Property Types for Mutable Data Structures in Java

Master's Thesis by

Joshua Bachmeier, B. Sc.

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

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Florian Lanzinger, M. Sc.
Second advisor:	Dr. Mattias Ulbrich
Third advisor:	Dr. Werner Dietl*

1st November 2021 – 1st August 2022

*University of Waterloo, Canada

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

Abstract

Property Types are a kind of user-defined refinement type about variables and fields in a program. They are verified by discharging as many properties as possible using a scalable type checker. The remaining assertions are forwarded to a less scalable but more powerful deductive verification tool. However, the design and implementation by Lanzinger et al. cannot function in the presence of aliasing and mutability.

In this thesis, we find that property checking can be performed safely on mutable data structures provided exclusive mutable access to the referenced object, which we define as property-safety. We study different approaches to aliasing control, including uniqueness, ownership and permissions. Based on this research, we present the Exclusivity Type System, which can be used to check the property-safety of program variables and class fields. Using flow-sensitive type refinement, we develop Mutable Property Types, which can track changes in a variable's property type over time. Impure methods can be annotated to specify how they change the Property Types of their receiver and arguments. We explain how the original Property Checker's program translation can be adapted to include correct assertions about the pre- and post-types of each method. We present a prototypical implementation of the Exclusivity Checker for Java programs using the Checker Framework.

Our work provides many insights into the nature of property type verification on mutable data structures and we devise the theoretical groundwork for performing this verification. To corroborate the reasonableness of the presented approach, we suggest a thorough analysis of our systems through formal proofs.

Contents

1 Introduction

In computer programming, there are many ways to perform static verification (i. e. at compile time) of formal correctness guarantees. The two most prominent approaches, both in research and practice, are *type checking* and *deductive verification*.

Most type systems can be naturally integrated with programming languages, scale very well and impose comparatively little overhead on the programmer. These advantages largely stem from the fact that type systems are decidable. However, decidability is bought at the cost of precision: Type checkers usually only check for a conservative approximation of the issue they are intended to protect from. That is, a type checker might reject programs as unsafe that would in reality never emit the problematic behaviour at run time (such errors are called *false positives*). This usually occurs in cases where the observation that the error in question never occurs depends on a complex dynamic condition that can only be known at run time.

In contrast, methods for deductive verification can be precise and reject exactly that set of programs that emit the problematic behaviour. The downside of deductive verification tools is that they do not scale well and impose a significant cognitive overhead. Programs need to be elaborately annotated using special formal specification languages, proving is often interactive and in most cases only semi-decidable.

In the context of Java [Gos+21], an example for a type system based approach to generalised program verification is the Checker Framework [Pap+08], which we use in our work; an example for a deductive verification is the KeY tool [Ahr+16].

Lanzinger et al. [Lan+21] introduce *property types*, as system that combines the scalability of type checking with the precision of deductive verification. In their work, declarations in programs are annotated with so called property types, which express a certain formal guarantee about the annotated object. These programs are processed using type checking techniques and any properties that the type system can already guarantee to be valid are removed from the program. The remaining properties (which might contain false positives) are translated into formal specifications and the thus annotated program can be further analysed using deductive verification techniques, possibly discharging further properties and leaving only those that are truly problematic. The Property Checker is implemented for Java using the Checker Framework to implement the type checker and JML [Lea+13] to discharge unverifiable properties to KeY.

However, the work of Lanzinger et al. suffers from a severe limitation: Only deeply immutable objects may be annotated by property types, because assignment to the field of an object might invalidate the property, which cannot be handled by their system. As virtually any real-world program in imperative programming languages employs mutability in some way, solving this problem is paramount.

```
void f() {
  @MinLength(5) List l5 = ...;
  @MinLength(4) List l4;

  l4 = l5;

  l4.removeFirst(); // -> l4: @MinLength(3)
}
```

Listing 1.1: Property types on mutable objects in the presence of aliasing.

1.1 Reference Aliasing and Mutability

To be precise, the problem is not mutability on its own, but mutability in *the presence of aliasing*. Reference aliasing describes a situation in which two references (e. g. local variables in a method or fields of a class) refer to the same memory location (object). In this case, one reference might be used to modify the object and the change will be visible to the other reference.

Listing 1.1 illustrates how this is problematic for property types. Here `l5` is annotated to contain at least 5 elements, `l4` is annotated to contain at least 4 elements. Then, `l5` is assigned to `l4`, which is valid, since `l5` also contains at least 4 elements. In the next step however, an element is removed from `l4`. This is no problem for `l4`, since the property of `l4` is known at that point and can be updated to reflect the change. However this line affects `l5` in the same way, which is not syntactically deducible. While it would even be possible for a sufficiently sophisticated type system to detect the aliasing of `l4` and `l5`, a situation where this is not possible can easily be imagined, e. g. if an `l5` and `l4` were fields of completely different objects.

This possible invalidation of foreign properties is the core problem of verifying property types for mutable data structures.

1.2 Approach

We aim to remedy this by working toward a complete solution to realise changing property types about mutable data structures. A high-level overview of our approach is shown in fig. 1.1. On a theoretical level, we establish a universal assumption that is required to hold about a reference for it to be safely annotated with a property. The assumption is verifiable by a type checker. Abstractly, it guarantees exclusive mutable access to the object behind a reference, thereby asserting that any property maintained by the holder of the reference cannot be invalidated through a foreign reference. By exposing the assumption, we achieve modularity: Different type systems of varying complexity and precision might be used to verify the assumption. There already exist a multitude of different approaches that could be used to achieve exclusive mutability, which we discuss in this thesis.

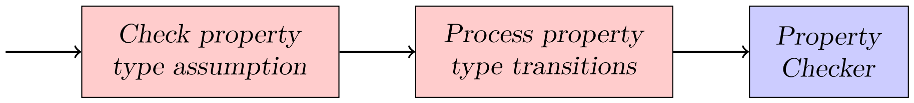
Check property type assumption
Process property type transitions
Property Checker

Figure 1.1: Simplified pipeline for modular mutable property type checking (see fig. 5.1 for detailed version).

The next issue lies in the very nature of mutability: The objects and therefore their properties can change. Consider again the example from listing 1.1: The call `l4.removeFirst()` changes `l4`'s property. To accommodate this fact, we propose a syntax to specify how a method affects the properties of its parameters. We explain how a program annotated with such *property transitions* can be translated so that it may be processed by Lanzinger's property-checking pipeline, with some adaptations to their system. Thereby, one can take full advantage of their combination of scalability and precision.

1.3 Overview

In this thesis, we answer the question of how non-trivial type properties can be verified in the presence of mutability and reference aliasing and how mutation and reference sharing need to be restricted to enable sound property verification. We also determine how this can be realised while still being able to discharge properties not verifiable by the type system to a deductive verification mechanism. The key contributions of our work are the following:

1. We expose a universal *assumption* for mutable property types that constitutes a type-system-verifiable assertion about references. We thereby achieve modularity: Different systems to verify the assumption can be combined with different system for the verification of program properties.
2. We propose a simple *Exclusivity Type System* that verifies this assumption and can also be used on its own as a system to prevent bugs induced by reference-aliasing. For this, we explicitly formalise rules for flow-sensitive type refinement, basing well-typedness on applicability of valid refinements and discuss the advantages and disadvantages of this unusual approach. We supply a prototypical implementation for Java using the Checker Framework.
3. We present *mutable property types*, a concept of modelling changing type properties using flow-sensitive type refinement that can be naturally integrated with the property types of Lanzinger et al.
4. We give a thorough survey of available work on aliasing control and access management in program verification. We organise the proposed systems based on methodology and discuss how they relate to each other.

We start by introducing the foundations upon which our work builds and establish basic formalisms in chapter 2. In chapter 3, we develop the property type assumption and present the Exclusivity Type System. Building upon this, we define a formalism to realise mutable property types in chapter 4. Our implementation of the Exclusivity Checker and a description of how our work can be integrated with the original Property Checker is given in chapter 4. We discuss related work in chapter 6, especially regarding methods of alias mitigation. Finally, we conclude our work in chapter 7 and discuss the advantages and disadvantages of our approach.

2 Foundations

In this chapter we establish the foundations that precede our work. While sections 2.1 and 2.2 summarise work done by others that we build upon, sections 2.3 and 2.4 introduce concepts and formalisms developed by us that will be used throughout this thesis.

2.1 Checker Framework

The Checker Framework [Pap+08] is a framework for verifying additional type systems for Java programs as a way to extend the standard Java type system. Examples include a *nullness*, an *interning* and a *tainting* checker[1]. While the Checker Framework ships many builtin checkers, its key feature is that it provides a framework and many utilities for writing third-party checkers of custom type systems.

In this section, we give a brief overview of the Checker Framework and especially those aspects most relevant to our work.

Java Annotations The Checker Framework builds upon the Java feature of Annotations. Technically, annotations are a special kind of interface type. While they do not directly affect program semantics, they provide information and instructions to tools and libraries [Gos+21, sections 9.6 and 9.7].

Annotations may take zero or more parameters and are written as `@Annotation`, `@Annotation("value")` or `@Annotation(key="value", ...)`. When annotations were introduced in Java 5, they could only appear before declarations, such as local variables, instance fields, methods and method parameters. Java 8 extended annotations so that they can now appear at any type use, such as instance creation expressions, casts, implements clauses and throws clauses. With this, it is possible to use annotations to implement pluggable type systems.

Listing 2.1 shows an example of an annotation definition and usage. Here, the custom annotation `@GreaterEq` is supposed to express that the annotated variable is greater than or equal to the given value. Java provides many builtin annotations, such as `@SupressWarnings` or `@Deprecated` that provide meta information to the compiler.

These annotations can then be processed by so called *annotation processors* which may perform checks on the supplied annotations or transform the program based on the annotations. The Checker Framework is one such annotation processor.

[1]A more complete list of checkers can be found in the Checker Framework Manual [Che22].

Type Checking To implement type checking of custom type systems, the Checker Framework employs a *visitor pattern* to scan each element in the abstract syntax tree and verify its well-typedness. The Checker Framework provides the base class `BaseTypeChecker` which already implements standard type-system behaviour such as basic subtyping rules during assignment and can be subclassed to overwrite or replace this functionality.

Dataflow Analysis Furthermore, the Checker Framework supports *flow-sensitive type refinement* [Die+11; Aut22] That is, it can sometimes deduce an expression's type to be more specific than what the programmer originally declared. Consider the example in listing 2.2, which is taken from the Checker Framework Manual. Here `myVar` is declared as `@Nullable String`, but it is treated as `@NonNull String` within the body of the `if` test. Dually to the type checking process, type refinement is performed by visiting each element in the programs control flow graph and calling a *transfer function* on the visited element.

Again, the Checker Framework provides a base class `CFTransfer` which implements standard refinement behaviour, but can be overwritten to implement more advanced refinement steps for custom type systems.

```java
@interface GreaterEq {
  int value();
}

class Foo {
  @GreaterEq(28) int
  add(@GreaterEq(17) int a, @GreaterEq(11) int b) {
    return a + b;
  }
}
```

Listing 2.1: Definition and usage of `GreaterEq` annotation.

```java
@Nullable String myVar;
...                     // myVar has type @Nullable String here.
myVar.hashCode();       // warning: possible dereference of null.
...
if (myVar != null) {
  ...                   // myVar has type @NonNull String here.
  myVar.hashCode();     // no warning.
}
```

Listing 2.2: Example code illustrating local type inference in Java.

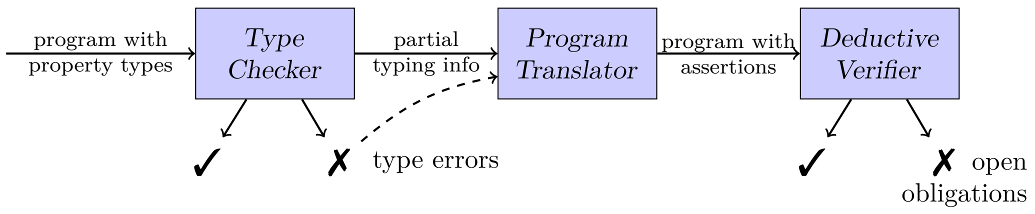

Figure 2.1: Property-type-checking pipeline [Lan+21].

