Scalability and Precision by Combining Expressive Type Systems and Deductive Verification

FLORIAN LANZINGER, ALEXANDER WEIGL, and MATTIAS ULBRICH, Karlsruhe Institute of Technology, Germany
WERNER DIETL, University of Waterloo, Canada

Type systems and modern type checkers can be used very successfully to obtain formal correctness guarantees with little specification overhead. However, type systems in practical scenarios have to trade precision for decidability and scalability. Tools for deductive verification, on the other hand, can prove general properties in more cases than a typical type checker can, but they do not scale well. We present a method to complement the scalability of expressive type systems with the precision of deductive program verification approaches. This is achieved by translating the type uses whose correctness the type checker cannot prove into assertions in a specification language, which can be dealt with by a deductive verification tool. Type uses whose correctness the type checker can prove are instead turned into assumptions to aid the verification tool in finding a proof. Our novel approach is introduced both conceptually for a simple imperative language, and practically by a concrete implementation for the Java programming language. The usefulness and power of our approach has been evaluated by discharging known false positives from a real-world program and by a small case study.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Pluggable type systems, Deductive verification, Refinement types

ACM Reference Format:
Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. 2021. Scalability and Precision by Combining Expressive Type Systems and Deductive Verification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 143 (October 2021), 29 pages. https://doi.org/10.1145/3485520

1 INTRODUCTION

Type systems are used very successfully to obtain formal correctness guarantees with little specification overhead. Type systems together with highly engineered type checkers scale well. In recent years, type systems and checkers have been devised for a number of program properties and for a number of programming languages. Recent examples include nullness type systems in C♯, Kotlin, and Scala, ownership types in Rust, and the optional type systems in Python and TypeScript.

However, part of many success stories of type systems in practical scenarios is that they trade precision for decidability and scalability. In order to scale well, they only check for a conservative, decidable approximation of the formal guarantee they are designed for. For example, a sound type system designed to guarantee that no null pointers are dereferenced at run time can often show that large parts of a program are free of null-pointer dereferences. In many cases, however, there remain places in the program where the non-nullness of variables depends on application-specific, more

Authors' addresses: Florian Lanzinger, lanzinger@kit.edu; Alexander Weigl, weigl@kit.edu; Mattias Ulbrich, ulbrich@kit.edu, Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, BW, Germany, 76131; Werner Dietl, wdietl@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada.

2475-1421/2021/10-ART143
https://doi.org/10.1145/3485520

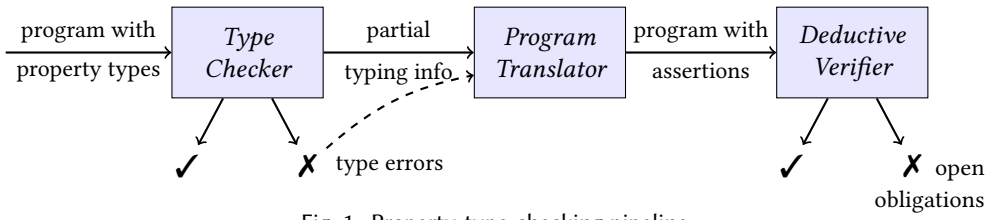

Fig. 1. Property-type-checking pipeline.

sophisticated conditions which are beyond the capabilities of a type checker. While type systems can help developers reduce these misses (so-called false positives), they cannot entirely remove them. Tools for deductive verification, on the other hand, can prove the properties behind a type system (non-nullness in the example) more often than a typical type checker can. Unfortunately, this increase in completeness comes at a price and they do not scale as well as type systems. In particular, they may require a significant specification and verification overhead from the developer.

This work presents a bridge between the two formal methods by combining the precision of deductive verification with the scalability of type systems. The idea is that the developer uses advanced type systems in their program to establish formal guarantees. The thus type-enriched program can then be checked by an automatic, well-scaling type checker. Only those remaining few false positives that could not be discharged by the type checker are then presented to a program verification tool which can use its full reasoning power to discharge the remaining formal obligations using deductive reasoning.

The combination of the two techniques is achieved by having the type checkers emit a transformed program in which all those typing constraints which they could not automatically prove to hold are transformed into formal specifications in the program (in the form of `assert` clauses) which are then to be proved by a deductive verifier. Moreover, those annotations that have been successfully checked by the type checkers are used as additional assumptions by the program verification tool to enhance its reach.

The workflow of the approach is sketched in Figure 1: The type checker may be able to confirm all property types (✓), but it may also terminate reporting typing errors (✗). The reported errors are passed on to the program translator, which encodes them as specification clauses and presents the thus instrumented program to the deductive verifier. Here, the remaining cases can either all be closed (✓) thus ensuring type safety. If a proof obligation remains unproven (✗), this may indicate an actual problem in the program or may still be a spurious type error due to lack of reasoning power of the prover, or due to too weak formal specifications. Interactive theorem provers can be used to inspect and analyze unclosed proofs.

Property type systems. The type system used in the approach can be any *property type system*, a term we use to designate a kind of pluggable type system, i.e., a type system which enriches a language's existing *base type system* without changing the run-time semantics [Bracha 2004]. The pluggable type systems we consider work by providing a set of *qualifiers*, which can be used to annotate a base type and restrict the set of valid instances. For example, `String` is the type of all strings, while `@Length(5) String` with the qualifier `@Length(5)` might be defined as the type containing only those strings of length 5. We define *property qualifiers* to be qualifiers which are defined by logical formulas or Boolean expressions over the typed variable. For example, the qualifier `@NonNull` in a nullness type system can be defined by the Boolean expression `subject != null`.

Property types are thus a kind of refinement type. Refinement types were introduced by Freeman and Pfenning [1991]. A refinement type is a refinement of some base type whose instances fulfill

some property. This notion is very similar to dependent types, i.e., types which depend on program values. The term "refinement types" is also used for dependent type systems whose types are restricted in a way that makes the type system decidable [Vazou et al. 2014, 2.1]. The term "type refinements" is also used in imperative languages to refer to type systems in which base types are extended by subtypes whose instances fulfill some property [Mandelbaum et al. 2003] [Chin et al. 2005].

Chin et al. [2005] introduce a system for *semantic type qualifiers* which allows users to define such type refinements along with typing rules in a domain-specific language. Markstrum et al. [2010] introduce the JavaCOP framework, which applies this idea to Java. Our work is implemented using the *Checker Framework* [Dietl et al. 2011a; Papi et al. 2008], another framework for pluggable Java type systems based on type qualifiers.

Java Implementation. We introduce our novel approach thoroughly on a conceptual level, but also by means of a concrete implementation for the Java programming language. The approach has been implemented on top of two well-established formal techniques for Java: The considered type systems and checkers are based on the Checker Framework. In the Checker Framework, pluggable types are implemented by means of type qualifiers that can be used to annotate Java types. Most type qualifiers can be turned into property qualifiers by providing an appropriate Boolean expression (e.g. `subject != null` for `@NonNull`). To make them accessible to deductive verification tools, property qualifiers are translated to specification clauses in the formal specification language *Java Modeling Language* (*JML*) [Leavens et al. 2013] which is the de facto standard for formal Java specifications. For the deductive verification, we use *KeY* [Ahrendt et al. 2016], a powerful deductive verification engine for JML-annotated Java code which supports both automatic and interactive verification, and the extended static checking facility of OpenJML [Cok 2011], a tool suite for verifying and run-time checking JML-annotated Java programs.

Contributions. The main contributions of this paper are:

- A general definition and formalization of a property type system for imperative languages. This comprises the outlines of typing rules for property types that make use of assume and assert statements in the language.
- A novel and sound approach to combine type checking and deductive verification by letting the type checker emit a program with some additional assertions. The deductive checker can close the correctness gaps that the type checker had to leave open by verifying these assertions. By using the deductive checker in addition to the type checker, we thus avoid false positives and are able to prove properties for more programs.
- A realization of the approach for Java using the Checker Framework for pluggable property types, and KeY and OpenJML for deductive verification.
  This realization includes a domain-specific language to define property types using Java expressions and a generic property type checker, again using the Checker Framework, to enforce these property types.
  The user can use this to define multiple pluggable property type systems. After the generic type checker has run once for each type system, the remaining properties of all type systems are compiled into a single JML specification.
- We demonstrate the usefulness and power of our approach in an evaluation in two parts: First, we extract some (simplified) code snippets from the Daikon Invariant Generator [Ernst et al. 2007] and discharge some false positives emitted by the Checker Framework's Nullness Checker. Second, we implement, specify, and verify a small case study.

```
boolean is_less_equal(@NonNull VarInfo v1, @NonNull VarInfo v2) {
  @Nullable Invariant inv = null; @Nullable PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    inv = instantiate(slice);
  }
  if (inv != null) {
    @SuppressWarnings("nullness")
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 2. Example of a false positive of the Checker Framework's Nullness Checker, which does not occur with our approach[1].

Overview. We start with a motivating example in Section 2, before Section 3 gives the theoretical definitions and theorems of property types and the translation of a property-typed program in a simple imperative language into a program with assertions and assumptions. Section 4 explains how this theoretical framework can be put into practice by explaining the realization of the pipeline consisting of a generic property type checker, a translation from property types to JML specifications, and the JML verification tools. We evaluate this framework in Section 5. In Section 6, we present related work on verification and types, and how it relates to our approach. Section 7 concludes the paper.

2 MOTIVATING EXAMPLE

Let us consider the source code excerpt in Figure 2 from the Daikon Invariant Generator [Ernst et al. 2007]. Daikon uses the Nullness Checker of the Checker Framework to ensure that null-pointer exceptions (NPEs) cannot be thrown at run time. We simplified the excerpt and made all relevant type qualifiers explicit. The `@Nullable` qualifier on a variable expresses that this variable might be `null`, while the `@NonNull` qualifier excludes `null` as a possible value. Whenever no explicit qualifier is given for a variable declaration, `@NonNull` is implicitly assumed by default for fields, parameters, and return types, and `@Nullable` for local variables.

Daikon contains approximately 60,000 lines of annotated Java code, excluding comments. Of those, 1129 contain explicit nullness annotations and there are 198 `@SuppressWarnings` annotations or `castNonNull` method calls to suppress false positive warnings from the Nullness Checker.

For this particular excerpt, the Nullness Checker detects a possible NPE for the method invocation on the `slice` variable. This is a false positive, because the implementation ensures that if the variable `inv` is non-null, the variable `slice` is also non-null. Therefore, the guarding `if`-statement of the first `return` statement prevents an NPE from being raised by the method invocation. While the Nullness Checker does have some flow-sensitive type refinement rules that allow it to refine static types in a flow-sensitive fashion, in this case, this refinement only affects the variable `inv`, which is refined from `@Nullable` to `@NonNull` inside the scope of the `if (inv != null)` statement.

[1]See the original file at https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/PptTopLevel.java#L2087

```
//@ requires v1 != null && v2 != null;
boolean is_less_equal(VarInfo v1, VarInfo v2) {
  //@ assume v1 != null && v2 != null;
  Invariant inv = null; PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    //@ assume slice != null;
    inv = instantiate(slice);
  }
  if (inv != null) {
    //@ assume inv != null;
    //@ assert slice != null;
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 3. (Simplified) translation of the example.

