
  @*/
public /*@helper@*/ void order(/*@nullable@*/ case_study.Product product) {
    Shop.__getInstance_trampoline().__addOrder_trampoline(
        Order.__INIT_trampoline(this, product));
}

/*@ public behavior
  @ diverges true;
  @ requires product_allowedfor || (...);
  @ requires_free !product_allowedfor || (...);
  @ assignable Shop.instance.orders;
  @*/
public /*@helper@*/  void __order_trampoline(
        /*@nullable@*/ case_study.Product product,
        boolean product_allowedfor) {
    order(product);
}
```

This works like all the previous trampoline examples we have seen.

Because the parameter `product_allowedfor` is false for all calls to `__order_trampoline()`, the right-hand part of the `requires` clause must be proven. This proof succeeds if and only if the customer's age is high enough to buy the product.

5.1.2. Lists and queues

In this sub-section, we present parts of the `List` and `Queue` classes.

❊

Listing 5.5 shows part of `List`.

As we can see, a list consists of a head of type `Object` (for simplicity's sake, we assume without proof that all objects that are put into a list are immutable), and a nullable tail of type `List`. Empty lists are represented by `null`.

The annotation `@Positive` on the list's size, as well as the implicit annotation `@NonNull` on the head pose no problem to the type checker.

In addition to these, we have some `@Length` annotations that tell us that a list's tail is one shorter than the list itself. This has to be verified in KeY.

Strictly speaking, the constructor shown in Listing 5.5 is incorrect because the `size` field may overflow. When proving the correctness of this program in KeY, we used the semantics of the mathematical integers instead of the modulo semantics of Java integers. When using real Java semantics, an additional pre-condition must be added to the constructor to ensure that the `tail` argument has a length no greater than $2^{31} - 2$.

❊

Listing 5.6 shows part of `Queue`.

As explained above, a queue consists of two lists `front` and `back`.

The `remove()` method requires that `front` not be empty. This property is encoded in the annotation `@FrontNonEmpty`.

The `toOkasaki()` method re-instantes the Okasaki invariant by calling `rotate()` to move all elements in `back` into `front` in the correct order.

The `Shop` class, shown in Listing 5.7, demonstrates how such a queue can be used.

The type system ensures that every time the `orders` queue is modified, the Okasaki invariant remains valid. It is easy to see that this is the case, since after every modification `Queue.toOkasaki()` is called.

Furthermore, the system ensures that `Queue.remove()` is only called on queues of type `@FrontNonEmpty`. The call to `orders.remove()` is the only part of the `Shop` class that is not well-typed, since `@Okasaki Queue` is not a sub-type of `@FrontNonEmpty Queue` (as an empty queue also satisfies the Okasaki invariant).

We must thus use KeY to try and show that inside the branch `if (orders.size() > 0)`, the queue's front cannot be empty. This can in fact be shown because we know that inside the branch `orders.size() > 0` holds, which is equivalent to `orders.front.size() + orders.back.size() > 0`. Since `orders` has the type `@Okasaki Queue`, we know that `orders.front.size() >= orders.back.size()`. Thus, it follows that `orders.front.size() > 0`.

```
1  public class List {
2      public Object head;
3
4      public @Nullable
5      @Length(min="this.size - 1", max="this.size - 1")
6      List tail;
7
8      public @Positive int size;
9
10     public
11     @Length(min="tail.size + 1", max="tail.size + 1")
12     List(Object head, List tail) {
13         this.size = tail.size() + 1;
14         this.head = head;
15         this.tail = tail;
16     }
17
18     public Object head() { return head; }
19
20     public
21     @Nullable
22     @Length(min="this.size - 1", max="this.size - 1")
23     List tail() { return tail; }
24
25     public @Positive int size() { return size; }
26 }
```

Listing 5.5: List.java

```java
public class Queue {
    public @Nullable List front;
    public @Nullable List back;

    public Queue(@Nullable List front, @Nullable List back) { ... }

    private static @Nullable List rotate(
            @Nullable List front, @Nullable List back, @Nullable List acc) {
        if (front == null && back == null) {
            return acc;
        } else if (front == null) {
            return rotate(null, back.tail(), List.cons(back.head(), acc));
        } else if (back == null) {
            return List.cons(front.head(), rotate(front.tail(), null, acc));
        } else {
            return List.cons(front.head(), rotate(front.tail(), back.tail(),
    List.cons(back.head(), acc)));
        }
    }

    @JMLClause(values={
        "ensures front == null && back == null ==> \result == 0",
        "ensures front == null && back != null ==> \result == back.size",
        "ensures front != null && back == null ==> \result == front.size",
        "ensures front != null && back != null"
                + " ==> \result == front.size + back.size",
        })
    public int size() { return List.size(front) + List.size(back); }

    public @Okasaki Queue toOkasaki() {
        if (back == null || (front != null && front.size() >= back.size())) {
            return this;
        } else {
            Queue result = new Queue(rotate(front, back, null), null);
            return result;
        }
    }

    public Queue remove(@FrontNonEmpty Queue this) {
        return new Queue(front.tail(), back);
    }
}
```

Listing 5.6: Queue.java