2.2 Property Types

As described in the introduction, this thesis is the advancement of property types to incorporate mutable data structures. Property types were introduced by Lanzinger et al. [Lan+21] To combine expressive type systems with deductive verification. Since Lanzinger's work forms the groundwork upon which this thesis is built, we will give a detailed explanation, especially of the theoretical constructs that are referred to in this thesis.

The Property Checker works in three stages, which are shown in fig. 2.1, taken verbatim from Lanzingers work. In the first stage, the *type checker* attempts to verify the given property-type-annotated program. If it encounters any type errors, these are passed to the *program translator*, which translates the program into an equivalent program annotated with formal specifications in JML [Lea+13]. These specifications encode the type properties that the type checker was unable to guarantee. In the final stage, a *deductive verifier*, such as KeY [Ahr+16] attempts verify the formally specified program. Any remaining open proof obligations in this stage might point to an actual bug in the code, but could also stem from other reasons, such as a lack of reasoning power of the prover or weak formal specification.

Since our work is almost exclusively concerned with the first stage, the type checking phase, this summary focuses on the formal background of the type system established by Lanzinger et al.

Property Qualifier Hierarchies property types are based on what is called *property qualifiers*. Property qualifiers are built over a system of *base types* (i. e. Java types). In the following the base type is denoted by T_s and the value of the object in question is s. The semantics of a parameterised property qualifier $@A$ can be subsumed as in eq. 2.1.

$$@A(v_1 \dots v_n) \implies \mathrm{Prop}_A(s, v_1 \dots v_n) \wedge \mathrm{Wf}_A(v_1 \dots v_n) \tag{2.1}$$

where $v_1 \dots v_n$ are the parameters. Prop_A is the formal condition (property) that the subject s' value must satisfy and Wf_A is the *well-formedness condition*, stating that the property instantiated with the parameters is even a valid property qualifier. The resulting *property type* is then called $T_A = @A(v_1 \dots v_n)T_s$, which is a subtype of T_s (the property type is a specialisation of the base type and keeps the original semantics in tact).

For example, eq. 2.2 shows a property qualifier stating that the subject's value is in a given interval. Here the base type T_s is integer.

$$@\text{Interval}(\min, \max) \implies \text{Prop}_{\text{Interval}}(s, \min, \max) \land \text{Wf}_{\text{Interval}}(\min, \max) \tag{2.2}$$

where

$$\begin{aligned} \text{Prop}_{\text{Interval}}(s, \min, \max) :\Longleftrightarrow{}& \min \leq s \leq \max \\ \text{Wf}_{\text{Interval}}(\min, \max) :\Longleftrightarrow{}& -2^{31} \leq \min \leq \max \leq 2^{31} - 1 \end{aligned}$$

These property qualifiers are then formed into a hierarchy which is used as a subtyping relation by the type checker. This is described by eq. 2.3: The base type of the more specific property type must be a subtype of the more general base type and the property of the subtype needs to imply that of the supertype.

$$\begin{aligned} &@A(a_1 \dots a_n)U \sqsubseteq @B(b_1 \dots b_n)V \\ &\quad \Longleftrightarrow\ U \sqsubseteq V\ \land\ \forall s : U\ .\ \text{Prop}_A(s, a_1 \dots a_n) \to \text{Prop}_B(s, b_1 \dots b_n) \end{aligned} \tag{2.3}$$

where $\sqsubseteq$ is the subtype relation[2].

Of course, these hierarchies are not formed automatically by the type checker based on the formal property. Instead, a subrelation <: is specified by the programmer such that

$$\forall T_A, T_B\ .\ T_A <: T_B \to T_A \sqsubseteq T_B$$

by providing a boolean expression that only depends on A's and B's parameters like for example in eq. 2.4:

$$@\text{Interval}(\min, \max) <: @\text{Interval}(\min', \max') :\Longleftrightarrow \min' \leq \min \land \max \leq \max' \tag{2.4}$$

This should be read as: An integer interval is a subtype of another integer interval, if and only if the former is contained in the latter.

For brevity, since we are mostly not concerned with property type parameters and base types, we usually denote $\mathbb{P}$ the property type lattice in question and write $\pi \in \mathbb{P}$ to denote a property like $@A(a_1 \dots a_n)U$ and $\text{Prop}_\pi(s)$ for $\text{Prop}_A(s, a_1 \dots a_n)$.

Program Translation Based on these properties, assertions are inserted into the program that check if the properties hold. For example, fig. 2.2 shows that before an assignment the type's property is asserted. Similar rules to inject assertions and assumptions are defined for method invocations and declarations. The thus translated program (with assertions already proven by the type checker removed), can then be analysed using deductive methods.

[2]In Lanzinger's work, $\preceq$ is used for this, but to be congruent with our syntax we use $\sqsubseteq$.

$$\frac{\Pi[x] = @A(v_1 \dots v_n)T}{\Pi \vdash x = y \rightsquigarrow \text{assert}\ \ \text{Prop}_A(y, v_1 \dots v_n); x \mathrel{\widehat{=}} y} \tag{Transl-Assign}$$

Figure 2.2: Example program translation rule from Lanzinger et al. [Lan+21] (syntax adapted to match our convention).

Implementation Lanzinger et al. implemented their proposal for Java using the Checker Framework (see section 2.1). Their checker generates type lattices based on input files containing definitions of property qualifiers and qualifier hierarchies in a domain-specific language. In the next step, their custom checker not only tries to verify the annotated program using the generated property types, but also outputs a JML annotated program that can be further analysed with deductive verification tool.

2.3 Field Assignment Language

As a basis to develop the underlying theory for our Exclusivity Type System (chapter 3) and, upon that, the Mutability Type System (chapter 4), we define a minimalistic pseudo-language, the Field Assignment Language (FAL). The language is designed to be similar to Java, but is not a formal sub-language.

Technically, FAL is a WHILE-language with recursive function calls and records (which we call classes). The fundamental and only mutating operation is the *field assignment*, i. e. the assignment of expressions to fields of `this` (the receiver of the current method). Direct assignment to local variables is also possible. The full grammar is given in fig. 2.3.

FAL is as simple as possible and contains only those language constructs necessary to reason about references, aliasing and simple program properties. As such, there are classes with fields and methods but no advanced object-orientation features such as polymorphism or inheritance.

A formal definition of the operational semantics of FAL is not within the scope of this work. However, we give a natural-language overview of the intuitive semantics of each of the syntactical constructs.

Assignment There are no general expressions, only variations of the assignment statement. The left-hand side of an assignment can only be a local variable or a field, while the right-hand side can also be an object instantiation (`new`-expression) or a method invocation.

The special form `this` indicates the receiving object of the current method and can never appear alone, but only as the left-most element in a field reference or method invocation. This prevents the `this`-reference to be leaked from the current context, which allows more flexible type-checking of method receivers later on.

Otherwise, access is only allowed to local variables and fields of `this`, not deeper. Methods can only be called on `this`, local variables and fields of `this`. For the sake of simplicity, there is no dynamic name resolutions, which means all accesses to fields are always of the form `this.field`. Therefore, local variables and fields can be distinguished lexically, reducing specification overhead.

Types Aside from classes, there is the primitive type `int`, which has the usual semantics attached to it. While we do not explicitly support arithmetic expressions, these can be thought of as invocations of special methods.

$$
\begin{aligned}
\text{Prog} &= \text{ClassDecl}\,* \\
&\quad \text{Class and method declarations:} \\
\text{ClassDecl} &= \texttt{class}\,\text{Ident}\,\texttt{\{}\,\text{FieldDecl}*\ \text{MthDecl}*\texttt{\}} \\
\text{FieldDecl} &= \text{Type Ident}\ \texttt{;} \\
\text{MthDecl} &= \text{Type Ident}\,\texttt{(}\,\text{RecvType}\texttt{,}\ \ \text{ParamDecl}*\texttt{)}\,\text{Stmt} \\
\text{ParamDecl} &= \text{ParamTypeTransition Ident}\ \texttt{,} \\
\text{RecvType} &= \text{TypeTransition ExclusivityModifier}\,\texttt{this} \\
\text{ParamTypeTransition} &= \text{TypeTransition PartialType} \\
\text{TypeTransition} &= \text{Property}\,\texttt{->}\,\text{Property} \\
&\quad \text{Statements:} \\
\text{Stmt} &= \text{VarDecl} \mid \text{Assignment} \mid \text{If} \mid \text{While} \mid \texttt{\{}\,\text{Stmt}\,*\texttt{\}} \\
\text{VarDecl} &= \text{Type Ident}\ \texttt{;} \\
\text{Assignment} &= \text{Var}\,\texttt{=}\,\text{Expr}\ \texttt{;} \\
\text{Var} &= \text{LocalVar} \mid \text{Field} \\
\text{Expr} &= \text{Var} \\
&\quad \mid \texttt{new}\,\text{Ident}\,\texttt{()} \\
&\quad \mid (\text{Var} \mid \texttt{this})\,\texttt{.}\,\text{Ident}\,\texttt{(}\,\text{Var}\,*\texttt{)} \\
\text{If} &= \texttt{if (}\,\text{Var}\,\texttt{)}\,\text{Stmt}\ [\,\texttt{else}\,\text{Stmt}\,] \\
\text{While} &= \texttt{while (}\,\text{Var}\,\texttt{)}\,\text{Stmt} \\
\text{LocalVar} &= \text{Ident} \\
\text{Field} &= \texttt{this.}\,\text{Ident} \\
&\quad \text{Type specifications:} \\
\text{Type} &= \text{Property PartialType} \\
\text{PartialType} &= \text{ExclusivityModifier Ident} \mid \texttt{int} \\
\text{ExclusivityModifier} &= \texttt{@ExclMut} \mid \texttt{@Immut} \mid \texttt{@RO} \mid \texttt{@ShrMut} \\
\text{Property} &= \textit{property type according to Lanzingers syntax} \\
\text{Ident} &= \textit{identifier chars}
\end{aligned}
$$

Figure 2.3: Grammar for Field Assignment Language (FAL).

Class types are equipped with an exclusivity modifier which are used to declare types in the Exclusivity Type System. While local variables and fields are additionally declared with a property type, method parameters instead have a property type transition `a -> b`. Here, `a` is the property that must hold for the parameter *before* the method invocation (precondition) and `b` is the property that is guaranteed to hold for the parameter *after* the method invocation (postcondition). The transitions implement a kind of flow-sensitivity and are fundamental for a mutable property type system, since they allow the programmer to express *how* an object is mutated by a method invocation. Note that transitions only apply to the property-part of the type, not the exclusivity modifier. This limitation is discussed in section 7.2.

While the uniqueness modifiers and property type declarations are mandatory in the grammar, we sometimes omit them throughout this work for the sake of brevity. In this case, a natural default element can be assumed in its place.

See section 2.2 for a more detailed explanation of property types. The nature of exclusivity and mutable property types is explained in more detail in chapters 3 and 4, respectively.

Control Flow Control flow constructs are limited to while- and if-statements. Conditions of while- and if-statements can only be local variables or fields, no other expressions such as method invocations. This is not a limitation, as the result of a method invocation can simply be assigned to a local variable beforehand. Because there is no boolean type, the type of a condition must be `int`, where a value of zero is regarded as *false* and values unequal to zero are regarded as *true*.

Statements can be grouped in scopes, where variables declared within a scope cannot be accessed outside of it. However, shadowing of outer variables is forbidden by type system construction the reason for which is explained in chapter 3.

Terminology In this thesis, we use some special terms when talking about FAL code, which we briefly define here to avoid ambiguity:

reference	A local variable or field of a class type that references an *object*.
object	An instance of a class that is pointed to (referenced) by one or more *references*. When more than one reference points to the same object, we speak of *aliasing*.
relevant properties	The relevant properties of an *object* are the properties of all *references* pointing to this object.
program point	A statement in a method of the program. It can be thought of as a line of source code. At each program point, all *references* have a specific (possibly refined) type. This should be differentiated from a *program state*, which would represent an execution state at run time (with concrete variable values, heap state, etc.).