Our approach allows us to use a deductive verification tool to avoid reporting the false positive. To achieve this, we associate each type qualifier A with a property $Prop_A(subject)$, in which *subject* is a placeholder for the typed value. This property connects the intention of the type qualifier to its semantic meaning to be used in the verification process. For example, the property for `@NonNull` is $Prop_{\texttt{NonNull}}$ = `subject != null` and for `@Nullable`, it is $Prop_{\texttt{Nullable}}$ = `true`. Moreover, since $Prop_{\texttt{NonNull}}$ implies $Prop_{\texttt{Nullable}}$, any value of the former type is also a value of the latter type and any type qualified with `@NonNull` is a subtype of that type qualified with `@Nullable`. In general, `@`A T can only be a subtype of `@`B T if $Prop_A$ implies $Prop_B$.

Following the approach sketched in Figure 1, the next step is the translation of the program and the information of the type checker into a formal specification that can be read by the verification engine. The translation of our example is seen in Figure 3. It is similar to the code in Figure 2, but instead of the type qualifiers it contains JML specification clauses (in lines beginning with `//@`).

The `requires` clause states a method precondition, which must hold whenever the method is called. In our example, the precondition expresses the fact that both parameters are `@NonNull`. The `assert` statements state properties which must hold whenever that point in the program is reached, while the `assume` statements state properties which are assumed to hold without proof whenever the statement is reached. The assume statements are added to exploit the type information already established by the type checker.

Hence, all of the facts that the type checker was able to show are translated into `assume` statements, while only the one fact it could not show — that `slice` is not null before the method call — is translated into an `assert` statement. If the program satisfies all assertions, we have proven that all type errors are indeed false positives. If the assertions cannot be proven, the program and/or specification are likely incorrect, though it may be that the deductive checker is simply not powerful enough.

In our case, the specification in Figure 3 can be shown to be correct by OpenJML or by KeY's automatic mode.

$$program ::= function^+$$

$$function ::= T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\ \{\ s\ \}$$

$s ::= x = l$	literal assignment
$\mid\ x_0 = x_1$	variable assignment
$\mid\ x_0 = f(x_1, \ldots, x_n)$	function call
$\mid\ s\ ;\ s$	composition
$\mid\ assert\ p$	assertion
$\mid\ assume\ p$	assumption

Fig. 4. Program syntax of TSimp and its variants, where T is either a base type or a property type, x_i are variable identifiers, l is a literal, and p is a formula over the signature Σ as defined in Section 3.2.

$$\frac{l \in literals(T) \qquad T \preceq \Gamma(x)}{\Gamma \vdash x = l}\ \text{wt-assign-lit} \qquad \frac{\Gamma(x_1) \preceq \Gamma(x_0)}{\Gamma \vdash x_0 = x_1}\ \text{wt-assign-var}$$

$$\frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1\ ;\ s_2}\ \text{wt-compose} \qquad \frac{sig(f) = (T_0, T_1\ y_1, \ldots, T_n\ y_n) \quad T_0 \preceq \Gamma(x) \qquad \forall i \in [1..n].\ \Gamma(x_i) \preceq T_i}{\Gamma \vdash x = f(x_1, \ldots, x_n)}\ \text{wt-call}$$

Fig. 5. Base type rules.

3 PROPERTY TYPE SYSTEMS

In this section we explain the concepts behind the combination of property types and deductive verification. Property types in this context refine the existing type system of an underlying programming language with additional property qualifiers.

In Section 3.1, we introduce the simple imperative language TSimp. In Section 3.2, we extend TSimp's basic type system by introducing property types, yielding the new language PTSimp. In Section 3.3, we define a translation injecting assertions into PTSimp programs, yielding TSimpAssert programs. Validity of these assertions ensures that every property-typed variable respects its property. This translation can be optimized by using the results of existing type checkers. In Section 3.4, we show how the translation can be modified such that only those type properties whose validity cannot be established by a type checker have to be translated as assertions. Properties already established by the checker are instead translated as assumptions. Section 3.5 concludes this section by giving a short overview of what kinds of type checkers this approach can be applied to and how.

3.1 TSimp: A Simple Typed Programming Language

We formalize our notion of property type systems using the simple imperative programming language TSimp given by the grammar in Figure 4, the typing rules in Figure 5, and the big-step operational semantics in Figure 6. The basic statements in TSimp are assignments and function calls. Explicit assertions and assumptions are not used in this section. They are introduced in Section 3.3.

Notably, there are no control flow structures like *if* and *while*. We use TSimp (and later PTSimp) to illustrate how we can insert assertions into a program with property types and how the validity

$$\frac{}{\sigma, x \;=\; l \rightsquigarrow \sigma[x \mapsto I(l)]}\ \text{sem-assign-lit} \qquad \frac{}{\sigma, x_0 \;=\; x_1 \rightsquigarrow \sigma[x_0 \mapsto \sigma(x_1)]}\ \text{sem-assign-var}$$

$$\frac{\sigma, s_1 \rightsquigarrow \sigma' \quad \sigma', s_2 \rightsquigarrow \sigma''}{\sigma, s_1 \;;\; s_2 \rightsquigarrow \sigma''}\ \text{sem-compose} \qquad \frac{sig(f) = (T_0, T_1\ y_1, \ldots, T_n\ y_n) \qquad body(f) = s \quad \sigma_0[y_1 \mapsto \sigma(x_1), \ldots, y_n \mapsto \sigma(x_n)], s \rightsquigarrow \sigma'}{\sigma, x \;=\; f(x_1, \ldots, x_n) \rightsquigarrow \sigma[x \mapsto \sigma'(result)]}\ \text{sem-call}$$

Fig. 6. Operational semantics.

of those assertions allows us to conclude the correctness of the program. We only add assertions before and after assignments and function calls. Thus, control flow structures could be inserted into the language without significantly changing any of the material presented hereafter. To keep TSimp as simple as possible, we leave them out.

There are no variable declarations either. The only variables that can be used inside a function are the function's parameters and the variable *result*. In addition, there is a temporary variable $temp_T$ for every type T. These temporary variables may not be used by TSimp programs; they are used later for the translation.

We assume a non-empty set BT of *base types* with a partial order $\preceq$ that defines the subtype relation on BT. For every base type $T \in BT$ the set $literals(T)$ of literals of T is disjoint from the set of literals of all other base types.

We define a *domain* D to be some non-empty set. We also define the *interpretation* I to be a function which maps every literal to a value in D and every type to a subset of D such that (1) $T_1 \preceq T_2$ implies $I(T_1) \subseteq I(T_2)$ for any types T_1, T_2, and (2) $l \in literals(T_1)$ implies $I(l) \in I(T_1)$ for any literal l.

A TSimp program is a set of program functions according to Figure 4. The program function $T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\ \{\ s\ \}$ has the signature $sig(f) = (T_0, T_1\ x_1, \ldots, T_n\ x_n)$ with return type T_0 and parameter types $T_1, \ldots, T_n$, and a body $body(f) = s$.

Type contexts and program states are used in the typing rules and semantics for TSimp, respectively. In the following, the placeholders x and f stand for arbitrary identifiers. For a partial function F we define the updated function $F[x_0 \mapsto y](x) =$ if $x = x_0$ then y else $F(x)$, as the function which maps x_0 to y and agrees with F on all other arguments.

Definition 3.1 (Type contexts). A type context Γ is a partial function which maps variable names to types in BT. We further define Γ_0 as the context $\{temp_T \mapsto T \mid T \in BT\}$, i.e., we have a variable $temp_T$ for every type T. These temporary variables must not be used in the program itself. They are only used by the translation defined in Section 3.3. Given a program function $T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\{\ s\ \}$, we define $\Gamma_f := \Gamma_0[result \mapsto T_0, x_1 \mapsto T_1, \ldots, x_n \mapsto T_n]$, the *type context for* f.

Definition 3.2 (Program states). A program state σ is a partial function which maps variable names to D. σ_0 is the program state whose domain is empty.

Figure 5 shows the typing rules for statements. A judgment of the form $\Gamma \vdash s$ means that s is well-typed in the context Γ. This allows us to define when a program function is well-typed.

Definition 3.3 (Well-typed functions). A program function $T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\{\ s\ \}$ is well-typed if (1) $\Gamma_f \vdash s$, (2) f does not call itself, not even indirectly.

Lastly, we define a *conformant state* as a state in which every initialized variable points to a value of the correct type. We define the notion of *valid starting states* for program functions.

Definition 3.4 (Conformant states and valid starting states). We say that a state σ *conforms to* Γ if (1) $dom(\sigma) \subseteq dom(\Gamma)$, and (2) $\forall x \in dom(\sigma).\ \sigma(x) \in I(\Gamma(x))$.

We call σ a *valid starting state* for f if it conforms to Γ_f, $dom(\sigma) = \{result, x_1, \ldots, x_n\}$, and $\sigma(result) = default$ where the x_i are f's parameters and $default \in D$ is some object in the domain which is the same for every program function.

The semantics of a well-typed program function $T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\{\ s\ \}$ is defined as the function whose domain is the set of all valid starting states, and which maps every such state σ to the state σ' where $\sigma, s \rightsquigarrow \sigma'$ according to the big-step operational semantics in Figure 6.

Because of the simplicity of the semantics rules, we do not prove the preservation and progress properties for TSimp.

3.2 Property Types

The type system of TSimp will now be refined by introducing the possibility to annotate types with type qualifiers.

Definition 3.5 (Property qualifiers). A *property qualifier* is a name A associated with a subject base type $T_A \in BT$, and a property $Prop_A$.

A property qualifier A can be used to annotate a base type $T \in BT$ yielding a *property type* $@A\ T$. The intention of this property type is that a typed element is both known to be of type T and to satisfy property $Prop_A$ of A. The property $Prop_A$ is a formula with only one free variable, which we call *subject*. When a property qualifier occurs in a program, *subject* is bound to the typed variable. We write $Prop_A(v)$ for the formula which results from substituting v for *subject* in $Prop_A$.

Thus, the original types BT are still the skeleton for the type system, but the additional property qualifiers introduce a more fine-grained subtype hierarchy. In this property type framework, the properties do not occur in the type designators (and, thus, not in the program) themselves, but appear in the definition of the qualifiers only and are then referred to by name from the program.