```
1  public class Shop {
2      private @Okasaki Queue orders = new Queue(null, null).toOkasaki();
3
4      public void addOrder(Order order) {
5          orders = orders.insert(order).toOkasaki();
6      }
7
8      public boolean processNextOrder() {
9          if (orders.size() > 0) {
10             orders = orders.remove().toOkasaki();
11             return true;
12         } else {
13             return false;
14         }
15     }
16 }
```

Listing 5.7: Shop.java

5.2. Evaluation

To evaluate this case study, we first give a rough overview over what properties we were able to show in Table 5.2.

As we can see, we were a able to show quite a few things with a relatively small specification overhead.

Obviously, having to specify the lattice files adds an overhead that would not be there if we had just used KeY, but this should be weighted against the fact that property types allow us to delegate the creation of these lattice files to one or a few members of a team, while the other programmers can simply use the annotations in a relatively intuitive way.

Another thing that adds unnecessary overhead is the fact that for some of the parts which required KeY, we had to add additional JML clauses to some methods. Some of this can be mitigated by encapsulating verification-heavy code. For some other parts, like the `assignable` clauses, it may be possible to infer the needed specifications programatically.

Nevertheless, using property types in a way that relies heavily on KeY does add a substantial specification overhead, though the overall size of the specifications (including JML clauses and property annotations) is still quite small. As evidenced by the property-type-to-JML translations, it is certainly smaller than if we had used JML exclusively.

Property	Use of KeY necessary for
`@Interval`	nothing.
`@Positive`	nothing.
`@Length`	showing that a list's tail is always one shorter than the list itself.
`@NonNull`	showing that the results of constructors are not null; refining nullable variables to non-null, e.g., in branches `if (x != null)`.
`@Okasaki`; `@FrontNonEmpty`	establishing the Okasaki invariant in `Queue.toOkasaki()`; refining `@Okasaki` to `@OkasakiNonEmpty` in `Shop.processNextOrder()`.
`@AllowedFor`	establishing and using the invariant of the `Order` class.

Table 5.2.: Shown properties

As for the verification overhead, Table 5.3 shows the time it took to verify the correctness of each class in KeY. The times for the individual methods in each class were added up. The contracts for trampoline methods were not proven, as they are correct by construction.

All proofs for this case study could be found by KeY's automatic mode, i.e., no manual intervention was necessary.

To find these proofs, KeY was executed on a computer with an Intel Core i7-4720HQ (2x2.60GHz) processor and 16 GB of RAM.

Class	Time for proof
`Customer`	3.8*s*
`List`	11.0*s*
`Main`	33.5*s*
`Order`	2.4*s*
`Product`	2.7*s*
`Queue`	239.4*s*
`Shop`	10.9*s*
total	303.6*s*

Table 5.3.: Proof times

This shows that at around five minutes, the verification overhead for this case study was very small. The class with the biggest overhead was `Queue`, of which the method `Queue.rotate()` took the longest time to prove.

The class with the second-biggest overhead was `Main`, which creates and uses objects of all of the other classes. This is mostly due to the fact that, as explained above – the type system does not support any form of variable substitution.

The correctness proofs for the other classes, which constitute the business logic of this program, all could be found in under fifteen seconds.

6. Conclusion & outlook

6.1. Conclusion

In the preceding chapters, we introduced and formalized a way to use Java annotations to encode some of the desired properties of a variable into that variable's type.

We introduced a general way to combine multiple such property types into a type hierarchy, and saw how we could use multiple such hierarchies in one program.

We also saw how property types can be translated to JML [Lea+13], and how this allows us to build a verification pipeline that is easier to use than a traditional formal verification tool while also not being as over-approximative as a traditional type system. Instead of relying on incomplete type refinement rules that are specific to one type system, the power of formal verification allows us to attempt to refine a variable's type whenever we want while still being able to verify that the refinement is correct.

Translating property types to JML also allows us to support dependent types without having to support these types in the type system itself.

While there exist pluggable type systems for Java, like those included in the Checker Framework [Che20], and also run-time validation frameworks like BeanValidation [Mor19], the framework presented in this thesis allows users to define their own pluggable type systems without very much programming work. Furthermore, it demonstrates that type systems that include refinement types or general dependent types – which are of course well-known to be well-suited to functional programming languages – can also be applied to imperative languages.

6.2. Outlook

First, this thesis focused on just proving or disproving the correctness of an existing program specification. It could be very helpful to programmers if – given an incorrect program – the verification pipeline were able to apply some kind of weakest-precondition calculus and insert additional assertions or checks into the program to make it correct. This could be combined with type inference on the side of the Checker Framework – as seen for example in [XLD20] – which checks if there exists some valid typing for a given program.