2.4 Type Contexts

In this work, we develop multiple type systems, namely the Exclusivity Type System (chapter 3) and the mutability type system (chapter 4). For this we work with so called *type contexts*, which assign types in a specific type system to variables and fields in a program state. To ease specification of type rules in the following chapters, we give some auxillary definitions regarding the handling of types and type contexts. As usual, the types in a type system form a lattice. We use uppercase blackboard bold letters to denote the set of elements in a type system, such as $\mathbb{E}$ for the exclusivity lattice or $\mathbb{P}$ for a property type lattice.

Basics We first define type contexts and some operations to access their elements and modify the contexts.

Definition 1 (Type context)**.** *A type context* Γ *is a* right-unique *relation*

$$\Gamma \subset L \times \mathbb{T}$$

where L *is the set of locations (variables or fields of the current receiver) and* $\mathbb{T}$ *is a type lattice.*

We refer to type contexts using capital Greek letters, e. g. $\mathcal{E}$ (calligraphic Epsilon) for exclusivity type contexts or Π (Pi) for property type contexts. The greek letter Γ (Gamma) is used to refer to an arbitrary type context in this section.

When not ambiguous, we sometimes leave out the set-braces and write just $x : \tau, x' : \tau'$ instead of $\{x : \tau, x' : \tau'\}$. As they are simply relations, type contexts can be combined using set union $\Gamma \cup \Gamma'$. Contrary to the definition of type context updates below (definition 3), set union does not account for duplicate entries in the type contexts, so special care needs to be taken when this simple form is used.

Definition 2 (Type context access)**.** *To access elements of a type context, we write*

$$\Gamma[x] = \tau :\Longleftrightarrow (x : \tau) \in \Gamma$$

Since Γ *is right-unique,* $\Gamma[x]$ *is unique for any* x*.*

Definition 3 (Type context update)**.** *When merging two type contexts, elements of the latter override those of the former.*

$$\Gamma \Lleftarrow \Gamma' := \Gamma' \cup \{(x : \tau) \in \Gamma \mid x \notin \operatorname{dom}(\Gamma')\}$$

where dom *is the domain of a relation, i. e.* $\operatorname{dom}(R) = \{a \mid \exists b.(a, b) \in R\}$*.*

The type context update is well defined, since $\Gamma \Lleftarrow \Gamma'$ *is right-unique if both* Γ *and* Γ' *are right-unique. In other words, the definitions ensures that no duplicate entries are ever added to a type context.*

Definition 4 (Restriction and subtraction)**.** *To restrict a type context to a certain set of variables, or to subtract a certain set of variables, we write:*

$$V \lhd \Gamma \coloneqq \{(x : \tau) \in \Gamma \mid x \in V\}$$
$$V \mathbin{\lhd\!\!\!-} \Gamma \coloneqq \{(x : \tau) \in \Gamma \mid x \notin V\}$$

Definition 5 (Type context join)**.** *We join two type contexts by joining all the elements in the cut of their domains*

$$\Gamma \sqcup \Gamma' \coloneqq \{(x : \Gamma[x] \sqcup \Gamma'[x]) \mid x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')\}$$

where $\tau \sqcup \tau'$ is the supremum (smallest upper bound) of τ and τ', with the usual lattice semantics.

Refinements Throughout this work, we need to distinguish between declared types (what the programmer wrote) and refined types (what our system inferred the type to actually be). For this, we write Γ_D for the declared and Γ_R for the refined type context. The refined types must always be at least as precise as the declared types, as per the definition of a subcontext in definition 6: $\Gamma_R \sqsubseteq \Gamma_D$

Definition 6 (Subcontext)**.** *A type context is a subcontext of another type context, if and only if its domain is a subset of the other domain and all the types are subtypes of the corresponding entry in the other context.*

$$\Gamma \sqsubseteq \Gamma' :\Longleftrightarrow x \in \mathrm{dom}(\Gamma') \land \Gamma[x] \sqsubseteq \Gamma'[x] \quad \forall x \in \mathrm{dom}(\Gamma)$$

Definitions for $\sqsubset$, $\sqsupseteq$ and $\sqsupset$ follow naturally.

If, given a declared type context, two refinement contexts are valid then their combination using each of the four operators is still valid (theorem 1). This can be seen easily by the definitions of the operators. Note that this is not the case for the join operator $\sqcup$.

Theorem 1 (Closedness)**.** *Let Γ_R, Γ'_R, Γ_D be type contexts such that*

$$\Gamma_R \sqsubseteq \Gamma_D \land \Gamma'_R \sqsubseteq \Gamma_D$$

then

$$\Gamma_R \mathbin{\lhd\!\!\!+} \Gamma'_R \sqsubseteq \Gamma_D$$
$$\Gamma_R \lhd \Gamma'_R \sqsubseteq \Gamma_D$$
$$\Gamma_R \mathbin{\lhd\!\!\!-} \Gamma'_R \sqsubseteq \Gamma_D$$
$$\Gamma_R \cup \Gamma'_R \sqsubseteq \Gamma_D$$

The last statement holds only if $\Gamma_R \cup \Gamma'_R$ is right-unique.

A notation that is frequently encountered in this thesis is $(\Gamma_D \blacktriangleleft\!\!+ \Gamma_R)[x]$, which is equal to the refined type of x if that exists and the declared type otherwise. In other words:

$$(\Gamma_D \blacktriangleleft\!\!+ \Gamma_R)[x] = \begin{cases} \Gamma_R[x] & \text{if } x \in \mathrm{dom}(\Gamma_R) \\ \Gamma_D[x] & \text{else} \end{cases}$$

Definition 7 (Refinement step)**.** *The refinement-step notation means that* FAL *code c is well-typed under contexts Γ_D, Γ_R and the check produces contexts Γ_D' and Γ_R':*

$$\Gamma_D, \Gamma_R \vdash c \mapsto \Gamma_D', \Gamma_R'$$

The declared context is unchanged by most type rules (except rules *T-Var-Decl* and *T-Mth-Decl), i. e. $\Gamma_D = \Gamma_D'$ in which case it is omitted from the right side.*

3 Exclusivity

As established in the introduction a requirement for *mutable* property types is the restriction of access to objects in a way that gives the holder of a reference sufficient control over the object to be able to assume and maintain a property.

In this chapter, we first establish precisely what this means in section 3.1, i. e. what kind of predicate about a reference must be asserted to safely maintain a property. Following this, we establish a simple type system for the Field Assignment Language (FAL) in section 3.2, which includes a type that implements said predicate.

3.1 Requirements for Property Types

In order to establish a predicate for safely maintaining a property about the object behind a reference at compile-time, a definition of what "safely" means in FAL is required.

When dealing with property types about mutable objects the central problem originates from aliasing: If the reference in question was unique, i. e. no other reference in the program could ever refer to the same object, maintaining the property would be no issue at all. Every mutation of the object would be done from a point in the program where the only relevant property was the one annotated to said reference. But, when allowing both aliasing and mutation, the object behind a reference could be changed from a point in the program where that particular reference's property is not known. Or, seen from the other point of view, the point in the program mutating an object can never be guaranteed to know all properties relevant for the object. These observations lead to a natural definition 8 of property safety.

Definition 8 (Property-safe reference)**.** *A reference (local variable or field) is* property-safe*, if and only if any access of the referenced object (including all transitively reachable objects)*

1. *does not mutate the object* or
2. *is performed through that reference.*

In chapter 4 we analyse in more detail how program correctness is achieved by this definition. Intuitively, at each program point either item 1 asserts that aliases need not be considered and item 2 asserts that mutations are *guarded* by the holder of the reference. An observation similar to item 1 has also been established by Lanzinger et al. [Lan+21, section 4.3]. The definition of property-safe references can easily be lifted to types as in definition 9.

Definition 9 (Property-safe type)**.** *Given a type lattice $\mathbb{T}$, $\tau \in \mathbb{T}$ is* property-safe*, if and only if for any reference x:*

$$x : \tau \implies x \textit{ is property-safe}$$

3.2 Exclusivity Type System

We now propose a type system which contains a type that is property-safe. In fact, it contains multiple property-safe types which are, however, all subtypes of a common property-safe type. In section 3.2.1 we present a suitable lattice and assign semantics to each of the lattice elements, while in section 3.2.2 we formally define the type rules that achieve said semantics.

3.2.1 Capabilities and Lattice

When handling references, we consider the following four *capabilities* attached to references, which can be thought of as operations the holder of a reference may perform.

READ	Reading the referenced object
WRITE	Mutating the referenced object
COPY	Copying the reference (shallow copy)
PROP	Assuming and maintaining a formal property about the referenced object

While COPY concerns the reference itself, READ, WRITE and PROP make a statement about the referenced object. When performing an operation on a transitively reachable object, the capabilities of all references in the chain must allow the operation. For example, when the object referenced by a has a field b and a has WRITE, but b has not then the object referenced by $a.b$ cannot be mutated, at least not through this reference. However, the reference $a.b$ itself could be mutated to point to another object, since a has WRITE.

Lattice A reference may be equipped with a subset of these capabilities and as such the type of the reference is an element of the power set of all capabilities:

$$\mathcal{P}(C), \text{ where } C = \{\textsf{Read}, \textsf{Write}, \textsf{Copy}, \textsf{Prop}\}$$

Together with the subset relation for subtyping ($\sqsubseteq := \subseteq$), $\mathcal{P}(C)$ already forms a lattice, but contains elements that are not useful for our purposes, e. g. $\{\textsf{Read}, \textsf{Write}\}$ without either COPY or PROP. Additionally, we impose no restriction on *reading* a reference, since reading a reference can never violate a property (see item 1 of definition 8) and therefore only consider elements which contain READ. As a result, we can define an *exclusivity lattice* $\mathbb{E}$ as a sublattice of $\mathcal{P}(C)$ in definition 10:

Definition 10 (Exclusivity lattice)**.**

$$\mathbb{E} := \{\text{RO}, \text{ShrMut}, \text{Immut}, \text{Restricted}, \text{ExclMut}, C\}$$

where $\top = \text{RO}$ *and* $\bot = C$*.*

Type	Capabilities[a]	Supertype	Aliases
RO	READ[b]		*any*
ShrMut	WRITE, COPY	RO	ShrMut
Restricted	PROP	RO	Immut[c]
Immut	COPY	Restricted	Immut
ExclMut	WRITE	Restricted	RO

[a]All types implicitly inherit capabilities of supertypes.
[b]All types allow copying as RO, which effectively gives RO COPY.
[c]Restricted references have a single *inactive* ExclMut alias, which is the "lent" reference.

Table 3.1: Explanation of exclusivity lattice elements.

Table 3.1 shows the capabilities granted by each type, additionally to the capabilities inherited by supertypes. For example, ShrMut $= \{$READ, WRITE, COPY$\} =$ RO $\cup \{$WRITE, COPY$\}$. Table 3.1 also shows the possible types of other references to the same object. For instance, a Immut reference might have aliases which are also Immut, but can never have a ShrMut or even ExclMut alias.

Notice that ExclMut ("exclusively mutable") does not grant COPY: Copying a reference would invalidate exclusivity, as it would create another reference to the same object. ShrMut ("shared mutable") on the other hand grants both WRITE and COPY. In this case, ShrMut aliases are explicitly allowed. Therefore, contrary to ExclMut, it must not grant PROP; the property might be violated by mutation through another reference. Immut ("immutable") on the other hand grants PROP while also granting COPY: There might be aliases, but since they also cannot mutate the object, a property can be assumed to hold. Therefore, we understand immutability as defined by Potanin et al. [Pot+13, section 2.1]. Lastly, Restricted represents a reference that may have a property but can neither be copied nor used for mutation. The use-case for this is a ExclMut field of an object that is accessed through a non-ExclMut reference. Such a reference must not be mutated, since the outer reference does not allow mutation, but might still be equipped with a property that can be assumed to hold. Incidentally, Restricted is a common supertype of ExclMut an Immut. By construction of FAL (fig. 2.3), no reference may be explicitly annotated with Restricted. Instead, the type is automatically generated during *access adaptation*. This is explained in more detail below.