We assume that properties are specified in first-order predicate logic using a fixed structure (Σ, D, I). Here, Σ is a signature, which for every type $T \in BT$ includes every literal $l \in literals(T)$ as a constant symbol and every valid variable name as a variable symbol. D is the same domain and I the same interpretation used in the preceding section. We assume that for every value $v \in D$, there exists a constant symbol $c \in \Sigma$ such that $I(c) = v$. In the following, we use c and v interchangeably.

To allow for more flexible type systems, property qualifiers may also take arguments.

Definition 3.6 (Parametrizable property qualifiers). A *parametrizable property qualifier* additionally has a tuple $(p_1, \ldots, p_n)$ of formal parameter names, a tuple $(T_1, \ldots, T_n)$ of parameter types, and a well-formedness condition Wf_A on the formal parameters.

To be used as a property qualifier, a parametrizable qualifier requires a tuple of actual parameters $(v_1, \ldots, v_n)$ that instantiate its formal parameters. Every actual parameter must be a literal of the correct type, i.e., $\forall i \in [1..n].\ \exists T \in BT.\ v_i \in literals(T) \wedge T \preceq T_i$. The property of a parametrizable qualifier may contain the parameters as free variables in addition to the subject. The well-formedness condition may contain the parameters as free variables, but not the subject. We write $A(v_1, \ldots, v_n)$, or $A(vs)$ for short, to denote the instantiation. We also write $Prop_A(v, vs)$ for the formula which results from substituting v for *subject* and v_i for p_i in $Prop_A$. Analogously, we write $Wf_A(vs)$ for the formula which results from substituting v_i for p_i in Wf_A. From now on, we consider non-parametrizable qualifiers A to be parametrizable qualifiers with $n = 0$ and $Wf_A = true$.

Not every construct $@A(vs)\ T$ makes an admissible property type in programs. The following conditions must be met:

(1) The target type T must be a subtype of A's subject type, i.e., $T \preceq T_A$ must hold.

(2) The well-formedness condition $Wf_A(vs)$ must hold. Since $Wf_A(vs)$ contains no free variables, the evaluation only depends on the structure (Σ, D, I). In particular, it does not depend on the program state.

The introduction of property types induces a new property type hierarchy based on the existing base type hierarchy. It is induced by the following subtyping definition for two property types $@A(as)\ T$ and $@B(bs)\ U$:

Definition 3.7 (Property subtyping). Given the relation $\preceq$ on base types, we define a relation $\preceq$ on property types by:

$$@A(as)\ T \preceq @B(bs)\ U \quad :\Longleftrightarrow \quad T \preceq U \text{ and } T_A \preceq T_B \text{ and}$$
$$\forall subject : T_A.\ Prop_A(subject, as) \rightarrow Prop_B(subject, bs) \text{ is valid.}$$

Note that since we only allow literals as qualifier parameters, we do not support dependent types. Appendix A[2] gives an overview of how we can include dependent property types by allowing variables as qualifier parameters.

Example 3.8 (Integer intervals). Assume that there is a type *int* with literals 0, 1, −1, 2, −2, ..., and that our logic includes symbols $\geq$ and $\leq$ with the usual interpretation. The property qualifier *NonNegative* has *int* as its subject type and the property $Prop_{NonNegative}$ is $subject \geq 0$. A further, parametrizable, qualifier *Interval* also has *int* as its subject type, two *int* parameters a and b, and the property $Prop_{Interval}$ is $a \leq subject \leq b$. The well-formedness condition $Wf_{Interval}$ could read $a \leq b$ if we want to disallow empty interval types.

We can observe that $@Interval(1, 2)$ *int* is a subtype of $@Interval(0, 3)$ *int* since every integer contained in the former interval is also contained in the latter. On the other hand, $@Interval(2, 1)$ *int* is not even a valid property type, since the well-formedness condition is violated.

The programming language PTSimp is the extension of TSimp such that every type in a program is a qualified property type. We consider type contexts Γ to map variables to property types instead of base types. The temporary variables $temp_T$ are mapped to $@true\ T$ where $Prop_{true} = true$. We also adapt the notion of conformant states to PTSimp such that in a conformant state, not only does every variable point to a value of the correct base type, but that value also satisfies the type property.

Definition 3.9 (Conformant states for PTSimp). A state σ conforms to a context Γ if (1) $dom(\sigma) \subseteq dom(\Gamma)$, (2) $\forall x \in dom(\sigma).\ \Gamma(x) = @A(vs)\ T \implies \sigma(x) \in I(T)$, and (3) $\forall x \in dom(\sigma).\ \Gamma(x) = @A(vs)\ T \implies Prop_A(\sigma(x), vs)$

We do not adapt the typing rules from Figure 5 for PTSimp, however. This is because defining subtype relations only via logical implications would require the type checker to be able to reason about these implications. In Section 3.4, we show how we can define a practical type hierarchy which approximates the hierarchy induced by the implications. For now, we define a PTSimp program to be well-typed if it is well-typed as a TSimp program in the base type system and if assertions inserted into the program hold in every reachable state. This is explained in detail in the following section.

3.3 Checking Property Type Systems

We define the correctness of a PTSimp program by inserting assertions and calling it correct if these assertions hold. For this purpose, we make use of the assertion and assumption statements from Figure 4. Their typing rules and semantics are shown in Figures 7 and 8, yielding a new

[2]The appendices are available in the supplementary material to this paper, found at https://doi.org/10.1145/3485520.

$$\frac{}{\Gamma \vdash \ \mathit{assert}\ p}\ \text{wt-assert} \qquad \frac{}{\Gamma \vdash \ \mathit{assume}\ p}\ \text{wt-assume}$$

Fig. 7. New base type rules for TSimpAssert.

$$\frac{\sigma \models p}{\sigma, \mathit{assert}\ p \rightsquigarrow \sigma}\ \text{sem-assert} \qquad \frac{\sigma \not\models p}{\sigma, \mathit{assert}\ p \rightsquigarrow \mathit{assertion\text{-}failure}}\ \text{sem-assert-fail}$$

$$\frac{\sigma \models p}{\sigma, \mathit{assume}\ p \rightsquigarrow \sigma}\ \text{sem-assume} \qquad \frac{\sigma \not\models p}{\sigma, \mathit{assume}\ p \rightsquigarrow \mathit{assumption\text{-}failure}}\ \text{sem-assume-fail}$$

$$\frac{\sigma \in \{\mathit{assertion\text{-}failure}, \mathit{assumption\text{-}failure}\}}{\sigma, s \rightsquigarrow \sigma}\ \text{sem-pass-failure}$$

$$\frac{\begin{array}{c}\mathit{sig}(f) = (T_0, T_1\ y_1, \ldots, T_n\ y_n) \qquad \mathit{body}(f) = s \\ \sigma_0[y_1 \mapsto \sigma(x_1), \ldots, y_n \mapsto \sigma(x_n)], s \rightsquigarrow \sigma' \\ \sigma' \in \{\mathit{assertion\text{-}failure}, \mathit{assumption\text{-}failure}\}\end{array}}{\sigma, x\ =\ f(x_1, \ldots, x_n) \rightsquigarrow \sigma'}\ \text{sem-call-failure}$$

Fig. 8. New semantics rules for TSimpAssert.

$$\frac{\Gamma(x) = @A(vs)\ \ T}{\Gamma, x\ =\ y \mapsto \mathit{assert}\ \mathit{Prop}_A(y, vs)\ ;\ x\ \widehat{=}\ y}\ \text{transl-assign}$$

$$\frac{\Gamma, s_1 \mapsto s_1' \qquad \Gamma, s_2 \mapsto s_2'}{\Gamma, s_1\ ;\ s_2 \mapsto s_1'\ ;\ s_2'}\ \text{transl-compose}$$

$$\frac{\mathit{sig}(f) = (@A_0(vs_0)\ \ T_0, @A_1(vs_1)\ \ T_1\ y_1, \ldots, @A_n(vs_n)\ \ T_n\ y_n) \qquad \Gamma(x) = @A(vs)\ \ T}{\Gamma, x\ =\ f(x_1, \ldots, x_n) \mapsto \left\{\begin{array}{c}\mathit{assert}\ \mathit{Prop}_{A_1}(x_1, vs_1); \ldots; \mathit{assert}\ \mathit{Prop}_{A_n}(x_n, vs_n); \\ \mathit{temp}_T\ \widehat{=}\ f(x_1, \ldots, x_n); \\ \mathit{assume}\ \mathit{Prop}_{A_0}(\mathit{temp}_T, vs_0); \mathit{assert}\ \mathit{Prop}_A(\mathit{temp}_T, vs); \\ x\ \widehat{=}\ \mathit{temp}_T\end{array}\right\}}\ \text{transl-call}$$

Fig. 9. Translation from PTSimp to TSimpAssert.

language TSimpAssert. In the figures, $\sigma \models p$ for a program state σ and a formula p means that p holds in σ. We also introduce two special end states *assertion-failure* and *assumption-failure*, which we define to never conform to any type context. In addition to what is shown in Figure 8, all rules in TSimpAssert's semantics except for sem-pass-failure have the implicit premiss $\sigma \notin \{\mathit{assertion\text{-}failure}, \mathit{assumption\text{-}failure}\}$.

Given a PTSimp program, we can then translate it into a TSimpAssert program according to the translation defined by Figure 9. $\Gamma, s \mapsto s'$ means that the statement s occurring in the context Γ is to be replaced by the statement s'. In addition to what is shown there, we add the statements

$$\mathit{assume}\ \mathit{Prop}_{A_1}(x_1, vs_1); \ldots; \mathit{assume}\ \mathit{Prop}_{A_n}(x_n, vs_n)$$

to the start of the body of every function with parameter names x_i and parameter types $@A_i(vs_i)\ \ T_i$. The symbol $\widehat{=}$ is equivalent to $=$, but it marks which statements have already been translated, and prevents rules from being applied twice on the same statement. The translation terminates when no more rules are applicable.

```
@Interval(0,21) int g(@Interval(0,12) int x) { ... }
@Interval(0,42) int f(@Interval(0,3) int x,
                      @Interval(0,1) int y, @Interval(2,3) int z) {
  x=1; x=y; x=z; result=g(x)
}
```

Fig. 10. An example of a PTSimp program.

```
int f(int x, int y, int z) {
  assume 0 ≤ x ≤ 3; assume 0 ≤ y ≤ 1; assume 2 ≤ z ≤ 3;
  assert 0 ≤ 1 ≤ 3; x=1;
  assert 0 ≤ y ≤ 3; x=y;
  assert 0 ≤ z ≤ 3; x=z;
  assert 0 ≤ x ≤ 12; temp_int=g(x); assume 0 ≤ temp_int ≤ 21;
  assert 0 ≤ temp_int ≤ 42; result=temp_int
}
```

Fig. 11. Translation of f from Figure 10 into TSimpAssert.