It would also be interesting to see if and how the property type system could be extended to allow for properties of mutable objects to be verified. One would probably

have to include an ownership system – like the one presented in [Die09] – to deal with the problems presented by reference aliasing. One would also have to investigate whether it would generally be more useful for an object to always have to belong to the type under which it was created, or for an object to be allowed to change its type under some circumstances.

Support for Java generics would also be very useful. While the Checker Framework does support generics, KeY does not. One would have to introduce a way to allow a method's JML contract depend on the instantiation of the type variables.

Furthermore, it would also be interesting to leverage the Checker Framework's support for dependent types in the property type system instead of delegating all of that to KeY.

Creating some means for the user to define lattice-specific type rules and to verify the correctness of those rules in KeY would greatly cut down on the verification overhead, especially for properties of primitive types. It would also put the property type checker closer to other type checkers developed in the Checker Framework, almost all of which implement some refinement rules. As it is, the type checker does not know anything about the type of the expression `a + b` even if it knows the most specific types of `a` and `b`.

Last but not least, the implementation of the type-checking pipeline could still be improved. Firstly, forcing the user to put the annotation types in the checker's class path is obviously not ideal for practical use. Supporting all meta-annotations in the Java library and the Checker Framework in the lattice language and using that to generate Java files for the annotation types programatically would make the type checker much easier to use. Secondly, some parts of the pipeline implementation are still missing. One could quite easily generate KeY-readable proof obligations to prove the lattice properties of the annotation hierarchy. One could also integrate the Checker Framework's initialization checker as well as the Glacier immutability checker into the property checker.

Bibliography

[Ahr+16] Wolfgang Ahrendt et al., eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, Dec. 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, July 2017. URL: www.SMT-LIB.org.

[Bra04] Gilad Bracha. "Pluggable type systems". In: *OOPSLA'04 Workshop on Revival of Dynamic Languages*. Oct. 2004.

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23 (05 Sept. 2013), pp. 552–593. ISSN: 1469-7653. DOI: 10.1017/S095679681300018X.

[Che20] Checker Framework developers. *Checker Framework Manual*. Version 3.3.0. Apr. 1, 2020. URL: https://checkerframework.org/manual/.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types*. Dec. 2013. ISBN: 9780262026659. URL: http://adam.chlipala.net/cpdt/. published.

[Cob+17] Michael Coblenz et al. "Glacier: Transitive class immutability for Java". In: *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*. Buenos Aires, Argentina, May 2017, pp. 496–506. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE.2017.52.

[Cok11] David R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5. DOI: 10.1007/978-3-642-20398-5_35.

[Die+11] Werner Dietl et al. "Building and Using Pluggable Type-Checkers". In: *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. May 2011, pp. 681–690. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985889.

[Die09] Werner Dietl. "Universe Types: Topology, Encapsulation, Genericity, and Tools". Doctoral Thesis ETH No. 18522. Ph.D. Department of Computer Science, ETH Zurich, Dec. 2009.

[Gos+15] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. Mar. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[Ham97] Graham Hamilton, ed. *JavaBeans*. Version 1.01. Sun Microsystems, Aug. 8, 1997. URL: https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/.

[JSV17] Ranjit Jhala, Eric Seidel, and Niki Vazou. *Programming with Refinement Types – Introduction to LiquidHaskell*. Mar. 2017. URL: https://ucsd-progsys.github.io/liquidhaskell-blog/.

[Lan18] Florian Lanzinger. "A Divide-and-Conquer Strategy with Block and Loop Contracts for Deductive Program Verification". Bachelor Thesis. Karlsruher Institut für Technologie, Apr. 2018.

[Lea+13] Gary T. Leavens et al. *JML Reference Manual*. Revision 2344. May 31, 2013. URL: http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf.

[Mey92] Bertrand Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279.

[Mor19] Gunnar Morling. *Bean Validation specification*. Version 2.0. Aug. 5, 2019. URL: https://beanvalidation.org/2.0/spec/.

[Oka95] Chris Okasaki. "Simple and efficient purely functional queues and deques". In: *Journal of Functional Programming* 5.4 (1995), pp. 583–592.

[SM11] Alexander Summers and Peter Müller. "Freedom before commitment". In: *ACM SIGPLAN Notices*. Vol. 46. Oct. 2011, p. 1013. ISBN: 9781450309400. DOI: 10.1145/2048066.2048142.

[Vaz+14] Niki Vazou et al. "Refinement Types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. ACM, Sept. 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161.

[Wac12] Simon Wacker. "Blockverträge". Studienarbeit. Karlsruher Institut für Technologie, Oct. 2012.

[XLD20] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. "Precise Inference of Expressive Units of Measurement Types". In: 4.OOPSLA (Nov. 2020). ISSN: 2475-1421. DOI: 10.1145/3428210.

A. Source code for the case study

A.1. Lattice files