Figure 3.1 visualises the exclusivity lattice. It contains the bottom type $\bot$, which represents an invalid type: A reference that can be mutated (WRITE $\in \bot$) and aliased (COPY $\in \bot$) is not property-safe, but PROP $\in \bot$. The dashed line separates the types that allow mutation (below) from those that do not (above). The dotted line separates the types that may be annotated with a property (inside the parabola) from those that may not (outside the parabola).

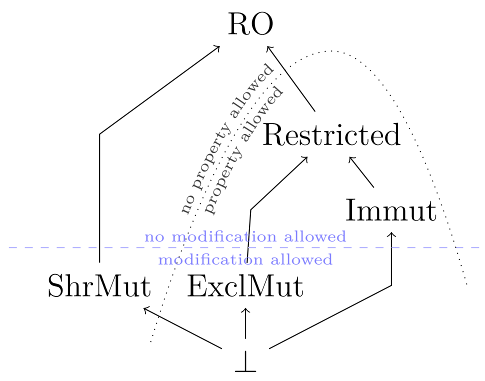

Figure 3.1: Lattice for exclusivity type system $\mathbb{E}$.

For convenience, we define three predicates that can be used to query the capabilities of a given type based on the lattice structure in definition 11. With this, we have fully abstracted the types from the original capabilities and can continue to define a set of type rules achieving the same semantics.

Definition 11 (Copyable, mutable and property-annotatable)**.**

$$\begin{aligned} \mathrm{prop}(\xi) &:\Longleftrightarrow \xi \sqsubseteq \mathrm{Restricted} \\ \mathrm{copy}(\xi) &:\Longleftrightarrow \mathrm{ExclMut} \not\sqsubseteq \xi \\ \mathrm{mut}(\xi) &:\Longleftrightarrow \mathrm{Immut} \not\sqsubseteq \xi \end{aligned}$$

for any $\xi \in \mathbb{E}$*.*

The predicates mut and prop can easily be visualised by studying fig. 3.1. copy can be thought of "must not be on the path from ExclMut to RO". Each of these is equivalent to explicitly checking for containment of the corresponding capability (e. g. $\mathrm{mut}(\xi) \iff \textsc{Write} \in \xi$).

3.2.2 Type Rules

We define the type rules structurally for all possible statements in a method, thus establishing a notion of well-typedness for methods. By convention, the greek ξ (minuscule xi) and variations (ξ', ξ_t, etc.) represent elements of the exclusivity lattice $\mathbb{E}$. $\mathcal{E}$ is a type context over $\mathbb{E}$ (see section 2.4). Literal FAL code is always set in `typewriter font`, variables representing FAL code however are not. The magic value `this` always refers to the current receiver. We assume that class, method and field declarations are globally available. For brevity we leave out all parts of the full type not significant for exclusivity (i. e. property and base type) in the type rules. The special form $\Gamma_D, \Gamma_R \vdash l \approx e \mapsto \Gamma'_R$ is called *pseudo-assignment* and states that an expression e is assignable to a variable l and what refinement this produces.

This type system is based on the concept of *type refinement*. In addition to the declared types $\mathcal{E}_D$ a refinement context $\mathcal{E}_R$ is maintained that contains more precise type judgements deduced by the type system based on the encountered statements. The fundamental working of this type system is that each rule specifies a type refinement using the refinement-step notation (definition 7) and contains the assumption that the newly refined types are compatible with the declared types. Multiple rules may therefore be tried to type a single statement. If none is applicable the statement is not well-typed.

Assignment At the heart of typing FAL in the Exclusivity Type System lies the assignment statement. Figure 3.2 shows the type rules for the assignment statement. Any assignment is first typed by rule T-Assign, which ensures that the current receiver is mutable when the left-hand side is a field. The rule then delegates further checking to the pseudo-assignment $l \approx e$, for which there are multiple rules that differ in the kind of right-hand side of the assignment in each conclusion.

The three rules for $(\mathcal{E}_D \Lleftarrow \mathcal{E}_R)[e] = \text{ExclMut}$ (rules T-Ref-Split-Mut, T-Ref-Split-Immut and T-Ref-Transfer) all have the same conclusion. Which one is to be applied can be statically distinguished by checking compatibility with the declared types. Each rule assumes a different refinement $\mathcal{E}$ for the right- and left-hand side of the assignment. Whichever rules assumes a valid refinement $\mathcal{E}$ (i. e. $\mathcal{E} \sqsubseteq \mathcal{E}_D$) may be applied. Consider for example rule T-Ref-Split-Mut. Here $\mathcal{E} = e : \text{ShrMut}, l : \text{ShrMut}$. The exclusive reference might be split into two mutably shared references, if this rule is applicable. But if e. g. $\mathcal{E}_D[l] = \text{Immut}$, this rule cannot be applied since $\text{ShrMut} \not\sqsubseteq \text{Immut}$. Intuitively, these three rules achieve the semantics that a given ExclMut reference may either be split into two ShrMut references, two Immut references or the exclusivity can be fully “transferred” to the left-hand-side, leaving the original reference RO.

Rules T-Ref-Copy, T-Ref-Copy-Ro and T-Ref-New are straightforward: Any copyable reference may be copied, any reference may be copied as RO and references to a newly instantiated objects are always ExclMut.

The most complex rule is T-Call. For better understanding, we break it up and examine each assumption individually:

$$\vdash \text{class}(v)[m] : \xi_t(\xi_1 \ldots \xi_n) \to \xi_r$$

Firstly, we assume the method signature to be available. ξ_t is receiver's, $\xi_1 \ldots \xi_n$ the parameters' and ξ_r the return type.

$$\forall i \in 1 \ldots n \;.\; \mathcal{E}_D \cup \{\texttt{tmp} : \xi_i\}, \{\texttt{tmp}\} \lhd \mathcal{E}_R^{i-1} \vdash \texttt{tmp} \approx a_i \mapsto \mathcal{E}_R^{i}$$

Next, each parameter is assigned the supplied reference. For each parameter with type ξ_i a temporary variable `tmp` is created and the argument a_i is pseudo-assigned to it ($\texttt{tmp} \approx a_i$) using the same rules as for the real assignment above. In each round, the resulting refinement context is carried to the next round. The refinement of `tmp` from the previous round is removed, because the left-hand side of this pseudo-assignment does not really exist in the callers context. Effectively, this pseudo-assignment is much simpler than the real assignment and can equivalently regarded as a function mapping ξ_i and a_i's type to the new refined type, as is shown in table 3.2. The table entries are the resulting $\mathcal{E}_R[a_i]$. Empty cells are for combinations which are ill-typed.

$$\frac{\begin{array}{c} l \text{ is local variable} \lor \text{mut}(\mathcal{E}_D[\texttt{this}]) \\ \mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \texttt{=} e\texttt{;} \mapsto \mathcal{E}'_R} \quad (\textsc{T-Assign})$$

$$\frac{\begin{array}{c} \mathcal{E} = e : \text{ShrMut}, l : \text{ShrMut} \qquad \mathcal{E} \sqsubseteq \mathcal{E}_D \\ \mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} \mathcal{E} \qquad (\mathcal{E}_D \mathrel{\triangleleft\!\!+} \mathcal{E}_R)[e] = \text{ExclMut} \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R} \quad (\textsc{T-Ref-Split-Mut})$$

$$\frac{\begin{array}{c} \mathcal{E} = e : \text{Immut}, l : \text{Immut} \qquad \mathcal{E} \sqsubseteq \mathcal{E}_D \\ \mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} \mathcal{E} \qquad (\mathcal{E}_D \mathrel{\triangleleft\!\!+} \mathcal{E}_R)[e] = \text{ExclMut} \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R} \quad (\textsc{T-Ref-Split-Immut})$$

$$\frac{\begin{array}{c} \mathcal{E} = e : \text{RO}, l : \text{ExclMut} \qquad \mathcal{E} \sqsubseteq \mathcal{E}_D \\ \mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} \mathcal{E} \qquad (\mathcal{E}_D \mathrel{\triangleleft\!\!+} \mathcal{E}_R)[e] = \text{ExclMut} \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R} \quad (\textsc{T-Ref-Transfer})$$

$$\frac{\begin{array}{c} \xi_e = (\mathcal{E}_D \mathrel{\triangleleft\!\!+} \mathcal{E}_R)[e] \qquad \mathcal{E} = l : \xi_e \qquad \mathcal{E} \sqsubseteq \mathcal{E}_D \\ \mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} \mathcal{E} \qquad \text{copy}(\xi_e) \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R} \quad (\textsc{T-Ref-Copy})$$

$$\frac{\begin{array}{c} \mathcal{E} = l : \text{RO} \qquad \mathcal{E} \sqsubseteq \mathcal{E}_D \\ \mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} \mathcal{E} \end{array}}{\mathcal{E}_D, \mathcal{E}_R \vdash l \approx e \mapsto \mathcal{E}'_R} \; (\textsc{T-Ref-Copy-Ro}) \qquad \frac{\mathcal{E}'_R = \mathcal{E}_R \mathrel{\triangleleft\!\!+} l : \text{ExclMut}}{\mathcal{E}_R, \mathcal{E}_D \vdash l \approx \texttt{new}\ C\texttt{()} \mapsto \mathcal{E}_R} \; (\textsc{T-Ref-New})$$

$$\frac{\begin{array}{c} \vdash \text{class}(v)[m] : \xi_t(\xi_1 \dots \xi_n) \to \xi_r \\ \forall i \in 1 \dots n \,.\, \mathcal{E}_D \cup \{\texttt{tmp} : \xi_i\}, \{\texttt{tmp}\} \triangleleft \mathcal{E}_R^{i-1} \vdash \texttt{tmp} \approx a_i \mapsto \mathcal{E}_R^i \qquad (\mathcal{E}_D \mathrel{\triangleleft\!\!+} \mathcal{E}_R)[v] \sqsubseteq \xi_t \\ \mathcal{E}'_R = \{f \mid f \text{ is field of } \texttt{this} \land (\text{prop}(\mathcal{E}_D[\texttt{this}]) \to v = \texttt{this})\} \cup \{\texttt{tmp}\} \triangleleft \mathcal{E}_R^n \\ \mathcal{E}''_R = \mathcal{E}'_R \mathrel{\triangleleft\!\!+} l : \xi_r \qquad \mathcal{E}''_R \sqsubseteq \mathcal{E}_D \end{array}}{\mathcal{E}_R^0, \mathcal{E}_D \vdash l \approx v\,.\,m\texttt{(}a_1 \dots a_n\texttt{)} \mapsto \mathcal{E}''_R} \quad (\textsc{T-Call})$$

Figure 3.2: Type rules for Assignments. Here, l is any Lhs, e any Expr, C and m any Ident, v and a_i any Var according to the grammar in fig. 2.3.

	right-hand side $((\mathcal{E}_D \mathrel{\triangleleft\!\!\!+} \mathcal{E}_R)[a_i])$				
left-hand side (ξ_i)	**RO**	**ShrMut**	**Immut**	**ExclMut**	**Restricted**
RO	RO	ShrMut	Immut	ExclMut	Restricted
ShrMut		ShrMut		ShrMut	
Immut			Immut	Immut	
ExclMut				RO	

Table 3.2: Resulting right-hand side refinement after pseudo-assignment in rule T-Call.

$$(\mathcal{E}_D \mathrel{\triangleleft\!\!\!+} \mathcal{E}_R)[v] \sqsubseteq \xi_t$$

The receiver reference is handled differently: It must simply be compatible with the method's declared receiver type. Because `this` can never appear alone by construction of the grammar, the reference cannot be leaked anyway.

$$\mathcal{E}'_R = \{f \mid f \text{ is field of } \texttt{this} \land (\text{prop}(\mathcal{E}_D[\texttt{this}]) \to v = \texttt{this})\} \cup \{\texttt{tmp}\} \triangleleft \mathcal{E}^n_R$$

Because the callee might modify shared references in a way that violates local refinements of fields of `this` that it doesn't know about, all refinements regarding references to objects possible affected by the method need to be removed from the callers refinement context. The second part of the predicate $(\text{prop}(\mathcal{E}_D[\texttt{this}]) \to v = \texttt{this})$ states that refinement invalidation is only necessary if either `this` is declared ShrMut or RO (because only then v might transitively contain a reference to `this`) or the method is called directly on `this`.

$$\mathcal{E}''_R = \mathcal{E}'_R \mathrel{\triangleleft\!\!\!+} l : \xi_r \qquad \mathcal{E}''_R \sqsubseteq \mathcal{E}_D$$

Finally, we refine the left-hand side to the return type of the method and ensure that the final resulting refinement context is valid.