Example 3.10. For an example of how this works, we consider the PTSimp function f given in Figure 10, which uses the interval property types from Example 3.8.

Its TSimpAssert translation is shown in Figure 11. The assumptions at the start fail if the function is not called in a valid starting state, i.e., with parameters of the correct property types. The assertions before every assignment fail if the assignment would break the assigned variable's type property. The assertion before the function call to g fails if g would be called in an invalid starting state. The assumption after the function call to g states that the variable holding the call's result has g's return type; thus it can only fail if the called function g is not correct. And lastly, the assertion after the function call fails if the type of *result* in f is incompatible with g's return type. We use the temporary variable *temp_int* to ensure that g's result is not assigned to *result* if this would violate *result*'s type property.

The assert statements are injected into the program for the purpose of a static analysis to discharge. However, they could in principle also be used for run-time assertion checking if one wants to resort to run-time verification rather than static analysis.

We can now define when such a translated program is correct.

Definition 3.11 (Correctness). A TSimpAssert function $T_0\ f(T_1\ x_1, \ldots, T_n\ x_n)\{\ s\ \}$ is *correct* if (1) it is well-typed as a TSimp function in the base type system (ignoring all assertions and assumptions), (2) all functions called by it are also correct, and (3) there exists no valid starting state σ from which an *assertion-failure* state can be reached. Reachability is defined below.

This definition only rules out assertion failures, not assumption failures. The fact than no assumption failures can occur in a correct function is shown in Theorem 3.14.

Definition 3.12 (Reachable states). We say that a state σ' is *reachable from* f if there exists a valid start state σ and a statement s such that f's body is either equal to s or equal to $s; t$ for some statement t and such that $\sigma, s \rightsquigarrow \sigma'$ (We consider the statement composition operator ; to be associative, i.e., $(s_1; s_2); s_3 = s_1; (s_2; s_3)$). In this case, we also say that σ' is *reachable from* σ *via* s.

We now show that the translation defined above gives us what we want: When a correct function is called in a valid starting state, then every state reachable from that starting state conforms to the function's type context. In other words, every variable always fulfills its type property.

We first prove that all states reachable from a correct function are conformant.

Theorem 3.13 (Preservation). *Let f be a function with body s such that its translation g with body t is correct. Let σ be a valid starting state for g. Let σ' be reachable from σ.*
Then either $\sigma' =$ assumption-failure or σ' conforms to the type context Γ_f for f.

Proof. By induction over the rules for $\rightsquigarrow$; see Figures 6 and 8. The only interesting cases are sem-assign-lit, sem-assign-var, and sem-call(-failure). For brevity's sake, we only prove conformance property (3), i.e., $\forall x \in dom(\sigma').\ \Gamma'(x) = @A(vs)\ T \implies Prop_A(\sigma'(x), vs)$.

For sem-assign-lit, we know that the translation inserted an assertion $assert\ Prop_A(l, vs)$ before the assignment $x = l$. If $Prop_A(l, vs)$ did not hold, the assertion would have ended in *assertion-failure*, which contradicts g's correctness. Rule sem-assign-var can be proven analogously. Rule sem-call can also be proven analogously. If σ' did not conform to Γ_f, then the assertion before the assignment to x would fail. For sem-call-failure, note that a correct function only calls other correct functions and that the assertions placed before a function call ensure that the function is called in a valid starting state. Since our language contains no recursion, we can assume that this theorem holds for any function called by f. Thus, any such called function cannot lead to an assertion failure.

It remains to be observed that the type property predicate $Prop_A$ for variable x only depends on the value $\sigma(x)$. In particular, it does not depend on the value of any other variable. Thus the property is guaranteed to be preserved until the next assignment to x. □

Having shown this, we now show that, if a function is correct, it does not end in an assumption/assertion failure, and it does in fact end, i.e., it has a terminal state.

Theorem 3.14 (Progress). *Let s, t, f, σ be defined as in Theorem 3.13. Let either t_1 or $t_1; t_2$ be equal to t. Then there exists σ' such that $\sigma, t_1 \rightsquigarrow \sigma'$ and $\sigma' \notin \{$assertion-failure, assumption-failure$\}$.*

Proof. We first show that there exists σ' such that $\sigma, t_1 \rightsquigarrow \sigma'$:

Since σ is a valid starting state, it cannot be in $\{$*assertion-failure*, *assumption-failure*$\}$. The only rules that have any premisses besides $\sigma \notin \{$*assertion-failure*, *assumption-failure*$\}$ are sem-compose and sem-call(-failure). The premisses for sem-compose can obviously always be fulfilled by instantiating them properly. For sem-call(-failure), remember that correct functions can only call other correct functions. Let h be the function being called. If h were called in an invalid starting state, the assertions placed before the call would fail, which contradicts f's correctness. Thus, h is always called with a valid starting state. Because our language does not have recursion, we can assume that this theorem holds for h without being circular. Therefore, h has an end state $\sigma' \notin \{$*assertion-failure*, *assumption-failure*$\}$ such that $result \in dom(\sigma')$, which means that all premisses for sem-call are fulfilled.

A consequence of f's correctness is that $\sigma' \neq$ *assertion-failure*.

Assumptions only occur at the start of a function body and after a function call. The assumptions at the start of the body of f cannot fail because the starting state σ is valid for f. The assumptions at the start of the body of any function called by f cannot fail either. As mentioned before, the assertions before a function call ensure that functions are only ever called in valid starting states. The assumption after a call to any function h called by f cannot fail either. If it did, the value returned by h would not have h's return type, which contradicts h's correctness. □

$$\frac{\Gamma(x) = @B(bs)\ U \qquad @B(bs)\ U <: @A(as)\ T}{assert\ Prop_A(x, as) \mapsto assume\ Prop_A(x, as)}\ \text{transl-enh-var} \qquad \frac{\models Prop_A(l, as)}{assert\ Prop_A(l, as); s \mapsto s}\ \text{transl-enh-lit}$$

$$\frac{\begin{array}{c} temp_T \text{ was last assigned the result of a call to } g \\ sig(g) = (@B(bs)\ U, \ldots) \qquad @B(bs)\ U <: @A(as)\ T \end{array}}{assert\ Prop_A(temp_T, as) \mapsto assume\ Prop_A(temp_T, as)}\ \text{transl-enh-temp-var}$$

Fig. 12. Enhanced translation rules for TSimpAssert.

3.4 Combining Deductive Verification and Type Checking

We now have a way of translating PTSimp programs into a format that can be checked by a deductive verification tool. However, we do not have a way of combining that deductive verification tool with a type checker: The deductive verification tool does not know whether an assertion already follows from the type system used by the type checker; it has to prove all assertions in the program.

In fact, the property type hierarchy given by Definition 3.7 is not at all practical to use in a type checker. To check whether two property types $@A\ T$, $@B\ U$ are subtypes, we have to check whether the implication $Prop_A \longrightarrow Prop_B$ is universally valid. Depending on the logical structure we use for the *Prop*s, this problem may be undecidable or, in many cases, beyond the capabilities of most type checkers.

Instead of working with the semantical subtype relationship, in this section, we operate on user-defined *property qualifier hierarchies* that are lifted to type hierarchies. These hierarchies have the advantage that they do not rely on the decidability of logical implications in general and that type checkers are decidable and can actually be implemented.

Definition 3.15 (Property qualifier hierarchy). Let PQ be the set of all instantiated property qualifiers $A(vs)$ that satisfy $Wf_A(vs)$. We call a partial order $<:$ over PQ a *property qualifier hierarchy* if it fulfills the following conditions:

(1) The order is compatible with the base-type relation between the subject types, i.e., $A(as) <: B(bs)$ implies $T_A \preceq T_B$ for any two qualifiers $A(as), B(bs)$.
(2) The order is compatible with the implications between the properties, i.e., $A(as) <: B(bs)$ implies that $\forall subject : T_A.\ Prop_A(subject, as) \longrightarrow Prop_B(subject, bs)$ is valid.

Any property qualifier hierarchy can be lifted to a hierarchy between property types via

$$@A(as)\ T <: @B(bs)\ U :\Longleftrightarrow A(as) <: B(bs) \text{ and } T \preceq U$$

for any pair of instantiated qualifiers $A(as)$, $B(bs)$ and base types $T, U \in BT$. Definition 3.15 ensures that $@A(as)\ T <: @B(bs)\ U$ implies $@A(as)\ T \preceq @B(bs)\ U$.

Given a property qualifier hierarchy, we obtain an *enhanced translation* by applying the rules given in Figure 12 to a TSimpAssert program.

The rules transl-enh-var and transl-enh-temp-var allow us to replace those assertions that are covered by a type checker for $<:$ by assumptions. The reason we replace assertions by assumptions instead of just removing them is that the assumptions allow the deductive verification tool to use the knowledge gained by the type checker, which might make its proofs easier. An example for when these assumptions come in useful can be seen in the case study in Section 5.

In the rule transl-enh-lit, $\models p$ means that the closed formula p holds under the fixed interpretation I used in the preceding sections and that this fact can be shown by the type checker. This rule allows us to remove type property assertions containing valid formulas without free variables. In this case,

```
int f(int x, int y, int z) {
  assume 0 ≤ x ≤ 3; assume 0 ≤ y ≤ 1; assume 2 ≤ z ≤ 3;
  x=1;
  assume 0 ≤ y ≤ 3; x=y;
  assert 0 ≤ z ≤ 3; x=z;
  assume 0 ≤ x ≤ 12; temp_int=g(x); assume 0 ≤ temp_int ≤ 21;
  assume 0 ≤ temp_int ≤ 42; result=temp_int
}
```

Fig. 13. Enhanced translation of f from Figure 10.

we remove the assertion instead of replacing it since such tautological formulas do not provide new constraints that could be used by the deductive verifier, nor are they helpful lemmas. Like the original translation, the enhanced translation terminates when no more rules are applicable.

Example 3.16. Consider again the program from Example 3.10 given in Figure 10 and its translation in Figure 11.

Suppose that we have a type hierarchy $<:$ which includes all relations of the form $@Interval\ int <: @Interval\ int$ where $a \geq c \wedge b \leq d$ except for the relation $@Interval(2,3)\ int <: @Interval(0,3)\ int$.