```
ident allowedfor;

annotation AllowedForEveryone() any :<==> "true" for "true";
annotation AllowedForNoone() any :<==> "false" for "true";

annotation AllowedFor(int age) case_study.Product
    :<==> "§subject§ != null && §subject§.ageRestriction <= §age§"
    for "§age§ >= 0";

relation AllowedForNoone() <: AllowedFor(y) :<==> "true";
relation AllowedFor(x) <: AllowedForEveryone() :<==> "true";
relation AllowedFor(x) <: AllowedFor(y) :<==> "§x§ < §y§";
```

Listing A.1: lattice_allowedfor

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
import org.checkerframework.framework.qual.SubtypeOf;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
@DefaultQualifierInHierarchy
@SubtypeOf({})
public @interface AllowedForEveryone {}
```

Listing A.2: AllowedForEveryone.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

```

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({AllowedForEveryone.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface AllowedFor {

    String age();
}
```

Listing A.3: AllowedFor.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;
import org.checkerframework.framework.qual.TargetLocations;
import org.checkerframework.framework.qual.TypeUseLocation;

@SubtypeOf({AllowedFor.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@TargetLocations({TypeUseLocation.EXPLICIT_LOWER_BOUND, TypeUseLocation.
    EXPLICIT_UPPER_BOUND})
public @interface AllowedForNoone {}
```

Listing A.4: AllowedForNoone.java

```
ident interval;

annotation IntervalTop() any :<==> "true" for "true";
annotation IntervalBottom() any :<==> "false" for "true";

annotation Interval(int min, int max) int
    :<==> "§subject§ >= §min§ && §subject§ <= §max§"
```

```
    for "§min§ >= 0 && §min§ <= §max§";

relation Interval(a,b) <: IntervalTop() :<==> "true";
relation IntervalBottom() <: Interval(a,b) :<==> "true";

relation Interval(a0, b0) <: Interval(a1, b1) :<==> "§a0§ >= §a1§ && §b0§ <= §b1§";

join Interval(a0, b0), Interval(a1, b1) := Interval("java.lang.Math.min(§a0§, §a1§)", "java.lang.Math.max(§b0§, §b1§)");

# overlapping
meet Interval(a0, b0), Interval(a1, b1)
    := Interval("java.lang.Math.max(§a0§, §a1§)", "java.lang.Math.min(§b0§, §b1§)")
    for "§b0§ >= §a1§ || §b1§ >= §a0§";

# non-overlapping
meet Interval(a0, b0), Interval(a1, b1) := IntervalBottom(); # "!(b0 >= a1 || b1 >= a0)" is implicit!
```

Listing A.5: lattice_interval

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
import org.checkerframework.framework.qual.SubtypeOf;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
@DefaultQualifierInHierarchy
@SubtypeOf({})
public @interface IntervalTop {}
```

Listing A.6: IntervalTop.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
```

```
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({IntervalTop.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface Interval {
14
15     String min();
16     String max();
17 }
```

Listing A.7: Interval.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10
11 @SubtypeOf({Interval.class})
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target({ElementType.TYPE_USE})
14 public @interface IntervalBottom {}
```

Listing A.8: IntervalBottom.java

```
1  ident length;
2
3  annotation UnknownLength() any :<==> "true" for "true";
4  annotation BottomLength() any :<==> "false" for "true";
5
6  annotation Length(int min, int max) case_study.List
7      :<==> "(§subject§ == null && §min§ <= 0 && §max§ >= 0) || (§subject§.size
       >= §min§ && §subject§.size <= §max§)"
8      for "§min§ >= 0 && §min§ <= §max§";
9
10 relation Length(a0, b0) <: Length(a1, b1) :<==> "§a0§ >= §a1§ && §b0§ <= §b1§
       ";
```

```
11
12 relation Length(a,b) <: UnknownLength() :<==> "true";
13 relation BottomLength() <: Length(a,b) :<==> "true";
14 relation BottomLength() <: UnknownLength() :<==> "true";
15
16 join Length(a0, b0), Length(a1, b1) := Length("java.lang.Math.min(§a0§, §a1§)
       ", "java.lang.Math.max(§b0§, §b1§)");
17
18 # overlapping
19 meet Length(a0, b0), Length(a1, b1)
20     := Length("java.lang.Math.max(§a0§, §a1§)", "java.lang.Math.min(§b0§, §b1
       §)")
21     for "§b0§ >= §a1§ || §b1§ >= §a0§";
22
23 # non-overlapping
24 meet Length(a0, b0), Length(a1, b1) := BottomLength(); # "!(b0 >= a1 || b1 >=
        a0)" is implicit!
```