Control-Flow Figure 3.3 shows typing of control flow structures. This is largely standard. Rule T-If types the then- and else-arm individually and joins the resulting refinement contexts. Rule T-While requires a fixed point iteration, because in the assumption $\mathcal{E}'_R$ (the result of the refinement-step) is rejoined with the original $\mathcal{E}_R$. Because of this, the refinement-step must be repeated until $\mathcal{E}_R \sqcup \mathcal{E}'_R = \mathcal{E}'_R$.

Note that in all control-flow rules we explicitly require the declared contexts to not be changed by the refinement-step. By this we achieve that the top-level statement is not a unscoped variable declaration.

Statement Composition and Variable Declarations Figure 3.4 shows the remaining statements. Rule T-Scope-Compose consecutively types each statement in the scope and removes all refinements not regarding references contained in the original declared context. Other refinements "survive" a scope, but new declarations naturally do not. Rule T-Var-Decl forbids redefinition as well as shadowing of outer variables, because rule T-Scope-Compose couldn't handle it.

$$\frac{\mathcal{E}_D, \mathcal{E}_R \vdash s_1 \mapsto \mathcal{E}_D, \mathcal{E}_R^a \qquad \mathcal{E}_D, \mathcal{E}_R \vdash s_2 \mapsto \mathcal{E}_D, \mathcal{E}_R^b \qquad \mathcal{E}_R' = \mathcal{E}_R^a \sqcup \mathcal{E}_R^b}{\mathcal{E}_D, \mathcal{E}_R \vdash \texttt{if (}v\texttt{)}\ s_1\ \texttt{else}\ s_2 \mapsto \mathcal{E}_R'} \quad \text{(T-If)}$$

$$\frac{\mathcal{E}_D, \mathcal{E}_R \sqcup \mathcal{E}_R' \vdash s \mapsto \mathcal{E}_D, \mathcal{E}_R'}{\mathcal{E}_D, \mathcal{E}_R \vdash \texttt{while (}v\texttt{)}\ s \mapsto \mathcal{E}_R'} \quad \text{(T-While)}$$

Figure 3.3: Type rules for control flow structures. Here, s, s_1 and s_2 are Stmt and v a Var according to the grammar in fig. 2.3.

$$\frac{\forall i \in 1 \ldots n \,.\, \mathcal{E}_D^{i-1}, \mathcal{E}_R^{i-1} \vdash s_i \mapsto \mathcal{E}_D^i, \mathcal{E}_R^i \qquad \mathcal{E}_R' = \mathrm{dom}(\mathcal{E}_D^0) \lhd \mathcal{E}_R^n}{\mathcal{E}_D^0, \mathcal{E}_R^0 \vdash \texttt{\{}s_1 \ldots s_n\texttt{\}} \mapsto \mathcal{E}_R'} \quad \text{(T-Scope-Compose)}$$

$$\frac{\mathcal{E}_D' = \mathcal{E}_D \cup \{v : \xi\} \qquad v \notin \mathrm{dom}(\mathcal{E}_D)}{\mathcal{E}_D, \mathcal{E}_R \vdash \xi\, v\ \texttt{;} \mapsto \mathcal{E}_R'} \quad \text{(T-Var-Decl)}$$

Figure 3.4: Type rules for remaining statements. Here, s, s_1 and s_2 are Stmt and v a Var according to the grammar in fig. 2.3.

Method Definitions And Access Adaption Finally, rule T-Mth-Decl in fig. 3.5 establishes well-typedness of a whole method. This rule forms the entry point for the type checker.

Naturally, the refinement context is initially empty. The main purpose of this rule is to build the declared type context, which consists of the parameter types, the field types and the special values `this` and `retval`. Returning a value from a function is done by assignment to the special variable `retval`.

For the field types, an *access adaptation* takes places according to the *access combination* operator (definition 12). This is necessary to account for the fact that all references in a chain must share a capability for it to take effect. $\xi \blacktriangleright \xi'$ returns the effective type of accessing a ξ' field through a ξ receiver. Intuitively, this is always simply the unchanged field type, except when the field is ExclMut without the receiver being also ExclMut or the field is ShrMut while the receiver reference does not allow modification.

$$\frac{\mathcal{E}_D, \varnothing \vdash s \mapsto \mathcal{E}_D, \mathcal{E}_R \qquad \mathcal{E}_D = \{a_i : \xi_i\}_{i \in 1..n} \cup \{f : \xi_t \blacktriangleright \xi_f\}_{(f:\xi_f) \in \text{fields(CurrentClass)}} \cup \{\texttt{this} : \xi_t, \texttt{retval} : \xi_r\}}{\vdash \xi_r\, m\texttt{(}\xi_t\, \texttt{this}, \xi_1\, a_1 \ldots \xi_i\, a_i\texttt{)}\ s} \tag{T-Mth-Decl}$$

Figure 3.5: Type rule for whole method bodies. Here, m is an Ident, a_i are Var and s is a Stmt according to the grammar in fig. 2.3.

Definition 12 (Access combination)**.**

$$\cdot \blacktriangleright \cdot \ : \ \mathbb{E} \times \mathbb{E} \setminus \{\text{Restricted}\} \to \mathbb{E}$$

$$\xi \blacktriangleright \xi' := \begin{cases} \text{Restricted} & \textit{if } \xi' \sqsubseteq \text{ExclMut} \land \xi \not\sqsubseteq \text{ExclMut} \\ \text{RO} & \textit{if } \xi' \sqsubseteq \text{ShrMut} \land \neg\, \text{mut}(\xi) \\ \xi' & \textit{else} \end{cases}$$

For examples of valid and invalid programs in the Exclusivity Type System, refer to section 5.3, where we present test cases for our Java implementation, which are almost identical in syntax to the corresponding FAL instances.

3.3 Well-Typedness

To tie everything together, we formally define well-typedness based on the type rules and connect it to property-safety (definition 9).

Definition 13 (Method well-typedness)**.** *A method m is well-typed, if and only if*

$$\vdash \xi_r\, m\texttt{(}\xi_t\, \texttt{this}, \xi_1\, a_1 \ldots \xi_i\, a_i\texttt{)}\ s$$

where $\xi_r\, m\texttt{(}\xi_t\, \texttt{this}, \xi_1\, a_1 \ldots \xi_i\, a_i\texttt{)}\ s$ is the method declaration.

Definition 14 (Program well-typedness)**.** *A program is* well-typed*, if and only if all methods of all classes in the program are well-typed.*

Theorem 2 (Correctness)**.** *In a well-typed program, at each program point*

$$\text{prop}((\mathcal{E}_D \Lsh \mathcal{E}_R)[x]) \implies x \textit{ is property-safe}$$

for all references x, with $\mathcal{E}_D$ and $\mathcal{E}_R$ being the type contexts at that program point.

We do not give a proof of theorem 2. Abstractly, however, its correctness follows from the equivalence of the type rules to the capability system and the selection of which types grant Prop.

4 Mutable Property Types

In chapter 3 we established an assumption for property types and defined a type system that provides this assumption. Abstractly, only references which are property-safe (definition 8) may be annotated with property types or be used in property definitions. We define this in more detail in section 4.2.

Initially however, we need to establish how the final property-type annotated program is produced. Adding mutability to property types necessitates the ability to change the property types of references. For example, a reference to a list might be annotated with `@MinLength(5)`. After adding an element to it we can know the type to be `@MinLength(6)` (which is more precise). On the other hand, removing an element generalises the type to `@MinLength(4)`. In section 4.1 we describe how this is accounted for and when type generalisations are possible.

Next, we give an idea of the changes necessary to the original Property Checker implementation in section 4.3 and finally argue how all of this produces an understanding of correctness compatible with the work of Lanzinger et al. [Lan+21] in section 4.4.

4.1 Property Refinement

In Field Assignment Language (FAL), each method's parameters a (as well as the receiver) are annotated with a property type transition

$$\pi \to \pi', \text{ where } \pi, \pi' \in \mathbb{P}$$

Semantically, π establishes the parameter's pre- and π' its postcondition. When invoking a method, $a : \pi$ must be guaranteed by the calling context. After the method returns, $a : \pi'$ may be assumed by the caller. On the other hand, the callee's context may assume $a : \pi$ at the beginning of the method but must guarantee $a : \pi'$ at the end of the method.

To achieve this, the program's property types are refined based on the declared type transitions to produce a *refined program*, where the pre- and postconditions are correctly included in the type contexts at each program point. Figure 4.1 shows an excerpt of the type rules governing this.

Rule T-PROP-CALL is used to type method invocations. As before, the method signature is assumed to be known. The types of the supplied parameter values as well as the receiver must be compatible with the declared pre-types of the method. Then their types can be refined to the declared post-types of the method.

Rule T-PROP-MTH-DECL shows the other side of this mechanism, the method declaration. Before typing the method's body statement with the refinement-step, the declared pre-types π_i of the parameters and receiver are added to the *refinement* context. It is important that they are not added to the declaration context, because these types may need to be generalised during typing of the method to reach the *target type* π'_i. Returning to our previous example, a method `remove` of a list might be annotated with `@MinLength(n) -> @MinLength(n-1) this` for the receiver parameter[1]. If the pre-type were added as a declared type it could never be violated and implementing such a method would be impossible. In the refinement produced by typing the body statement s, the parameters and receiver must then fulfill the post-type.

Rules T-Prop-Assign and T-Prop-Var-Decl are straightforward. Assignment transfers the refinement of the right-hand side to the left-hand side and when declaring a variable, its property type is stored in the declaration context.

The rules for the remaining statements are omitted here for brevity, since they are largely similar to the rules for the Exclusivity Type System shown in section 3.2.2 and provide no deeper insight into the property refinement mechanism.

$$\dfrac{\begin{array}{c}\vdash \mathrm{class}(v)[m] : (\pi_t \rightarrow \pi'_t)(\pi_1 \rightarrow \pi'_1 \ldots \pi_n \rightarrow \pi'_n) \rightarrow \pi_r \\ (\Pi_D \mathbin{\lhd\!\!+} \Pi_R)[v] \sqsubseteq \pi_t \qquad (\Pi_D \mathbin{\lhd\!\!+} \Pi_R)[a_i] \sqsubseteq \pi_i \\ \Pi'_R = \Pi_R \mathbin{\lhd\!\!+} v : \pi'_t, a_1 : \pi'_1 \ldots a_n : \pi'_n, l : \pi_r\end{array}}{\Pi_D, \Pi_R \vdash l\ \texttt{=}\ v\,\texttt{.}\,m\texttt{(}a_1 \ldots a_n\texttt{)} \mapsto \Pi'_R} \quad (\textsc{T-Prop-Call})$$

$$\dfrac{\begin{array}{c}\Pi_R = a_i : \pi_i \ldots a_n : \pi_n, \texttt{this} : \pi_t \qquad \Pi_D = \texttt{retval} : \pi_r \\ \Pi_D, \Pi_R \vdash s \mapsto \Pi_D, \Pi'_R \\ (\Pi_D \mathbin{\lhd\!\!+} \Pi'_R)[\texttt{this}] \sqsubseteq \pi'_t \qquad \forall i \in 1..n\,.\,(\Pi_D \mathbin{\lhd\!\!+} \Pi'_R)[a_i] \sqsubseteq \pi'_i\end{array}}{\vdash \pi_r\ m\texttt{(}\pi_t \rightarrow \pi'_t, \pi_1 \rightarrow \pi'_1 a_1 \ldots \pi_n \rightarrow \pi'_n a_n\texttt{)}\,s} \quad (\textsc{T-Prop-Mth-Decl})$$

$$\dfrac{\Pi'_R = \Pi_R \mathbin{\lhd\!\!+} l : \Pi_R[v]}{\Pi_D, \Pi_R \vdash l = v \mapsto \Pi'_R} \quad (\textsc{T-Prop-Assign}) \qquad\qquad \dfrac{\Pi'_D = \Pi_D \mathbin{\lhd\!\!+} v : \pi}{\Pi_D, \Pi_R \vdash \pi v \mapsto \Pi'_D} \quad (\textsc{T-Prop-Var-Decl})$$

Figure 4.1: Type rules for establishing property type pre- and postconditions for methods.