Given a type checker for this hierarchy, we can translate the function f as shown in Figure 13:

The assertion containing the trivially true closed formula $0 \leq 1 \leq 3$ before the assignment $x = 1$ has been removed. All of the assertions which follow from relations in $<:$ and can thus be verified by the checker have been turned into assumptions. Only the single assertion which is not trivial and does not follow from such a relation remains and has to be shown by the deductive verification tool.

We now re-prove Theorems 3.13 and 3.14 using this enhanced translation. These proofs tell us that the enhanced translation is just as good as the original translation in that it still ensures that in a correct function, every variable always fulfills its type property.

Theorem 3.17 (Preservation, II). *Let f be a function with body s such that its enhanced translation g with body t is correct. Let σ be a valid starting state for g. Let σ' be reachable from σ.*
Then either $\sigma' =$ assumption-failure or σ' conforms to the type context Γ_f for f.

Theorem 3.18 (Progress, II). *Let s, t, f, σ be defined as in Theorem 3.17. Let either t_1 or $t_1; t_2$ be equal to t. Then there exists σ' such that $\sigma, t_1 \rightsquigarrow \sigma'$ and $\sigma' \notin \{$assertion-failure, assumption-failure$\}$.*

Proof. Let g_o with body t_o be the original translation for f.

Then for both theorems, it suffices to show that if g is correct, g_o is also correct, i.e., if no *assertion-failure* is reachable from g, then no *assertion-failure* is reachable from g_o. The claims then follow from the original theorems.

To show this, it suffices to show that no assertion in g_o that can be removed or replaced by an assumption in g can lead to an *assertion-failure*. This follows from the enhanced translation rules: The rule transl-enh-lit only removes assertions containing valid closed formulas, i.e., assertions that can never fail. Suppose that the rule transl-enh-var replaces a possibly failing assertion with an assumption. Then the assertion/assumption is executed in a reachable state σ^* in which a variable x does not respect its type property $Prop_A(x, as)$. But for such a state σ^* to be reachable, there must have been a failing assertion before an assignment to x which was removed, which we have shown to be impossible. Suppose that the rule transl-enh-temp-var replaces a possibly failing assertion with an assumption. Then the assertion/assumption is executed in a reachable state σ^* in which a

temporary variable $temp_T$ does not respect its type property $Prop_A(temp_T, as)$. Temporary variables are only assigned in function calls. Thus, for such a state σ^* to be reachable, the function h whose result was assigned to $temp_T$ must have returned a result which does not have h's return type, which contradicts h's correctness. □

3.5 General and Dedicated Type Checkers

The approach presented in Section 3.4 can be applied to many different type checkers. Section 3.4 assumes the simplest possible checker, which given some property type hierarchy only checks a program for some simple subtyping rules.

When we bring this approach from TSimp to a real language like Java, then a more powerful type checker can obviously do much more than just enforcing subtyping rules. Type refinement, type inference, and other features allow us to discharge many more assertions and only leave the really tricky cases to a deductive verification tool.

There are two general possibilities for what kind of type checker to use:

(1) Use a general-purpose property-agnostic type checker that can be instantiated with any property type hierarchy but is not very powerful and probably not useful without being combined with a deductive verification tool.
This allows us to create custom property type hierarchies for application-specific properties. The only code that needs to be written are the definitions of the type qualifiers and the qualifier hierarchy, which can be given as input to the general checker.
An example for this is the checker presented in Section 4.1. This checker was developed alongside this paper using the Checker Framework. It can be instantiated with any property type hierarchy and is essentially the Java implementation of what is shown in Section 3.4.

(2) Use an optimized special-purpose checker designed for a particular property, like the Checker Framework's Nullness Checker.
This allows us to use deductive verification to add precision to an existing, established type hierarchy and checker, thus allowing it to verify more programs.
For that, the user must provide an appropriate property and well-formedness condition for every qualifier used by the checker. These must be chosen such that the checker can guarantee that in every reachable program state, every initialized variable satisfies its property, and that the well-formedness condition of every qualifier appearing in the program holds.
As mentioned in Section 2, the correct property for `@NonNull` would be `subject != null`.

It is possible to use multiple property type systems in the same program, some of which can be checked using a general-purpose checker and some using a special-purpose checker. There is one type checker invocation for every type system. The translations resulting from all of those checker invocations can easily be combined: Instead of, for example, adding a single assertion or assumption before an assignment, we add one for every type system.

4 VERIFYING PROPERTY TYPES IN JAVA

In this section, we explain how we implemented a property type system as introduced in Section 3 in Java. We implemented a generic property-agnostic checker as described in Section 3.5 using the Checker Framework. This checker takes a definition of a property qualifier hierarchy written in a domain-specific language and checks for the usual subtype and well-formedness properties. This is explained in more detail in Section 4.1.

This checker includes an algorithm that translates a program using property qualifiers to a program annotated with JML specifications. This algorithm works much like the one described in

```
annotation MinLength(int min)
    java.util.List # T_MinLength
    :<==> "subject.size() >= min" # Prop_MinLength
    for "min >= 0"; # Wf_MinLength
annotation UnknownLength() any :<==> "true" for "true";
relation MinLength(min0) <: MinLength(min1) :<==> "min0 >= min1";
relation MinLength(min) <: UnknownLength() :<==> "true";
```

Fig. 14. Example qualifier hierarchy definition.

Section 3.3 and Section 3.4. Section 4.2 explains how this theoretical algorithm can be applied to Java and JML.

The JML-annotated code can then be given to a deductive verifier. The pipeline works just like the one in Figure 1, though in addition, it includes two preliminary steps: First, the user must ensure that the property qualifier hierarchy defined by them satisfies the requirements defined in Section 3.4. Second, they must ensure that the Java program satisfies some constraints, which are discussed in Section 4.3.

4.1 The Generic Java Property Checker

The Java Property Checker is configured using a hierarchy definition. Figure 14 shows what such a hierarchy definition looks like. Everything after a hash symbol # is a comment. Every property qualifier is defined by a statement of the form `annotation` A T_A `:<==>` $Prop_A$ `for` $\mathtt{Wf}_A$`;`. The qualifier hierarchy is defined by statements of the form `relation` A `<:` B `:<==>` P where P is a Boolean expression which always terminates normally, and whose value only depends on A's and B's parameters. Then A `<:` B if and only if P evaluates to `true`.

The user must take care that their qualifier hierarchy is well-defined and satisfies Definition 3.15. These requirements are not checked by the current implementation.

Figure 14 defines a hierarchy of qualifiers `@MinLength(n)` where a list has the type `@MinLength(n) List` if its length is at least `n`. The qualifier `@UnknownLength` serves as a top element: It can apply to any base type (as opposed to `@MinLength` which can only apply to `Lists`) and its property is `true`. The Checker Framework requires the qualifier hierarchy to be a bounded lattice. To support this, the hierarchy definition language also has `meet` and `join` declarations with which to define the greatest lower bound and least upper bound of any pair of qualifiers. For brevity's sake, these are not included in Figure 14. These declarations are trusted, i.e., their correctness is not checked.

The type checker checks for the usual subtype properties that form the basis for almost all Checker Framework type systems and are provided for by the Checker Framework: Whenever Java would require the base types of two expressions to have a certain subtype relationship, the checker requires the property types to have that same relationship. The only exception is that the checker allows for the qualifiers on parameters of overriding methods to be contravariant (but as usual in Java, if the base types of the parameters differ, the method is overloaded instead of overridden). In addition, a primitive or string literal can be assigned to any variable whose type property it satisfies. This is implemented by compiling all type properties as Java methods and calling those methods with the literals as parameters.

4.2 The JML Translation Algorithm

Here, we present an implementation of an algorithm that takes a Java program which uses a property qualifier hierarchy as described in Section 3.4 and inserts JML specifications. We do not

```
T temp = e; /*@assert Prop_A(temp);*/ v = temp;
```

Fig. 15. JML translation of an assignment.

```
class C {
  @Length("5") List l0;
  C(@Length("5") List l) { helper(); this.l0 = l; l0.removeFirst(); }
  void helper() { @Length("5") List l = this.l0; }
}
```

Fig. 16. (Wrong) usage of property types in Java.

prove that Theorems 3.17 and 3.18 apply to the JML translation. Proof sketches to that effect can be found in the master's thesis by Lanzinger [2021].

The translation adds an assertion or assumption before every assignment, local variable or field declaration[3], and return statement. Methods and method calls are handled via contracts instead of assertions; this is explained below.

Because Java expressions may contain side-effects and JML expressions must not, we first evaluate the right-hand side of an assignment `v=e;` and save the result in a temporary variable. We then refer to that temporary variable in the assertion or assumption, and afterwards assign the temporary variable to the actual left-hand side variable v. The result of this translation is shown in Figure 15.

In Section 3, we also added assertions before every method call and an assumption after every method call. This approach cannot be exactly applied to Java because unlike in TSimp, in Java we do not only want to check that every method satisfies its own type signature; we also want to check that every method satisfies the type signature of the method it overrides, i.e., that its return type is covariant and its parameter types are contravariant.

We thus instead add a JML contract to every method and constructor, which contains the parameters' type properties as preconditions and the result's type properties as postconditions. JML's semantics require a method to satisfy the contracts of its parent method. This approach however leads to a different problem: Just as we replace assertions which have been proven by the Property Checker by assumptions, we want to replace preconditions which have been proven by the Property Checker by *free preconditions*, i.e., preconditions which can be assumed by the callee but do not have to be proven by the caller. However, a contract is specified once for each method; we cannot specify a separate contract for each method call. To solve this, we add a so-called *trampoline method* for every method and constructor m and replace every call to m with a call to m's trampoline. The trampoline method has an additional Boolean parameter b_i for every regular parameter x_i. Its contract includes every parameter x_i's type property twice: once as a regular precondition, which is used if b_i is false, and once as a free precondition, which is used otherwise.

The structure of trampoline methods, along with the rest of the JML translation algorithm is shown in Appendix B.

4.3 Constraints on Java Programs

Java is considerably more complex than the conceptual language TSimp. Many of the concepts extend naturally to full Java, but to ensure soundness some constraints are necessary. These are explained in this section.

[3] One cannot add assertions/assumptions before inline field initializations. The translation therefore moves such initializations to the beginning of the constructor(s).

Figure 16 contains a Java class using the property qualifier `@Length(value)` with the property `subject.size() == value`. It shows what can go wrong when one naïvely transplants the theoretical presentation into Java. The example has two serious issues:

(1) The assignment in `helper()` looks good in isolation, but actually `helper()` is called before `l0` has been initialized. This means that the local variable `l` is assigned the value `null`, which does not respect the necessary type property.
(2) While there is no invalid assignment, the call to `removeFirst()` breaks the type property of both `C::l0` and the parameter `l`.