Listing A.9: lattice_length

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7  import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @Retention(RetentionPolicy.RUNTIME)
11 @Target({ElementType.TYPE_USE})
12 @DefaultQualifierInHierarchy
13 @SubtypeOf({})
14 public @interface UnknownLength {}
```

Listing A.10: UnknownLength.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
```

```java
import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({UnknownLength.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface Length {

    String min() default "0";
    String max() default "2147483647";
}
```

Listing A.11: Length.java

```java
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({Length.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface BottomLength {}
```

Listing A.12: BottomLength.java

```
ident nullness;

annotation NullTop() any :<==> "true" for "true";
annotation NullBottom() any :<==> "false" for "true";

annotation Nullable() java.lang.Object :<==> "true" for "true";
annotation NonNull() java.lang.Object :<==> "§subject§ != null" for "true";

relation NullBottom() <: NullTop() :<==> "true";
relation NullBottom() <: Nullable() :<==> "true";
relation NullBottom() <: NonNull() :<==> "true";

relation NonNull() <: NullTop() :<==> "true";
relation NonNull() <: Nullable() :<==> "true";

```

```
relation Nullable() <: NullTop() :<==> "true";
```

Listing A.13: lattice_nullness

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.DefaultFor;
import org.checkerframework.framework.qual.SubtypeOf;
import org.checkerframework.framework.qual.TypeKind;

@DefaultFor(
        typeKinds = {TypeKind.BOOLEAN, TypeKind.BYTE, TypeKind.CHAR, TypeKind
    .DOUBLE, TypeKind.FLOAT, TypeKind.INT, TypeKind.LONG})
@SubtypeOf({})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface NullTop {}
```

Listing A.14: NullTop.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.RelevantJavaTypes;
import org.checkerframework.framework.qual.SubtypeOf;

@RelevantJavaTypes({Object.class})
@SubtypeOf({NullTop.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface Nullable {}
```

Listing A.15: Nullable.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
```

```
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
9  import org.checkerframework.framework.qual.RelevantJavaTypes;
10 import org.checkerframework.framework.qual.SubtypeOf;
11
12 @DefaultQualifierInHierarchy
13 @RelevantJavaTypes({Object.class})
14 @SubtypeOf({Nullable.class})
15 @Retention(RetentionPolicy.RUNTIME)
16 @Target({ElementType.TYPE_USE})
17 public @interface NonNull {
18
19 }
```

Listing A.16: NonNull.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({NonNull.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface NullBottom {}
```

Listing A.17: NullBottom.java

```
1  ident okasaki;
2
3  annotation TopOkasaki() any :<==> "true" for "true";
4  annotation BottomOkasaki() any :<==> "false" for "true";
5
6  annotation Okasaki() case_study.Queue
7      :<==> "§subject§ != null && (§subject§.back == null || (§subject§.front
       != null && §subject§.front.size >= §subject§.back.size))"
```

```
8       for "true";
9
10  annotation FrontNonEmpty() case_study.Queue
11      :<==> "§subject§ != null && §subject§.front != null && §subject§.front.
        size > 0"
12      for "true";
13
14  annotation OkasakiNonEmpty() case_study.Queue
15      :<==> "§subject§ != null && §subject§.front != null && §subject§.front.
        size > 0 && (§subject§.back == null || (§subject§.front != null && §
        subject§.front.size >= §subject§.back.size))"
16      for "true";
17
18  relation FrontNonEmpty() <: TopOkasaki() :<==> "true";
19  relation Okasaki() <: TopOkasaki() :<==> "true";
20
21  relation OkasakiNonEmpty() <: TopOkasaki() :<==> "true";
22  relation OkasakiNonEmpty() <: Okasaki() :<==> "true";
23  relation OkasakiNonEmpty() <: FrontNonEmpty() :<==> "true";
24
25  relation BottomOkasaki() <: TopOkasaki() :<==> "true";
26  relation BottomOkasaki() <: Okasaki() :<==> "true";
27  relation BottomOkasaki() <: OkasakiNonEmpty() :<==> "true";
28  relation BottomOkasaki() <: FrontNonEmpty() :<==> "true";
```

Listing A.18: lattice_okasaki

```
1   package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3   import java.lang.annotation.ElementType;
4   import java.lang.annotation.Retention;
5   import java.lang.annotation.RetentionPolicy;
6   import java.lang.annotation.Target;
7   import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
8   import org.checkerframework.framework.qual.SubtypeOf;
9
10  @Retention(RetentionPolicy.RUNTIME)
11  @Target({ElementType.TYPE_USE})
12  @DefaultQualifierInHierarchy
13  @SubtypeOf({})
14  public @interface TopOkasaki {}
```

Listing A.19: TopOkasaki.java

```java
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({Okasaki.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface OkasakiNonEmpty {

}
```

Listing A.20: OkasakiNonEmpty.java

```java
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({TopOkasaki.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface Okasaki {

}
```