4.2 Property Safety

In a FAL program, all references are annotated with a type composed of an exclusivity type, a property type and a base type. To connect the exclusivity and the property

[1]For illustrative purposes, the property is a dependent type in this example, which is not currently possible in property types.

$$\frac{\begin{array}{c}\Pi_D, \Pi_R \vdash l\ \mathtt{=}\ v\mathtt{.}m\mathtt{(}a_1 \dots a_n\mathtt{)} \mapsto \Pi'_R \\ \vdash \mathrm{class}(v)[m] : (\pi_t \to \pi'_t)(\pi_1 \to \pi'_1 \dots \pi_n \to \pi'_n) \to \pi_r\end{array}}{\begin{array}{l}\Pi_D, \Pi_R \vdash x\ \mathtt{=}\ v\mathtt{.}m\mathtt{(}a_1 \dots a_n\mathtt{);} \\ \quad \rightsquigarrow \begin{cases}\mathtt{assert}\ \mathrm{Prop}_{\pi_t}\mathtt{(}v\mathtt{);}\ \mathtt{assert}\ \mathrm{Prop}_{\pi_1}\mathtt{(}a_1\mathtt{)}\dots\mathtt{assert}\ \mathrm{Prop}_{\pi_n}\mathtt{(}a_n\mathtt{);} \\ \qquad\qquad \mathtt{tmp} \mathrel{\widehat{=}} v\mathtt{.}m\mathtt{(}a_1 \dots a_n\mathtt{);} \\ \mathtt{assume}\ \mathrm{Prop}_{\pi'_t}\mathtt{(}v\mathtt{);}\ \mathtt{assume}\ \mathrm{Prop}_{\pi'_1}\mathtt{(}a_1\mathtt{)}\dots\mathtt{assume}\ \mathrm{Prop}_{\pi'_n}\mathtt{(}a_n\mathtt{);} \\ \quad \mathtt{assume}\ \mathrm{Prop}_{\pi_r}\mathtt{(tmp);}\ \mathtt{assert}\ \mathrm{Prop}_{(\Pi_D \triangleleft\!\!\!+ \Pi'_R)[v]}\mathtt{(}v\mathtt{);} \\ \qquad\qquad x \mathrel{\widehat{=}} \mathtt{tmp;}\end{cases}\end{array}} \quad (\textsc{Transl-Call-Mut})$$

Figure 4.2: Excerpt of adapted program translation rules.

component of each type, we define type validity in definition 15 such that only property-safe types may have a non-trivial property.

Definition 15 (Valid type). *Given a type system $\mathbb{T}$ and a property type system $\mathbb{P}$ with $\tau \in \mathbb{T}, \pi \in \mathbb{P}$, the compound type $\tau\pi$ is* valid *if and only if*

$$\pi \sqsubset \top \implies \tau \textit{ is property-safe}$$

where $\top \in \mathbb{P}$ is the top-element that every object fulfills, i. e. $\forall x\,.\ \mathrm{Prop}_\top(x)$.

Using this definition, we can lift our definition 9 of property-safety to whole methods (and thereby programs) in definition 16.

Definition 16 (Property-safe method). *A method is* property-safe *if all types occurring in it are valid.*

As stated in theorem 3, this definition of property-safety can be achieved with exclusivity types.

Theorem 3 (Property-safety of refined method). *A fully refined method using exclusivity types is property-safe if at each program point, for any reference x*

$$(\Pi_D \triangleleft\!\!\!+ \Pi_R)[x] \sqsubset \top \to \mathrm{prop}((\mathcal{E}_D \triangleleft\!\!\!+ \mathcal{E}_R)[x])$$

where $\Pi_D, \Pi_R, \mathcal{E}_D, \mathcal{E}_R$ are the type contexts at that particular program point.

4.3 Adapted Program Translation

The translation of programs annotated with property types to programs with assertions as defined by Lanzinger et al. [Lan+21, fig. 9, rule Transl-Call] must be adapted to account for changing properties, in particular the rule for method invocation. Figure 4.2 shows a sketch of the adapted version of this rule. The main difference to the original version is that after the method invocation, the post-types of the parameters are assumed. Similar adaptations must be applied to the remaining translation rules.

4.4 Correctness

Lanzinger et al. define correctness based on the translation of property-type annotated programs to programs containing assertions of the properties and the infallibility of these assertions. We do not formally adapt their definitions for our purposes, since Lanzinger et al. even use a different base language, but theorem 4 encompasses a similar meaning.

Theorem 4 (Property-correctness)**.** *If a method is property-safe and correct according to Lanzinger et al. [Lan+21, definition 3.11] with respect to a property type lattice* $\mathbb{P}$*, then at every program point*

$$x : \pi \implies \mathrm{Prop}_\pi(x)$$

for all $\pi \in \mathbb{P}$*.*

This theorem binds together the results of section 3.1 and this chapter. The crux lies in the requirement for the method to be property-safe: For any reference x that does not satisfy $\mathrm{prop}(x)$, $\pi = \top$ and since is $\mathrm{Prop}_\top(x)$ universally valid, the above implication holds. For any reference that satisfies $\mathrm{prop}(x)$ (so π might be non-trivial), validity of the implication follows from the construction of the property type transitions and the program translation.

5 Implementation

We provide a partial implementation of our proposed enhancements to the Property Checker of Lanzinger et al. The Property Checker, as well as our enhancements, target the Java Programming Language [Gos+21], which is similar to, but not a formal superset of the Field Assignment Language (FAL). However, because FAL is explicitly designed to accommodate an easy implementation in Java, this does not account for many issues.

Fully implemented is the Exclusivity Checker (chapter 3) as a completely separate checker, which could also be used on its own. This is described in section 5.1. Further integration with the original Property Checker, especially the property refinement and validation of property-safety (chapter 4) is currently unimplemented. However, in section 5.2 we give an outlook on how such an implementation would look like.

5.1 Exclusivity Checker

The Exclusivity Type System is implemented as a custom checker[1] for the Checker Framework [Pap+08]. As such, it can be used as an annotation processor for any Java program.

Recall the principle design of the Exclusivity Type System: Multiple type rules might be available for the same syntactical construct (e. g. an assignment). Each type rules specifies a refinement that might be used to type the statement. In each step, a type rule is applied that produces a refinement compatible with the declared types.

The main functionality therefore implemented as a *transfer function*, which determines the correct type refinement and applies it to the Checker Framework's internal *store*, which represents the type contexts. To determine which type rule to apply, each one is tried in turn and the first one that is applicable is chosen (this is implemented via a `RuleNotApplicable` exception). The order in which the rules are tried is predefined and relevant, as multiple type rules might be applicable. For example, copying a reference as RO is always possible (as long as compatible with the declared types), but might not be the option yielding the most expressive typing, since RO is the least powerful reference type.

An issue with this design is that the Checker Framework does not allow the reporting of errors from the transfer function: It is intended to simply produce more precise typing judgements, while the actual type checking is performed in the checker component (a part of the transfer function might even be called multiple times, e. g. as part of a fixed-point iteration, which would result in the same error being reported

[1]The implementation is publicly available online: `https://github.com/joshuabach/exclusivity-checker`

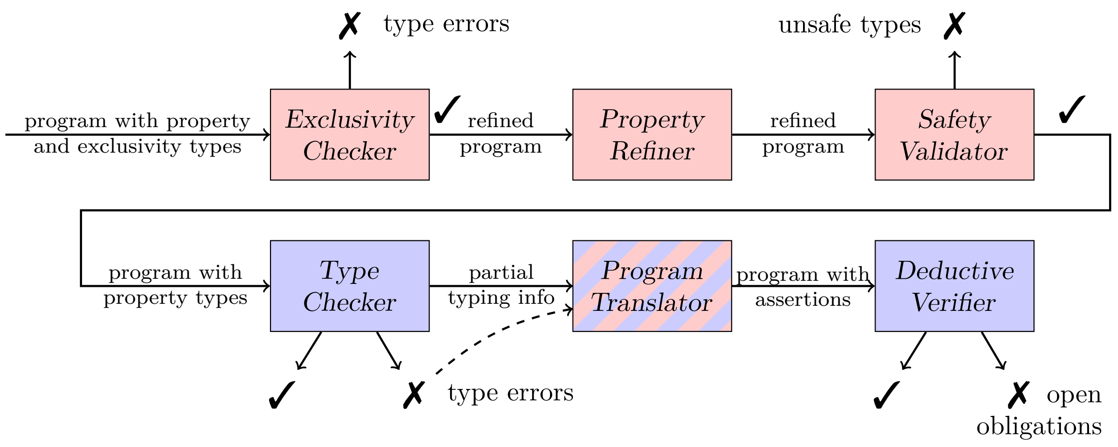

Figure 5.1: Enhanced property-type-checking pipeline.

multiple times). But, when no rule is applicable, our transfer function needs to report an error. To solve this problem, it refines the problematic reference to $\perp$ in such a case. This is later caught during type validation by the type checking component which analyses the store in effect *after* each statement and can report a suitable error.

Our Exclusivity Checker implementation currently supports at least all Java constructs that have a direct equivalent in FAL, such as assignments and method invocations but not others such as binary operators, arrays or string literals. To make the implementation usable for real-world applications, it should be extended to support these, which should be mostly unproblematic and require no further theoretical considerations.

5.2 Outlook: Integration with Property Checker

Based on the exclusivity type annotations it can be determined which references are property-safe. Even more, the Exclusivity Checker has possibly refined the existing exclusivity annotations to allow (refined) properties on even more references. Naturally, after the Exclusivity Checker has run we can be sure that these annotations are correct. With the property type assumption ensured, a checker for our mutable property types could be implemented. While such an implementation does not currently exist, we propose the following design for it.

Figure 5.1 shows our proposed enhanced property-checking pipeline (recall the original in fig. 2.1). While the Exclusivity Checker is implemented as an entirely different checker, the remaining components would all be part of the enhanced Property Checker.

If no type error is found by the Exclusivity Checker, the refined program is passed to the property refiner, which realises the logic from section 4.1 and is implemented in the Checker Framework as a *transfer function*. The resulting refined types are then passed to the safety validator, which ensures that they are valid (definition 15). It is important that the validator also needs to check that only property-safe references are used in properties with classes as base types. The safety validator is implemented as part of the Checker Framework's *type validator*. Only if all types are property-safe, it is safe to pass the refined property type program to the Property Checker. The only change that must be made to the original Property Checker is to the implementation of the program translator, so that it correctly translates the property type transitions to assertions and assumptions. It should be noted that the Property Checker implementation by Lanzinger et al. currently places most of its functionality in the main checker component of the Checker Framework (`TypeVisitor`), leaving the transfer function (`CFTransfer`) free to house the property refiner.

5.3 Test cases

Even though the Exclusivity Checker only supports very Rudimentary java programs, some simple examples can still be presented to illustrate its usage. All examples shown here are passing test cases from our implementation.

In all test cases, the special comment *`// :: error: <key>`* instructs the Checker Framework's testing mechanism that the test case is expected to report an error in the following source line. The error key *`type.invalid`* indicates that the type of a reference has been refined to $\perp$ (because no refinement was applicable). More nuanced and helpful error reporting could be implemented in the future.

Listing 5.1 shows a simple instance of the reference-splitting mechanism. The assignment `a = x` is valid, even though the left-hand side is not a subtype of the right-hand side, because rule T-Ref-Split-Mut can be applied to refine both `a` and `x` to ShrMut. In listing 5.2 we illustrate that even an ExclMut reference can be copied as RO arbitrarily often without losing or violating the exclusivity of the original reference (see rule T-Ref-Copy-Ro). The method called in line `this.mth()` of listing 5.3 might reassign `this.field` (which cannot be determined by the caller), to a non-exclusive reference. Because of this, the refinement of `a` needs to be invalidated to be on the safe side (see rule T-Call). This mechanism could be made more precise by providing a means to specify which fields might be written by a method, similar to JML's `assignable`-clause [Lea+13, section 9.9.9]. Lastly, listing 5.4 illustrates the first assumption of rule T-Assign. Since `this` is RO, it cannot be modified and the field assignment produces an error.