We prevent these and some other problems by introducing the following constraints:

Initialization. We must take into account when objects are initialized. To that end, we define an object o to be initialized if all fields that are reachable from o fulfill their type property.

Summers and Müller [2011] introduce a pluggable type system for initialization which is implemented in the Checker Framework. There are three qualifiers `Initialized`, `UnderInitialization`, and `UnknownInitialization`, where references of a type `@Initialized` T are guaranteed to point to initialized objects[4]. The Property Checker's implementation is based on the Initialization Checker: It only considers type properties of objects annotated with `Initialized` and checks that these annotations are correct.

Immutability/monotonicity. We only allow variables referring to immutable objects to be annotated with property qualifiers[5]. This ensures that assignments to the subject's fields do not invalidate its type property. Potanin et al. [2012] give an overview over different notions of immutability in object-oriented languages. For our approach, *deep immutability* is necessary, i.e., fields of immutable objects may be neither reassigned nor mutated. This includes transitive fields, i.e., if `a` is immutable, then so are `a.b` and `a.b.c`.

Our implementation currently includes no immutability type system. Depending on the immutability type system used, the immutability checks could take place at different times. A *class-based* immutability system is one in which a variable's immutability can be determined only by its class. Such a system could be checked separately from the actual type checking, together with the constraints on properties outlined below: One would only have to iterate through the subject types T_A for every qualifier A and check that T_A is either primitive (and thus immutable) or an immutable class type. Non-class-based immutability systems on the other hand might have to be integrated into the Property Checker.

Actually, requiring immutability is an over-approximation: When using a dedicated type system for some specific properties instead of the general-purpose Property Checker checker, it is possible to allow some mutations as long as the type system ensures that all type properties are monotonic, i.e., an object can never be mutated in a way that would invalidate its type property.

The best example for this is a nullness type system: The value of the property $Prop_{NonNull}$ = `subject != null` is completely independent from the object pointed to by the subject, so objects pointed to by variables of a type `@`*NonNull* T could be allowed to be mutated more freely[6]. Things get more complicated when one also considers flow-sensitive type refinement.

[4] Summers and Müller [2011] actually define an object as initialized when all reachable fields are not null, not—as we do—when all reachable fields satisfy some arbitrary property, but as noted in Section 5.8 of their paper, their approach also supports more general invariants like our type properties.
[5] Actually, because in the Checker Framework every variable must be annotated, we allow qualifiers whose property is `true` even for mutable objects.
[6] Some restrictions are still necessary. Objects whose fields are non-null cannot be mutated completely freely without breaking the type properties of the fields.

Constraints on properties. In our implementation, properties $Prop_A$ and well-formedness conditions Wf_A are not logical predicates but Boolean expressions defined by the user for every qualifier. To ensure that the value of these expressions does not change, we require that

(1) parameters of qualifiers are of primitive or String type[7]. Java also allows class literals and arrays, which we exclude because they are not supported by the theoretical framework presented here.
(2) well-formedness conditions and properties only contain calls to pure (deterministic and side-effect free) methods.
(3) the value of a well-formedness condition depends only on its parameters.
(4) the value of a property depends only on its parameters, the subject, and any fields reachable from the subject.
(5) the evaluation of well-formedness conditions and properties always terminates normally.

For points (2) to (4), both the Checker Framework [Checker Framework developers 2020, 20] and JML [Leavens et al. 2013, 7.1.1.3] include ways to specify that a method is pure [Rudich et al. 2008] (via the `@Pure` annotation in the Checker Framework or the `/*@pure@*/` annotation in JML).

To specify and verify that an expression's value depends only on certain variables, one could either use KeY's information flow analysis [Ahrendt et al. 2016, Ch. 13] or, more simply, perform a conservative syntactical analysis (e.g., checking that no other variables are ever referred to). None of these options are currently a part of our implementation.

For technical and pragmatic reasons, we enforce some additional constraints as to what Java programs we allow. First, since the theoretical framework presented in the preceding sections is not equipped to deal with generic types, we only consider Java programs without generics. The implementation is currently limited to programs without generics, but the approach can be extended to also support generics. Although the support of generics in KeY is very limited, the Checker Framework and OpenJML support them. To support generics, we would have to introduce a way to allow a method's JML contract to depend on the instantiation of type variables. Second, we only allow property qualifiers to appear on variable (local variable, parameter, or field) declarations, on return types, and on constructor declarations. Qualifiers on `new` expressions, casts, and `instanceof` expressions are not supported by the framework presented here. Third, we do not allow property qualifiers[8] to appear on static fields. This is a constraint of the initialization type system, as it is generally not possible to say when a static field will be initialized [Summers and Müller 2011, 5.3]. Finally, the JML translation assumes that there are no assignments in nested expressions (e.g., an expression like `a + (b = c)` would be illegal)[9], that a variable in an inner scope never has the same name as a different variable in an outer scope, and that all fields and methods that are referred to in qualifier properties and well-formedness conditions are public[10].

[7] Actually, we only allow Strings which contain Java expressions that evaluate to a value of primitive or String type. This allows us to support dependent qualifiers (see Appendix A) by allowing users to use expressions whose value is not known at compile-time as parameters.
[8] Except for qualifiers whose property is `true`.
[9] Otherwise, we would not be able to add assertions/assumptions before the assignment. A more elegant translation implementation would automatically unroll such expressions.
[10] JML has the modifier `spec_public` for fields that should be public for the specification but have some other visibility for the actual program code [Leavens et al. 2013, 2.4]. Because our checker implementation does not use JML and sometimes inspects the code of the program being checked using Java's reflection API, we chose to have this requirement instead of implementing our own version of these modifiers.

```
if (invInfo.vars() != null) {
  @SuppressWarnings("nullness")
  @NonNull List var_perms = invInfo.var_permutations();
}
```

Fig. 17. Dependency between two return values[12].

5 EVALUATION

In this section, we present an evaluation in two parts. First, we show that our approach can be used to detect false positives in type checking by applying the JML translation to several code snippets taken (and made self-contained) from Daikon [Ernst et al. 2007]. As a reminder, Daikon contains 1129 explicit nullness annotations and there are 198 `@SuppressWarnings` annotations or `castNonNull` method calls. Second, we demonstrate our approach in a case study in the form of a small program that was annotated with property qualifiers, then type checked using the generic property type checker, and then the JML translation is checked by KeY and OpenJML. This case study shows that our approach reduces the verification overhead significantly as opposed to just using a deductive verification tool like KeY.

False positives in Daikon. There is already one example for a false positive result by the Checker Framework's Nullness Checker in Section 2.

We extracted a few more simplified examples from Daikon and used the JML translation to discharge the type checker's false positive results; this works with both KeY and OpenJML. Like the example in Section 2, the other examples are also mostly based on dependencies between different variables.

For example, in Figure 17, `invInfo.var_permutations()` cannot return `null` if `invInfo.vars()` is not `null`. We verified the method containing this snippet by adding JML clauses to those two methods as seen in Figure 18.

The Checker Framework supports limited method pre- and postconditions via annotations like `@EnsuresQualifierIf(result=true, expression="#1", qualifier=Odd.class)`[11], which states that if the result of the annotated method is true, the first parameter will be set to an odd number. These annotations are currently not supported by the Property Checker. Instead, we use the annotation `@JMLClause` to inject additional JML clauses into the contracts generated by the translator. This allows us to use the property that `invInfo.var_permutations()` cannot return `null` if `invInfo.vars()` does not in our proof.

Another interesting property is seen in Figure 19, where at least one parameter must be non-null. Again, we added this condition as a JML clause, allowing us to close the proof.

Case study. We also implemented a small web shop case study to measure the potential verification speedup when using the property type checker and KeY versus using just KeY by itself. The class diagram in Figure 20 shows all classes except for the class `Main` and their most important fields and methods. The case study consists of 269 total lines of Java code and 102 lines of property qualifier

[11] Example from https://checkerframework.org/api/org/checkerframework/framework/qual/EnsuresQualifierIf.html
[12] See the original file at https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/DiscReasonMap.java#L149
[13] See the original file at https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/inv/InvariantInfo.java
[14] See the original file at https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/diff/DetailedStatisticsVisitor.java#L148

```
public class InvariantInfo {
  public final @Nullable String vars;
  @JMLClause("ensures \result == vars")
  public @Nullable String vars() { return this.vars; }
  @JMLClause("ensures \result == null <==> vars == null")
  public @Nullable List var_permutations() { ... }
}
```

Fig. 18. JML specifications for `invInfo`[13].

```
@JMLClause("requires inv1 != null || inv2 != null")
public static int determineArity(
    @Nullable Invariant inv1, @Nullable Invariant inv2) {
  @SuppressWarnings("nullness")
  @NonNull Invariant inv = (inv1 != null) ? inv1 : inv2;
  ...
}
```

Fig. 19. Dependence between parameters[14].

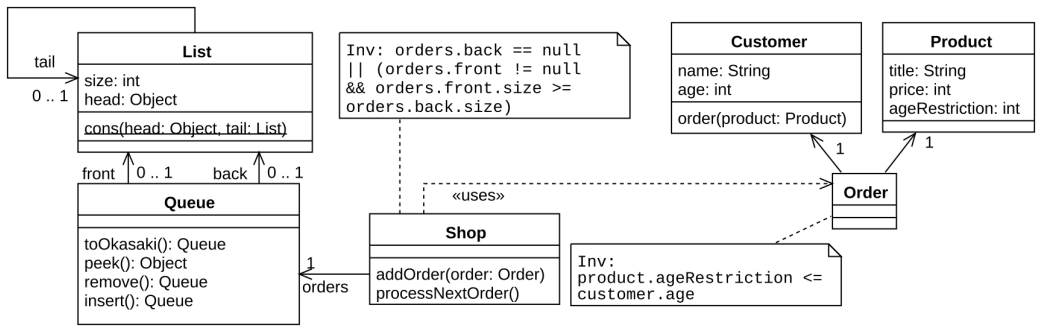

Fig. 20. UML diagram for the case study.

definitions. It implements a queue of orders, where an order is a pair consisting of a customer and a product such that the customer is old enough to buy the product.

We use seven property qualifier hierarchies: (1) a hierarchy for `@Positive` and `@Negative` integers; (2) a hierarchy for `@Interval(min,max)` qualifiers for more general integer intervals; (3) a hierarchy for `@Length(min,max)` qualifiers for lists of a particular length; (4) a nullness hierarchy with `@NonNull` and `@Nullable` qualifiers; (5) a hierarchy for `@AgedOver(age)` qualifiers to represent customers over a certain age; (6) a hierarchy for `@AllowedFor(age)` qualifiers to represent products with some age restriction. This, along with the `@AgedOver(age)` hierarchy, is used to guarantee that for every order, the customer is old enough to buy the product. (7) a hierarchy containing several qualifiers whose properties encode certain invariants or properties for the queue class, in particular, an invariant used by Okasaki [1995] to guarantee amortized constant-time insertion and removal. These hierarchies are defined in the language presented in Section 4.1. The full hierarchy definitions can be seen in Appendix C. The most important type properties for the `@AllowedFor(age)` hierarchy and for the

[15]The case study war run on a computer with an AMD Ryzen 7 PRO 4750U (8x1.7GHz) CPU and 32 GB of RAM.