Listing A.21: Okasaki.java

```java
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

```

```
import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({TopOkasaki.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface FrontNonEmpty {

}
```

Listing A.22: FrontNonEmpty.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({OkasakiNonEmpty.class, FrontNonEmpty.class})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
public @interface BottomOkasaki {}
```

Listing A.23: BottomOkasaki.java

```
ident sign;

annotation TopSign() any :<==> "true" for "true";
annotation BottomSign() any :<==> "false" for "true";

annotation NonNegative() int :<==> "§subject§ >= 0" for "true";
annotation NonPositive() int :<==> "§subject§ <= 0" for "true";
annotation Negative() int :<==> "§subject§ < 0" for "true";
annotation Positive() int :<==> "§subject§ > 0" for "true";
annotation Zero() int :<==> "§subject§ == 0" for "true";

relation NonNegative() <: TopSign() :<==> "true";
relation NonPositive() <: TopSign() :<==> "true";

relation Positive() <: TopSign() :<==> "true";
relation Positive() <: NonNegative() :<==> "true";

```

```
relation Negative() <: TopSign() :<==> "true";
relation Negative() <: NonPositive() :<==> "true";

relation Zero() <: TopSign() :<==> "true";
relation Zero() <: NonNegative() :<==> "true";
relation Zero() <: NonPositive() :<==> "true";

relation BottomSign() <: TopSign() :<==> "true";
relation BottomSign() <: NonNegative() :<==> "true";
relation BottomSign() <: NonPositive() :<==> "true";
relation BottomSign() <: Negative() :<==> "true";
relation BottomSign() <: Positive() :<==> "true";
relation BottomSign() <: Zero() :<==> "true";
```

Listing A.24: lattice_sign

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
import org.checkerframework.framework.qual.SubtypeOf;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE})
@DefaultQualifierInHierarchy
@SubtypeOf({})
public @interface TopSign {}
```

Listing A.25: TopSign.java

```
package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.checkerframework.framework.qual.SubtypeOf;

@SubtypeOf({TopSign.class})
@Retention(RetentionPolicy.RUNTIME)
```

```
12 @Target({ElementType.TYPE_USE})
13 public @interface NonNegative {
14 }
```

Listing A.26: NonNegative.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({TopSign.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface NonPositive {
14 }
```

Listing A.27: NonPositive.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({NonPositive.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface Negative {
14 }
```

Listing A.28: Negative.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
```

```
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({NonNegative.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface Positive {
14 }
```

Listing A.29: Positive.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({NonNegative.class, NonPositive.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface Zero {
14 }
```

Listing A.30: Zero.java

```
1  package edu.kit.iti.checker.property.subchecker.lattice.case_study_qual;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  import org.checkerframework.framework.qual.SubtypeOf;
9
10 @SubtypeOf({Positive.class, Negative.class, Zero.class})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Target({ElementType.TYPE_USE})
13 public @interface BottomSign {
14 }
```

Listing A.31: BottomSign.java

A.2. Program files

```
package case_study;

import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
import edu.kit.iti.checker.property.checker.qual.*;

public class Customer {

    public String name;
    public @Interval(min="14", max="150") int age;

    @JMLClause(values={
        "assignable this.*",
        "ensures \fresh(this) && \fresh(this.*)",
        "ensures this.name == name && this.age == age"})
    // :: error: inconsistent.constructor.type
    public Customer(String name, @Interval(min="14", max="150") int age) {
        this.name = name;
        this.age = age;
    }

    @JMLClause(values={"assignable \nothing"})
    public String getName() {
        return name;
    }

    @JMLClause(values={"assignable \nothing"})
    public @Interval(min="14", max="150") int getAge() {
        return age;
    }

    @JMLClause(values={"assignable Shop.instance.orders"})
    public void order(@AllowedFor(age="this.age") Product product) {
        // :: error: argument.type.incompatible
        Shop.getInstance().addOrder(new Order(this, product));
    }
}
```