Our implementation contains many more test cases, which we cannot present all here. This small excerpt has been selected to be most illustrative.

```
void splitMut() {
    @ReadOnly Foo x;
    @ShrMut Foo a;
    @ExclMut Foo b;
    x = new Foo();       // x is refined to @ExclMut
    a = x;               // x is updated to @ShrMut
    // :: error: type.invalid
    b = x;               // invalid, x is not @ExclMut anyomre
}
```

Listing 5.1: Splitting an exclusive reference into two shared references.

```
void refCopyRo(@ExclMut Foo a) {
    @ReadOnly Foo x;
    @ReadOnly Foo y;
    @ReadOnly Foo z;
    // a stays @ExclMut for all of these
    x = a; y = a; z = a;
}
```

Listing 5.2: Copying an exclusive reference as read-only.

```
void invalidate(@ExclMut Foo this) {
    @ExclMut Foo a;
    this.field = new Foo(); // field is refined to @ExclMut
    this.mth();
    // :: error: type.invalid
    a = this.field;         // refinement of field has been forgotten
}
```

Listing 5.3: Forgetting of possibly invalid refinements after method invocation.

```
void assignReadOnlyThis(@ReadOnly Foo this) {
    // :: error: assignment.this-not-writable
    this.foo = new Foo();
}
```

Listing 5.4: Cannot assign to field when receiver is read-only.

6 Related Work

In this chapter we discuss approaches similar to ours, or targeting the same problem and organise the bulk of related work based on each of the employed methods. Since our main contribution is the Exclusivity Checker (chapter 3) and thus the bulk of our original work is related to uniqueness and mutability, we will focus on discussion of other approaches to aliasing control. Even further, we give a structured overview of the field and relate the different approaches to each other in order to give a understanding of the relation between the different concepts. For a comparison of the property type approach itself to other work on type systems and formal verification, please refer to the discussion of related work by Lanzinger et al. [Lan+21, section 6]

6.1 Aliasing Mitigation

Reference aliasing is one of computer science' most persistent problems. As such, many approaches have been suggested and developed to control access to and sharing of references. Typically these approaches are divided into *ownership-* and *permission-*based systems, as well as classical *uniqueness* enforcement systems. While these all serve slightly different goals and approach the problem from different angles, they are closely related; Zhao and Boyland [ZB08] even argue that any ownership system can be represented by (fractional) permission types. On the other hand, Östlund et al. [Öst+08] propose an approach based on uniqueness and ownership and claim to be able to encode fractional permission types in their system. Assigning any of these approach to a single category is at best difficult. Nevertheless, we try to group the work reviewed in this section roughly into these categories.

6.1.1 Uniqueness

The oldest approach to limit aliasing is probably the concept of unique references, i. e. the guarantee that no other reference points to the same object. Unique references are sometimes also called *free* or *linear*. A large body of work exists regarding uniqueness which we selectively review here.

One of the first approaches is presented by Hogg [Hog91]. They realise that aliasing is often unproblematic in local contexts well-known to the programmer and only becomes an issue when references escape their set of familiar objects to a scope in which the data structures implicit invariants are less known. They formalise and verify side-effect free methods to build so called *Islands* as a way of encapsulating related sets of objects. Within these islands, objects may be freely aliased, but all accesses from outside an island must go through a single reference. While the term was not yet used at the time, this is already a very rudimentary form of ownership. Hogg also introduces the often cited concept of *destructive reads*: To maintain uniqueness, a unique reference must be invalidated (e. g. set to `null`) when read, to avoid the creation of an alias. Destructive reads are found in one form or another in almost any work on uniqueness and manifests itself as rule T-Ref-Transfer in our Exclusivity Type System.

A even simpler conception of *absolute* uniqueness, by means of completely unsharable objects is presented by Minsky [Min96]. They introduce the concept of methods *consuming* references (making them unusable by the caller unless explicitly returned by the callee), which serves a similar purpose as destructive reads and makes a comeback much later in the Rust programming language [KN19, section 5.3]. Due to the lack of internally aliasable object groups, Minsky's work is more restrictive then Islands, but also more similar to our approach, since we also reason about absolutely unique references with ExclMut.

A major milestone has been laid by Noble, Vitek and Potter [NVP98] with *Flexible Alias Protection*. They further elaborate on the observation that not all aliasing is problematic and even argue that most if not all real issues stem from the visibility of internal changes to other aliases. To remedy this, they adapt information hiding techniques found in many programming languages to alias-protected containers: In a container, fields are divided into *argument* and *representation* fields, separating the container's read-only from its mutable state. The arguments can be thought of as references to other containers relevant to this one. A container may only modify its representation and may only access the read-only state of its arguments. Due to these rules, a container cannot access the mutable state of another container, so problematic state-changes are never visible to aliases. As with Islands, this approach comes very close to the concept of ownership, without naming it as such.

Realising the burden that destructive reads place both on the programmer and the formal systems designer, Boyland [Boy01] proposes a method of enforcing unique variables without destructive reads termed *Alias Burying*. Instead of invalidating a unique variable when it is read, they require all aliases of that variable to be *dead*, i. e. reassigned before being used again, if ever[1]. Boyland argues that while destructive reads require a change to language semantics (a variable is actively invalidated when read), Alias Burying can work as a completely separate check and the program can be compiled and run independently of it. On another note, Boyland observes that most methods never leak their receiver, making uniqueness assertions about it much more simple then for other arguments. This is similar to our restriction in Field Assignment Language (FAL) that `this` may never appear on its own.

As a generalisation of their previous and other work, Boyland, Noble and Retert [BNR01] develop *Capability Sharing*, as a means to provide a generic framework which can be used to model different kinds of variable and field annotations by building all possible combinations of capabilities into a lattice. This is very similar to the capability system which we use to define the exclusivity lattice and even includes

[1]Dead variables can be determined using well-established data-flow analysis techniques, such as *live-variable analysis*.

an *exclusive-write* capability, which almost exactly corresponds to our ExclMut type. Boyland, Noble and Retert even go a step further and include a notion of ownership into the capabilities, making their system more expressive, albeit more complex than ours.

Lastly, the Checker Framework ships a very simple aliasing checker itself [Che22, chapter 26]. It is compromised mainly of a `@Unique` and a `@MaybeAliased` annotation which form a 2-element lattice (with `@Unique` $\sqsubset$ `@MaybeAliased`). References are leaked (and thus uniqueness violated) at assignments, method calls, return and throw statements. Sadly, the checker does not support uniqueness annotation at fields, making it unusable for our purposes. The reason for this is that a field `@Unique Object f` might already be aliased using the two expressions `a.f` and `b.f`, even though `f` itself was never leaked using any of the above expressions. In our work, we circumvent this issue by forbidding access to fields other than `this`, making it easier to argue about field aliasing.

6.1.2 Ownership

The trend in contemporary research has been to move away from classical uniqueness systems and towards more complex approaches based on ownership or permissions, which are also more formally regarded as type systems. While the idea of encapsulating objects and managing access through a singular reference has appeared at least as far back as 1991 [Hog91], ownership types have first been explicitly introduced by Clarke, Potter and Noble [CPN98] in 1998. Since then, the concept has become very popular and has been adapted and repurposed many times. A survey of the landscape of ownership type systems has been performed by Clarke et al. [Cla+13] in 2013. Recently, ownership types have made it into mainstream programming with the Rust programming language [KN19] in 2010, first formally analyzed by Jung et al. [Jun+17].

Here, we review Generic Universe Types (GUT) by Dietl et al. [Die09; DDM12; DM05], which we initially considered as a base for mutable property types, since it is also implemented for Java in the Checker Framework [Pap+08]. GUT is a powerful ownership type system that groups related objects into so called contexts, which form the elements in the ownership tree. Variables annotated with `rep` are owned by the current `this`, variables annotated as `peer` have the same owner as `this` (are siblings in the ownership tree). Additionally, the annotation `any` is used for references with an arbitrary owner. Taken together, `peer` and `any` are similar to the *argument* mode from Flexible Alias Protection. GUT implements an *owners-as-modifier* discipline: Only the current receiver and `peer` and `rep` references are mutable. We ultimately disregarded GUT as a basis for mutable property types, since the inclusion of `peer` references does not sufficiently restrict the set of relevant properties as defined in section 2.3: Each `peer`-reference in a context might still be used to invalidate properties of other references in that context to the same object.

Dietl introduces a concept called *viewpoint adaptation*, which serves as the inspiration of our access adaptation (see definition 12). When a method is called, the type of the argument (which is declared from the callee's point of view) must be translated to the caller's point of view. To put it simple: An argument declared as `peer` on a method with a `rep` receiver must be considered `rep` by the caller, since the owner of `this` in the callee's context is the caller's context. Similarly, we also use access adaptation to combine the types in a transitive field access into a single type. However, our access adaptation is a slightly different concept, because the types in Dietl's viewpoint adaptation are fundamentally interpreted relative to `this`, while our exclusivity types are more based on permissions which are technically independent of the current receiver.

Within the context of GUT, Dietl and Müller [DM13, section 4.4] also introduce the concept of *relevant invariant semantics*. In that work, they study how object ownership can ease program verification. They argue that when analysing invariants regarding a multi-object data structure, which is similar to a property type dependent on some of its subject's fields, a classical *visible state semantics* (that a particular invariant must only hold in the pre- and post-states of each method call) is insufficient. They propose that in this case, a method m called on receiver o may assume and has to preserve the invariants of o and its peers and all objects transitively owned by those objects. In our work, we similarly require each method to specify how it affects the Properties of its arguments and receiver, but further limit the *relevant properties* of an object by restricting the transitivity described above: In FAL, each object may only directly access fields of `this`, any deeper accesses has to be guarded by a method.

6.1.3 Combinations of Ownership and Uniqueness

It can be argued that the concept of ownership *contexts* is a formalisation of the idea found in many uniqueness systems that there be groups in which aliases are permissible.

Clarke and Wrigstad [CW03] double-down on this observation and use an ownership system to reason only about references into a context from outside of it. They ultimately argue that this "External Uniqueness is unique enough". While they once again use destructive reads to achieve uniqueness, they claim that their system could easily be enhanced to incorporate a more sophisticated mechanism, such as Alias Burying [Boy01].

The aforementioned `Joe`$_3$ by Östlund et al. [Öst+08] is a extension of External Uniqueness to incorporate verification of immutability [Pot+13]. While it cannot be used to directly make assertions about the absence of mutable aliases to object referenced mutably, the authors claim that their system can be used to model fractional permissions, which would enable such guarantees.

Bottom Line Throughout most of the work presented in this and the previous section, the common concept emerges that there are certain local contexts within which aliasing is unproblematic and should be permitted. While this may be true, it makes these approaches unusable for our work, since we require total absence of mutable aliases per our definition of property-safety (see definitions 8, 9 and 16).

6.1.4 Permissions

Another approach to aliasing control, which is similar to Boyland, Noble and Retert [BNR01]'s Capabilities, are *permissions.* Contrary to ownership and uniqueness, permission-based approaches argue more about what operations may be performed via a reference at hand, than how the referenced object is arranged within the global object graph.

A very similar goal to ours is pursued by Foster, Terauchi and Aiken [FTA02]. They present *Flow Sensitive Type Qualifiers*, which, akin to property types [Lan+21], also constitute a means to annotate program variables with user-defined properties. To this end, they implement many of the ideas that are included in our work as well, such as the splitting of type contexts into one about declared and one about refined types (called *Store* in their work). To handle aliasing, they perform a conservative *may-alias*-analysis in a first program pass, where each expression result is assigned an abstract location, which is a static approximation of the run time heap objects: Every two expressions that *may* alias refer to the same abstract location, however, two expressions can refer to the same abstract location and never actually alias at run time. In the next step, the type contexts then map locations (instead of expressions) to qualified types, thereby ensuring all possibly relevant qualified types are known for each expression. Further, each abstract location has a specific *linearity* at each program point. The linearity is an element of the lattice $0 \sqsubset 1 \sqsubset \omega$, where 0 represents an unallocated expression, 1 a linear expression and ω a non-linear expression. Whenever a location is reallocated, the linearity increases ($0 \mapsto 1 \mapsto \omega \mapsto \omega \mapsto \cdots$). Thus linearity 1 is similar to our ExclMut: The location is allocated once and the type qualifier may be changed by mutation (strong update). Linearity ω is a more powerful version of our Immut: While Immut references can never be changed in our system, non-linear locations may be mutated, but only insofar as the type qualifier is not changed (weak update).