Table 1. Run times for the case study[15].

	Raw translation	Enhanced translation (all files)	Enhanced translation (only files with unfinished proofs)
No. of trials (KeY)	10	10	10
Mean (KeY)	517 s	490 s	311 s
Deviation (KeY)	14 s	24 s	4 s
No. of trials (OpenJML)	10	10	0
Mean (OpenJML)	232 s	475 s	N/A
Deviation (OpenJML)	39 s	123 s	N/A

Okasaki queue hierarchy are shown in Figure 20 in the notes attached to the `Order` and `Shop` classes respectively.

An Okasaki queue is made up of two lists, called *front* and *back*. The *Okasaki invariant*, shown in the note attached to the `Shop` class in Figure 20, states 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*. Reestablishing the Okasaki invariant (by moving items from *back* to *front*) after every insertion and removal guarantees that those two operations can be done in amortized constant time [Okasaki 1995, 4][16].

All in all, the program contains 111 explicit annotations, of which 50 are `@JMLClauses` and the rest are property qualifiers.

In addition to the classes shown in Figure 20, there is a `Main` class which instantiates some orders and adds them to the order queue managed by the singleton `Shop` instance. The well-typedness of this `Main` class and that of the `Shop` class can be shown by the Property Checker, while the well-typedness of the other classes cannot be proven by the Property Checker alone. The reasons for this are twofold: First, the other classes often use dependent qualifiers, which are supported by the JML translator, but not by the checker. For example, the constructor `List(Object head, List tail)` returns an object of type `@Length(min="tail.size + 1", max="tail.size + 1") List`, which is dependent on `tail`. Second, while the `Main` and `Shop` classes only use and preserve the type properties, the other classes must establish them. For example, the fact that the method `Queue::toOkasaki` shown in Figure 21 establishes the Okasaki invariant cannot be proven by the type checker. For the `if` branch, the checker would have to see that the path condition implies that the invariant holds for `this` and that `this` thus has the type `@Okasaki Queue`. For the `else` branch, the checker would have to substitute the newly created queue into the invariant definition and evaluate it. While both of these things are beyond our current Property Checker, the *LiquidHaskell* tutorial by Jhala et al. [2017], which also implements Okasaki queues, shows that such properties are expressible and provable in a refinement type system. The `Main` and `Shop` classes on the other hand can use the guarantees established by the other classes in ways that can be proven by the checker. For example, by simply calling `orders.toOkasaki()` whenever the `orders` queue is modified, we guarantee that the invariant is preserved. We used some tricks here to get around the missing support for dependent types in the checker: The `Customer` constructor has the dependent signature `@AgedOver(age="age") Customer(String name, int age)`, so we added additional builder methods like `@AgedOver(age="18") customer18(String name)`. While the correctness of these builders must be established by KeY, the `Main` class which uses them can then be proven in the checker.

[16] Actually, this only works for lazy lists, which we do not use in this case study for simplicity's sake.

```
public @Okasaki Queue toOkasaki() {
  if (back == null || (front != null && front.size() >= back.size())) {
    // Okasaki invariant already holds.
    return this;
  } else {
    // Invariant does not hold.
    // Establish it by moving all items in back to front.
    return new Queue(rotate(front, back, null), null);
  }
}
```

Fig. 21. The method which establishes the Okasaki invariant.

```
@AllowedFor(age="18") Product product18 = ...;
@AgedOver(age="18") Customer customer18 = ...;
addOrder18(customer18, product18);
addOrder18(customer18, product18);
```

Fig. 22. `addOrder` requires additional specifications if the Property Checker is not used.

We say in Section 4 that the Property Checker requires every class which can be used with a qualifier to be immutable. In our case study, this applies to all classes except for `Main` and `Shop`. We made these classes immutable by making all of their transitive fields final. Methods which one could expect to modify an object, like `Queue::remove`, instead return a new object.

The proof times for the case study are shown in Table 1. We ran the case study three times:

First, we tried to prove all type properties using only KeY by running the JML translator without running the Property Checker first. This is the *raw translation*. This version required some additional JML specifications to be provable in KeY; instead of the 50 `@JMLClauses` mentioned above, we needed 5 more for a total of 55. For example, in Figure 22, KeY is unable to prove the correctness of the second call for `addOrder18` because it cannot guarantee that the first call did not modify the customer's age to be under 18. By annotating `addOrder18` with `assignable Shop.instance.orders`, we guarantee that no heap location except for the order queue is modified. The Property Checker on the other hand knows that the type property for `@AgedOver(age="18")` holds in *every* reachable state and can thus discharge this method call, so that it does not need to be proven by KeY. The total number of assertions and method call preconditions in the raw JML translation is 117, which, as seen in Table 1, take on average 517 seconds to prove in KeY.

Next, we repeated the process, but this time running the Property Checker before the JML translation. It is able to prove 90 of those assertions and preconditions, leaving 27 false positives. The JML translation which considers this is the *enhanced translation*. We have two options of how to proceed with the enhanced translation: We can either load every method in KeY, even those methods which have been completely discharged by the Property Checker (in which case KeY symbolically executes the methods without proving anything); this leads to a very similar run time of 490 seconds, though we no longer need the 5 additional JML specifications. Or we can only load those methods in which there remains something to be proven for KeY, in which case KeY's average run time decreases to 311 seconds. For all versions, the type checker and JML translator run for 8 seconds on average with a standard deviation of 0 seconds[17].

[17]For want of an implementation, we did not perform the preliminary checks outlined in Section 4.3.

This tells us that proving the comparably simple properties which the Property Checker can prove has no large impact on KeY's run time, which is instead dominated by the loading times, the symbolic execution, and proving the harder properties which the Property Checker cannot prove. Thus, discharging only *some* but not *all* of the assertions in a given method with the Property Checker does not substantially improve KeY's run time. However, if we write our program in such a way that the Property Checker can prove *all* assertions in a given method, then that method does not even have to be loaded in KeY, which does lead to a significant run time improvement.

In addition, we have seen that the Property Checker's ability to prove that type properties hold in every reachable state complements KeY's abilities by allowing us to prove a program's correctness with fewer JML specifications.

To confirm this, we also ran the case study in *OpenJML*, a fully automatic JML verification tool that generates first order verification conditions and uses SMT solvers to discharge them. Since OpenJML always proves all proof obligations in a program, we were not able to turn off the proof obligations which have been completely discharged by the checker. In addition, OpenJML reports some spurious or intentional errors, namely: (1) integer overflows, which we did not consider and told KeY to ignore, (2) possible null dereferences in the type property definitions and JML specifications, which we also ignore because they never occur if our case study is used correctly, (3) errors in the trampoline methods, which are correct by construction but not provable in OpenJML. We thus only used two versions of the case study with OpenJML: First, we tried to prove the raw translation. Ignoring the spurious/intentional errors listed above, OpenJML reports 25 warnings. Second, we tried to prove the enhanced translation, which (again, ignoring the spurious ones) leads to no warnings. The fact that OpenJML takes longer to run on the enhanced translation than on the raw translation is due to the fact that for the raw translation, the warnings lead to some proof obligations not being considered. We can thus see that using a type checker in addition to a verification system lowers the specification burden as well as the verification burden.

From a development point of view, the classes which we had to prove in KeY/OpenJML (i.e., the queue and list implementations, as well as the classes for customers, products, and orders) probably need to be changed less often than the classes we were able to prove completely in the Property Checker (the shop class which manages the orders, and the main class which instantiates the shop and places some orders), which further decreases the verification burden.

The missing support for dependent types, as well as the fact that we used the generic Property Checker instead of stronger checkers, led to some unnecessary overhead. For example, the Property Checker does not know that the result of a constructor is always `@NonNull`, and as a result every constructor had to go through the JML translation to be proved correct. We believe that with some improvements to the Property Checker, or with the use of dedicated checkers, the verification time could be lowered even further.

6 RELATED WORK

There is a large body of work on both type systems and formal verification. This section briefly discusses the relation to work in run-time verification, dependent and refinement types, and static verification.

Run-time verification. Run-time verification is sometimes a sensible alternative to the compile-time verification offered by deductive verification tools and type checkers. Like type checkers, it scales well and is easy to use, and like deductive verification, it is quite powerful. In fact, run-time verification has no false positives at all. However, it cannot statically guarantee that a program always respects its specification.

Cheon and Leavens [2002] introduce a run-time checker for JML which transforms JML-annotated Java code in such a way that exceptions are thrown if a JML specification is violated. Our approach could use this run-time checker to enforce the remaining property type assertions at run time if a static analysis is not desired or not possible.

Bean Validation [Morling 2019] is an example of run-time verification for Java which uses annotations. 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 declarations of types, fields, methods, constructors, parameters, and container elements of *JavaBeans* only. 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 [Morling 2019, 6.4].

The *Ada* programming language [Intermetrics, Inc. 2016] has *subtype predicates* which can be used to define a subtype by constraining the supertype instances to those for which the predicate holds. These predicates are checked by inserting assertions into the code which are checked at run time.

Knowles and Flanagan [2010] introduce an approach for *hybrid type checking*, which combines static type systems with dynamic run-time checks: A type use whose correctness cannot be established at compile time can be translated into a dynamic cast whose correctness is checked at run time. Unlike our approach, where the type checker outputs two possible results for a program (definitely well-typed, or unknown), in hybrid type checking, there are three possible results (definitely well-typed, definitely ill-typed, or unknown). To achieve an analogous extension in our approach, we would have to annotate the type qualifier hierarchies not only with implications between the property types, but also with inconsistencies between them. This would allow the type checker to report guaranteed true errors (i.e., without any potential for false positives) if inconsistent types are encountered. Furthermore, although this is not considered in this paper, KeY is sometimes able to generate counterexamples for programs that do not satisfy their JML specification, thus proving them to be definitely ill-typed.

In contrast to these run-time approaches, we combine two static verification approaches to achieve both scalability and precision.