Listing A.32: Customer.java

```
package case_study;

import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
import edu.kit.iti.checker.property.checker.qual.*;

public class List {

    private Object head;

    private @Nullable
    @Length(min="this.size - 1", max="this.size - 1")
    List tail;

    public @Positive int size;

    @JMLClause(values={
        "assignable this.*",
        "ensures \fresh(this) && \fresh(this.*)"})
    public
    @Length(min="tail.size + 1", max="tail.size + 1")
    // :: error: inconsistent.constructor.type
    List(Object head, List tail) {
        // :: error: assignment.type.incompatible
        this.size = tail.size() + 1;
        this.head = head;
        // :: error: assignment.type.incompatible
        this.tail = tail;
    }

    @JMLClause(values={
        "assignable this.*",
        "ensures \fresh(this) && \fresh(this.*)"})
    public
    @Length(min="1", max="1")
    // :: error: inconsistent.constructor.type
    List(Object head) {
        this.size = 1;
        this.head = head;
        // :: error: assignment.type.incompatible
        this.tail = tail;
```

```
41      }
42
43      @JMLClause(values={
44          "assignable \nothing",
45          "ensures \fresh(\result) && \fresh(\result.*)"})
46      public static List cons(Object head, @Nullable List tail) {
47          if (tail == null) {
48              return new List(head, tail);
49          } else {
50              return new List(head);
51          }
52      }
53
54      @JMLClause(values={"assignable \nothing"})
55      public Object head() {
56          return head;
57      }
58
59      @JMLClause(values={"assignable \nothing"})
60      public
61      @Nullable
62      @Length(min="this.size - 1", max="this.size - 1")
63      List tail() {
64          // :: error: return.type.incompatible
65          return tail;
66      }
67
68      @JMLClause(values={
69          "assignable \nothing",
70          "ensures \result == this.size"
71          })
72      public @Positive int size() {
73          return size;
74      }
75
76      @JMLClause(values={
77          "assignable \nothing",
78          "ensures l != null ==> \result == l.size",
79          "ensures l == null ==> \result == 0"
80          })
81      public static @NonNegative int size(@Nullable List l) {
82          if (l == null) {
83              return 0;
```

```
        } else {
            // :: error: method.invocation.invalid
            return l.size();
        }
    }
}
```

Listing A.33: List.java

```
package case_study;

import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
import edu.kit.iti.checker.property.checker.qual.*;

public class Main {

    // :: error: inconsistent.constructor.type
    private Main() { }

    public static void main(String[] args) {
        // :: error: assignment.type.incompatible
        @AllowedFor(age="18") Product product18 = new Product("Louisiana
    Buzzsaw Carnage", 10, 18);
        // :: error: assignment.type.incompatible
        @AllowedFor(age="6") Product product6 = new Product("Tim & Jeffrey,
    All Episodes", 10, 6);

        Customer customer18 = new Customer("Alice", 18);
        Customer customer14 = new Customer("Bob", 14);

        //Customer customer6 = new Customer("Charlie", 6);

        // :: error: argument.type.incompatible
        customer18.order(product18);
        // :: error: argument.type.incompatible
        customer18.order(product6);

        //customer14.order(product18);

        // :: error: argument.type.incompatible
        customer14.order(product6);

        Shop.getInstance().processNextOrder();
        Shop.getInstance().processNextOrder();
```

```
34          Shop.getInstance().processNextOrder();
35
36          Shop.getInstance().processNextOrder();
37      }
38  }
```

Listing A.34: Main.java

```
1   package case_study;
2
3   import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
4   import edu.kit.iti.checker.property.checker.qual.*;
5
6   public class Order {
7
8       public Customer customer;
9       public @AllowedFor(age="this.customer.age") Product product;
10
11      @JMLClause(values={
12          "assignable this.*",
13          "ensures \fresh(this) && \fresh(this.*)",
14          "ensures this.customer == customer && this.product == product"})
15      // :: error: inconsistent.constructor.type
16      public Order(Customer customer, @AllowedFor(age="customer.age") Product
        product) {
17          this.customer = customer;
18          // :: error: assignment.type.incompatible
19          this.product = product;
20      }
21
22      @JMLClause(values={"assignable \nothing"})
23      public Customer getCustomer() {
24          return customer;
25      }
26
27      @JMLClause(values={"assignable \nothing"})
28      public @AllowedFor(age="this.customer.age") Product getProduct() {
29          // :: error: return.type.incompatible
30          return product;
31      }
32  }
```

Listing A.35: Order.java

```java
package case_study;

import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
import edu.kit.iti.checker.property.checker.qual.*;

public class Product {

    public String title;
    public @Interval(min="0", max="2147483647") int price;
    public @Interval(min="0", max="18") int ageRestriction;

    @JMLClause(values={
        "assignable this.*",
        "ensures \fresh(this) && \fresh(this.*)",
        "ensures this.title == title && this.price == price && this.
    ageRestriction == ageRestriction"})
    // :: error: inconsistent.constructor.type
    public @AllowedFor(age="ageRestriction") Product(
            String title,
            @Interval(min="0", max="2147483647") int price,
            @Interval(min="0", max="18") int ageRestriction) {
        this.title = title;
        this.price = price;
        this.ageRestriction = ageRestriction;
    }

    @JMLClause(values={"assignable \nothing"})
    public String getTitle() {
        return title;
    }

    @JMLClause(values={"assignable \nothing"})
    public @Interval(min="0", max="2147483647") int getPrice() {
        return price;
    }

    @JMLClause(values={"assignable \nothing"})
    public @Interval(min="0", max="18") int getAgeRestriction() {
        return ageRestriction;
    }
}
```

Listing A.36: Product.java

```
1  package case_study;
2
3  import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
4  import edu.kit.iti.checker.property.checker.qual.*;
5
6  public class Queue {
7
8      public @Nullable List front;
9      public @Nullable List back;
10
11     @JMLClause(values={
12         "assignable this.*",
13         "ensures \fresh(this) && \fresh(this.*)",
14         "ensures this.front == front && this.back == back"})
15     // :: error: inconsistent.constructor.type
16     public Queue(@Nullable List front, @Nullable List back) {
17         this.front = front;
18         this.back = back;
19     }
20
21     @JMLClause(values={"assignable \nothing"})
22     private static @Nullable List rotate(@Nullable List front, @Nullable List
       back, @Nullable List acc) {
23         if (front == null && back == null) {
24             return acc;
25         } else if (front == null) {
26             // :: error: method.invocation.invalid
27             return rotate(null, back.tail(), List.cons(back.head(), acc));
28         } else if (back == null) {
29             return List.cons(
30                     // :: error: method.invocation.invalid
31                     front.head(),
32                     // :: error: method.invocation.invalid
33                     rotate(front.tail(), null, acc));
34         } else {
35             return List.cons(
36                     // :: error: method.invocation.invalid
37                     front.head(),
38                     // :: error: method.invocation.invalid
39                     rotate(front.tail(), back.tail(), List.cons(back.head(),
       acc)));
40         }
```

```
41      }
42
43      @JMLClause(values={"assignable \nothing"})
44      public @Okasaki Queue toOkasaki() {
45          // :: error: method.invocation.invalid
46          if (back == null || (front != null && front.size() >= back.size())) {
47              // :: error: return.type.incompatible
48              return this;
49          } else {
50              Queue result = new Queue(rotate(front, back, null), null);
51              // :: error: return.type.incompatible
52              return result;
53          }
54      }
55
56      @JMLClause(values={"assignable \nothing"})
57      public @Nullable List front() {
58          return front;
59      }
60
61      @JMLClause(values={"assignable \nothing"})
62      public @Nullable List back() {
63          return back;
64      }
65
66      @JMLClause(values={
67          "assignable \nothing",
68          "ensures front == null && back == null ==> \result == 0",
69          "ensures front == null && back != null ==> \result == back.size",
70          "ensures front != null && back == null ==> \result == front.size",
71          "ensures front != null && back != null ==> \result == front.size +
    back.size"
72          })
73      public int size() {
74          return List.size(front) + List.size(back);
75      }
76
77      @JMLClause(values={"assignable \nothing"})
78      public Object peek(@FrontNonEmpty Queue this) {
79          // :: error: method.invocation.invalid
80          return front.head();
81      }
82
```

```
83      @JMLClause(values={"assignable \nothing"})
84      public Queue remove(@FrontNonEmpty Queue this) {
85          // :: error: method.invocation.invalid
86          return new Queue(front.tail(), back);
87      }
88
89      @JMLClause(values={"assignable \nothing"})
90      public Queue insert(Object o) {
91          return new Queue(front, List.cons(o, back));
92      }
93
94      @JMLClause(values={"assignable \nothing"})
95      public @Okasaki Queue removeSafe(@FrontNonEmpty Queue this) {
96          return remove().toOkasaki();
97      }
98
99      @JMLClause(values={"assignable \nothing"})
100     public @Okasaki Queue insertSafe(Object o) {
101         return insert(o).toOkasaki();
102     }
103 }
```

Listing A.37: Queue.java

```
1  package case_study;
2
3  import edu.kit.iti.checker.property.subchecker.lattice.case_study_qual.*;
4  import edu.kit.iti.checker.property.checker.qual.*;
5
6  public class Shop {
7
8      public static Shop instance = new Shop();
9
10     @JMLClause(values={
11         "assignable \nothing",
12         "ensures \result == instance"
13         })
14     public static Shop getInstance() {
15         return instance;
16     }
17
18     // :: error: assignment.type.incompatible
19     private @Okasaki Queue orders = new Queue(null, null);
20
```

```
21      @JMLClause(values={
22          "assignable this.*",
23          "ensures \fresh(this) && \fresh(this.*)"})
24      // :: error: inconsistent.constructor.type
25      private Shop() { }
26
27      @JMLClause(values={"assignable \nothing"})
28      @Okasaki Queue getOrders() {
29          return orders;
30      }
31
32      @JMLClause(values={"assignable orders"})
33      public void addOrder(Order order) {
34          orders = orders.insertSafe(order);
35      }
36
37      @JMLClause(values={"assignable orders"})
38      public boolean processNextOrder() {
39          if (orders.size() > 0) {
40              // :: error: method.invocation.invalid
41              orders = orders.removeSafe();
42              return true;
43          } else {
44              return false;
45          }
46      }
47  }
```

Listing A.38: Shop.java

List of Definitions and Algorithms

List of Theorems

List of Examples

List of Listings

List of Figures and Tables

Index