Dependent types and refinement types. The property types introduced in this paper can be thought of as *refinement types*. Refinement types were first introduced by Freeman and Pfenning [1991] for the ML language. Vazou et al. [2014] introduce the refinement type system used in the language *LiquidHaskell*. 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 precondition, a function postcondition, 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 [Chlipala 2013, 1.2.2]. The type system of LiquidHaskell limits itself to SMT-decidable predicates. The well-typedness of a program is decided by translating all refinement predicates to SMT formulas and using an SMT solver [Vazou et al. 2014, 1, 2.1]. Full dependent type systems go a step further and make no restrictions about which values can appear in types. This comes at the cost of the well-typedness of a program no longer being decidable. An example of a language with full dependent types is Idris [Brady 2013].

Our translation of property types to JML formulas allows us to use the full power of a deductive verification tool like KeY in addition to a type checker. While our type checker implementation does not support dependent types, JML clauses are allowed to contain arbitrary side-effect free Java expressions. Appendix A briefly explains how dependent property types can be included in our approach.

Stump and Wehrman [2006] propose a notion of *property types* for Java. They extend generics to allow not just types but also values as type arguments. They associate such parameterized classes with properties. For example, the class `Leq<int i, int j>` can be instantiated as `Leq<3, 4>` to express the fact that $3 \leq 4$. They adapt the Curry-Howard correspondence [Howard 1980] to imperative languages: They use instances of classes like `Leq<3,4>` as proofs that the respective property holds. The *trusted* code that creates these proof objects is encapsulated from the *untrusted* client code which uses the property types.

Our property type approach does not make proof objects available at run time. Instead, we introduce a pluggable type system that has no effect on a program's run-time semantics. While our approach allows the user to separate a program into untrusted code which can be checked with a type checker, and a trusted core whose correctness cannot be established by a type checker, this is not obligatory. Our approach is able to prove the correctness of both the untrusted and the trusted code within the same pipeline.

Toman et al. [2020] introduce the *ConSORT* system, which shows how ownership types can be used to allow refinement types to work with mutable objects. Like our work, this is an application of refinement types to imperative languages which reduces the number of false positives reported by type checkers. Unlike our work, it achieves this reduction not by combining a type checker with a deductive verification system, but by providing a more powerful type system which combines refinement and ownership types.

Sammler et al. [2021] introduce *RefinedC*, another combination of refinement and ownership types. It is similar to our work in that the type refinements are translated into another language, in this case Coq [Bertot and Castéran 2004]. However, unlike in our approach, there is no light-weight type checker that runs before this translation. Instead, everything is proven in Coq.

Static verification and type systems. Ownership and ownership type systems [Clarke et al. 2012] are frequently used to ensure program correctness or support program verification. Dietl and Müller [2012] survey the use of ownership for program verification. Universe types [Dietl and Müller 2004] is an ownership type system for Java and JML that is geared towards modular verification of programs.

The Rust programming language uses an ownership and borrowing system. *RustBelt* [Jung et al. 2017] is an approach to proving the soundness of the Rust programming language—more specifically, the correctness of the memory safety guarantees claimed by Rust's ownership type system—whereby the soundness of all programs which use a safe subset of Rust is proven once and for all, while the soundness of programs using unsafe features is established by generating a verification condition which can be shown in the Coq [Bertot and Castéran 2004] proof assistant. The approach of guaranteeing the correctness of most programs with the help of a type system and verifying the programs/libraries for which this is not possible using a proof assistant is very similar to our approach of combining type systems with deductive verification. However, we do not limit proofs to memory safety.

Dafny [Leino 2017] is a programming language designed for deductive verification. Its type system distinguishes between non-null types (`object`) and nullable types (`object?`). The verification engine comprises a state-of-the-art approximating nullness type checker. An assertion for the absence of null values is automatically added in those cases where the type checker cannot guarantee null safety. Dafny follows similar ideas as our approach, but again limited to the one built-in property of non-nullness.

JavaCOP [Markstrum et al. 2010] is a framework for pluggable Java type systems based on the theoretical framework by Chin et al. [2005]. Unlike the Checker Framework, which provides declarative meta-annotations and a Java API for the development of type checkers, JavaCOP provides

a domain-specific declarative language for type systems. The hierarchy definition language outlined in Section 4.1 is similar to, but far less expressive than JavaCOP's language. The focus of our work is not on the definition of the types and type systems, but on the translation of these types to a specification language.

7 CONCLUSION AND FUTURE WORK

This paper introduces a novel approach to avoid false positives in type checkers by translating types to specifications. This approach works for all *property type systems*, which are type systems in which every type is associated with some logical property that expresses its meaning.

We implemented this approach for Java by providing a generic property checker written with the Checker Framework and a translator of property types to JML specifications. The property type checker can be instantiated with simple user-defined property type systems and the resulting JML specifications can be verified with KeY or OpenJML.

Our approach combines the scalability of static type checkers with the precision of deductive verification. We prove the soundness of the approach for a minimal language and illustrate the benefits of the approach using case studies: removing false positives of a nullness checker and speeding up deductive verification.

This paper focused on proving the correctness of an existing program specification. It would be very helpful if, given an incorrect program, the verification pipeline were able to insert additional assertions or checks at critical places in the program. This can be combined with type inference on the side of the Checker Framework—as demonstrated for example by Dietl et al. [2011b], Huang et al. [2012], Xiang et al. [2020]—which determines a valid typing for a given program, if one exists.

It would also be interesting to see if and how the property type system can be extended to allow for non-monotonic properties of mutable objects to be verified. For that, one would have to leverage an ownership system like Universe Types to deal with the problems presented by reference aliasing.

As mentioned in Section 4.3, the approach can also be extended with support for generics.

Furthermore, the current implementation is incomplete, as it neither contains the preliminary checks from Section 4.3 nor a checker for the well-formedness of property qualifier hierarchies.

It would be interesting to more deeply explore how our approach works with type checkers stronger than our generic Property Checker, to offer a definition language for stronger type systems, and to also offer a way to prove the correctness of the given type rules using a deductive verifier.

DATA AVAILABILITY STATEMENT

The versions of the Property Checker, KeY, and OpenJML used for this paper, along with source code and proof files for the evaluation, are available on Zenodo [Lanzinger et al. 2021].

ACKNOWLEDGMENTS

We would like to thank the reviewers of this paper for their valuable feedback.

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants program, RGPIN-2020-05502, and an Early Researcher Award from the Government of Ontario. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSERC or the Governments of Ontario or Canada.

REFERENCES

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

Gilad Bracha. 2004. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (Sept. 2013), 552–593. Issue 05. https://doi.org/10.1017/S095679681300018X

Checker Framework developers. 2020. Checker Framework Manual. https://checkerframework.org/manual/ Version 3.3.0.

Yoonsik Cheon and Gary T. Leavens. 2002. A Runtime Assertion Checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice* (Las Vegas, Nevada, USA) *(SERP '02)*. CSREA Press, Las Vegas, Nevada, USA, 322–328.

Brian Chin, Shane Markstrum, and Todd Millstein. 2005. Semantic Type Qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/1065010.1065022

Adam Chlipala. 2013. *Certified Programming with Dependent Types*. http://adam.chlipala.net/cpdt/

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2012. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*, D. Clarke, J. Noble, and T. Wrigstad (Eds.). Springer. https://doi.org/10.1007/978-3-642-36946-9_3

David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 472–479. https://doi.org/10.1007/978-3-642-20398-5_35

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd Schiller. 2011a. Building and Using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. Association for Computing Machinery, 681–690. https://doi.org/10.1145/1985793.1985889

Werner Dietl, Michael D. Ernst, and Peter Müller. 2011b. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, Mira Mezini (Ed.). Springer Berlin Heidelberg, 333–357. https://doi.org/10.1007/978-3-642-22655-7_16

Werner Dietl and Peter Müller. 2004. Universes: Lightweight Ownership for JML. *Journal of Object Technology (JOT), Special Issue: ECOOP 2004 Workshop FTfJP* (Oct. 2004).

Werner Dietl and Peter Müller. 2012. Object Ownership in Program Verification. In *Aliasing in Object-Oriented Programming*, D. Clarke, J. Noble, and T. Wrigstad (Eds.). Springer, 289–318. https://doi.org/10.1007/978-3-642-36946-9_11

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

William Alvin Howard. 1980. The Formulae-as-Types Notion of Construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan (Eds.). Academic Press.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 879–896. https://doi.org/10.1145/2384616.2384680

Intermetrics, Inc. 2016. Ada Reference Manual. http://ada-auth.org/arm.html

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Transactions on Programming Languages and Systems* 32, 2, Article 6 (Feb. 2010), 34 pages. https://doi.org/10.1145/1667048.1667051

Florian Lanzinger. 2021. *Property Types in Java: Combining Type Systems and Deductive Verification*. Master's Thesis. Karlsruher Institut für Technologie.

Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. 2021. *Property Checker – Scalability and Precision by Combining Expressive Type Systems and Deductive Verification*. https://doi.org/10.5281/zenodo.5483138

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. JML Reference Manual. http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf Revision 2344.

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

Yitzhak Mandelbaum, David Walker, and Robert Harper. 2003. An Effective Theory of Type Refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 213–225. https://doi.org/10.1145/944705.944725

Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative Pluggable Types for Java. *ACM Transactions on Programming Languages and Systems* 32, 2, Article 4 (Feb. 2010), 37 pages. https://doi.org/10.1145/1667048.1667049

Gunnar Morling. 2019. Bean Validation specification. https://beanvalidation.org/2.0/spec/ Version 2.0.

Chris Okasaki. 1995. Simple and efficient purely functional queues and deques. *Journal of Functional Programming* 5, 4 (1995), 583–592.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Association for Computing Machinery, 201–212. https://doi.org/10.1145/1390630.1390656

Alex Potanin, Östlund Johan, Yoav Zibin, and Michael D. Ernst. 2012. Immutability. In *Aliasing in Object-Oriented Programming*, D. Clarke, J. Noble, and T. Wrigstad (Eds.). Springer. https://doi.org/10.1007/978-3-642-36946-9_9

Arsenii Rudich, Ádám Darvas, and Peter Müller. 2008. Checking Well-Formedness of Pure-Method Specifications. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5014)*, Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere (Eds.). Springer, 68–83. https://doi.org/10.1007/978-3-540-68237-0_7

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Aaron Stump and Ian Wehrman. 2006. Property Types: Semantic Programming for Java. In *13th International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD 2006)*. 9 pages.

Alexander J. Summers and Peter Müller. 2011. Freedom before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1013–1032. https://doi.org/10.1145/2048066.2048142

John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (proceedings of the 19th acm sigplan international conference on functional programming ed.) *(ICFP '14)*. Association for Computing Machinery, 269–282. https://doi.org/10.1145/2628136.2628161

Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. 2020. Precise inference of expressive units of measurement types. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 142:1–142:28. https://doi.org/10.1145/3428210