While not named as such, the linearities used by Foster, Terauchi and Aiken are a sort of *fractional permissions* or *fractional ownership* (each non-linear reference only represents a fraction of the ownership on the whole object).

This concept is mo more explicitly defined in ConSORT [Tom+20]: Here, all references are annotated with ref^n, where $n \in [0;1]$ is the fraction. The semantics can be subsumed as follows:

- ref^0: The reference is read-only, but mutable aliases might exist (similar to our RO).
- ref^n, where $n \in (0;1)$: The reference is read-only and no mutable aliases may exist (similar to our Immut).
- ref^1: The reference is mutable and any aliases are ref^0 (similar to our ExclMut).

ConSORT does not include a counterpart to ShrMut. Whenever a reference ref^n is copied, it is split into two $\text{ref}^{\frac{n}{2}}$. For example, splitting of a ref^1 (ExclMut) results in two $\text{ref}^{\frac{1}{2}}$ (Immut), which corresponds to our rule T-Ref-Split-Immut. However, ConSORT provides a means to combine to references ref^n and ref^m back into ref^{n+m}, making it in particular possible to recover the original ref^1 reference when collecting all the references split from it. This is a concept akin to *borrowing* found in many ownership type systems, which is not possible in our system.

6.2 Pseudo-Languages

Before settling on defining our own in-house pseudo-language FAL, we alternatively considered using already established languages intended for formal reasoning.

The two most obvious candidates are Featherweight Java [IPW01] and Middleweight Java [BPP03], which provide the advantage of being Java subsets. However, while the former is not expressive enough (it is purely functional and in particular does not support assignments, which is paramount for modelling mutability in procedural languages) the latter is already to complex, including many language features analysis of which is out of the scope of this work, such as inheritance and polymorphism.

Another interesting candidate is the Object Calculus by Abadi and Cardelli [AC95], which is a kind of imperative, object-oriented, typed lambda calculus. Because intended explicitly for use in formal analysis to make program reasoning as comfortable as possible, it is in the spirit of formal calculi at its core very simplistic and even simplest high-level constructs such as statement sequencing and let-bindings are built from fundamental primitives. Because our work is explicitly focused on the Java programming language and includes an implementation, the Object Calculus would be unsuitable, since in the end the type rules must be applied to Java statements, which is easier when the fundamental syntactical elements in the theoretical language closely map real Java statements.

Many of the projects discussed in the previous section also define their own pseudo-languages, including only those language constructs relevant and interesting for the problem in question. In this spirit, we also defined our own simple language to avoid the necessity of formalising large amounts of "boilerplate" language constructs, while still staying close the targeted Java programming language.

7 Conclusion

In this thesis, we work towards a complete solution of flow-sensitive property types for mutable data structures, both on a theoretical and an implementational level. To this end, we formulate the notion of *property safety* as a universal assumption enabling the safe verification of property types as defined by Lanzinger et al. [Lan+21] in the presence of aliasing and mutability (section 3.1). We develop the *Exclusivity Type System* (chapter 3), a simple refinement-based type system that can check property safety of references and provide a prototypical implementation (chapter 5). The Exclusivity Type System is also useful on its own as a type system for aliasing control. Further, we extend Lanzinger's original approach for property types to mutable data structures by presenting *mutable property types* (chapter 4), which employ flow-sensitive type refinement to model changing properties.

In the course of our work on the Exclusivity Type System, we found the field of static aliasing control to be very vast and complex, containing many different – and at second glance maybe not so different – approaches based on uniqueness, ownership, permissions or a combination of those. We present a thorough overview of the field and try to sensibly organise the large body of work (chapter 6).

7.1 Discussion

To ease further research on our subject, we review our own work and expose possible drawbacks and disadvantages in this section.

Correctness While we abstractly argue the correctness of the Exclusivity Type System (i. e. that it contains a property-safe type, see theorem 2), we do not give a formal proof verifying the complete type system construction structurally. In fact, we suspect the two most intricate constructions to be likely candidates for posing problems in this regard. Firstly, the removal of possibly invalidated refinements in rule T-Call employs a complex condition that relies on intricate assumptions about the possibility of transitive modification of the current receiver by the invoked method. Secondly, the access adaptation employed in rule T-Mth-Decl, which is based on Dietl's viewpoint adaptation [Die09], differs significantly from its original in some regards. For example, access adaptation only affects types of fields, where viewpoint adaptation is also – and mainly – about method parameter types. While we discuss the distinction between the two concepts in section 6.1.2, it should be further analysed whether an access combination that is closer to the original viewpoint adaptation could increase the exclusivity type's expressiveness and might even be required for soundness.

Analysis of these potential issues is further impeded by the lack of a formal operational semantics of Field Assignment Language (FAL). Arguing about type system soundness without a formalism to model program states and execution flow is imprecise at best.

Implementation The design of our Exclusivity Type System follows the fairly unique approach of basing well-typedness on the applicability of any of the available refinements. The Checker Framework on the other hand follows a more established type system design and clearly separates well-typedness from type refinement, where type refinement is purely optional and merely enhances the precision of the type checker. This discrepancy lead to difficulties in implementing our system with the Checker Framework, as is discussed in section 5.1.

Additionally, the implementation currently only supports a very limited subset of Java. To make it usable for real-world applications further development is required.

Usability Due to the deliberate simplicity of the Exclusivity Type System, it is not obvious whether it is actually expressive enough to model meaningful and sufficiently complex programs. For instance, it would be useful to apply the type transition approach employed for mutable property types to exclusivity types: A method parameter could then e. g. be annotated as `@ExclMut -> @ShrMut` to express that the supplied reference must be ExclMut and will be mutably shared after the method returns. Equally, an annotation like `@ExclMut -> @ExclMut` could then be used to express that the reference is merely “borrowed” and will be available without restriction after the method returns. A thorough evaluation of the expressiveness and usability of the Exclusivity Type System and further research into possible improvements is required.

The key feature of the property type approach is the ability to analyse false positives by the type checker with deductive methods. However, in our approach this is only possible for false positive property type errors. If the Exclusivity Checker is unable to prove an e. g. an ExclMut annotation, there is currently no way to *forward* these cases to the deductive verifier. Exclusivity annotations are much different from property annotations, since they do not follow simple subtyping-rules but require more sophisticated handling of statements that might duplicate references, making this non-trivial. A method of analysing false positive type errors in a ownership type system is studied by Jung et al. [Jun+17] for the Rust programming language. Their work could provide further insight to how such a thing could be allowed for property types.

7.2 Future Work

Independently of the issues discussed in the previous section, further extensions of our work and additional research based on our findings are thinkable. In this section we suggest selective future work that seems most promising to improve our system or build upon it.

First and foremost, the integration with and adaptation of Lanzinger's Property Checker as proposed in section 5.2 should be completed on a theoretical level and then implemented.

While we already provide an overview of research on aliasing control, a more elaborate survey and evaluation of the different approaches is called for. Since many of the approaches are similar in core or even (claim to be) able to model other approaches, a formalisation of a common base is a worthwhile goal. This has partly been done before [ZB08; Öst+08; BNR01], but it seems likely that a more fundamental "common denominator" of uniqueness, ownership and permissions can be established.

Much of the available work on aliasing control rightfully finds that organisation of objects into internally aliasable groups is very useful (e. g. Islands [Hog91] or `peer` contexts in Universe Types [Die09]). However, this conflicts with our design for mutable property types insofar as we require the existence of absolutely (mutably) exclusive references. A possible approach to working around this might be to reason about properties of whole contexts as one, e. g. complete ownership-subtrees. How this is possible and might be formalised should be studied in the future.

Another likely straightforward extensions to mutable property types could be allowing multiply independent property type transitions annotated to a same variable. For example, the list parameter of a method `binaryInsert` might be annotated with both `@Sorted -> @Sorted` and `@MinLength(n) -> @MinLength(n+1)` to express how different possible properties of a reference are affected. Especially when combined with having dependent property types, this promises to yield a significant increase of expressiveness. This extension should be trivial to realise with the Checker Framework and Lanzinger's property type implementation.

List of Figures

List of Tables

List of Type Rules

List of Definitions and Theorems

List of Listings

Bibliography

[AC95] Martin Abadi and Luca Cardelli. “An Imperative Object Calculus”. In: *Theory and Practice of Object Systems* 1.3 (1995), pp. 151–166. DOI: 10.1002/j.1096-9942.1995.tb00016.x.

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt and Mattias Ulbrich. *Deductive Software Verification – The KeY Book*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-49812-6.

[Aut22] Various Authors. *A Dataflow Framework for Java.* 2022. URL: https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf (visited on 08/01/2022).

[BNR01] John Boyland, James Noble and William Retert. “Capabilities for Sharing”. In: *ECOOP – Object-Oriented Programming.* Springer Berlin Heidelberg, 2001, pp. 2–27. DOI: 10.1007/3-540-45337-7_2.

[Boy01] John Boyland. “Alias Burying: Unique Variables Without Destructive Reads”. In: *Software: Practice and Experience* 31.6 (2001), pp. 533–553. DOI: 10.1002/spe.370.

[BPP03] G.M. Bierman, M.J. Parkinson and A.M. Pitts. *MJ: An imperative core calculus for Java and Java with effects.* Tech. rep. UCAM-CL-TR-563. University of Cambridge, Computer Laboratory, Apr. 2003. DOI: 10.48456/tr-563.

[Che22] The Checker Framework Developers. *The Checker Framework Manual. Custom pluggable types for Java.* Version 3.23.0. 2022. URL: https://checkerframework.org/manual/ (visited on 07/25/2022).

[Cla+13] Dave Clarke, Johan Östlund, Ilya Sergey and Tobias Wrigstad. “Ownership Types: A Survey”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 15–58. DOI: 10.1007/978-3-642-36946-9_3.

[CPN98] David G. Clarke, John M. Potter and James Noble. “Ownership Types for Flexible Alias Protection”. In: *ACM SIGPLAN Notices* 33.10 (Oct. 1998), pp. 48–64. ISSN: 0362-1340. DOI: 10.1145/286942.286947.

[CW03] Dave Clarke and Tobias Wrigstad. “External Uniqueness Is Unique Enough”. In: *ECOOP – Object-Oriented Programming.* Springer Berlin Heidelberg, 2003, pp. 176–200. DOI: 10.1007/978-3-540-45070-2_9.

[DDM12] Werner Dietl, Sophia Drossopoulou and Peter Müller. “Separating Ownership Topology and Encapsulation With Generic Universe Types”. In: *ACM Transactions on Programming Languages and Systems* 33.6 (2012), pp. 1–62. DOI: `10.1145/2049706.2049709`.

[Die+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu and Todd W. Schiller. “Building and Using Pluggable Type-Checkers”. In: *Proceedings of the 33rd International Conference on Software Engineering – ICSE*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 681–690. DOI: `10.1145/1985793.1985889`.

[Die09] Werner Dietl. “Universe Types: Topology, encapsulation, genericity, and tools”. PhD thesis. ETH Zurich, 2009. DOI: `10.3929/ethz-a-005951213`.

[DM05] Werner Dietl and Peter Müller. “Universes: Lightweight Ownership for JML”. In: *The Journal of Object Technology* 4.8 (2005), p. 5. DOI: `10.5381/jot.2005.4.8.a1`.

[DM13] Werner Dietl and Peter Müller. “Object Ownership in Program Verification”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 289–318. DOI: `10.1007/978-3-642-36946-9_11`.

[FTA02] Jeffrey S. Foster, Tachio Terauchi and Alex Aiken. “Flow-Sensitive Type Qualifiers”. In: *ACM SIGPLAN Notices* 37.5 (May 2002), pp. 1–12. ISSN: 0362-1340. DOI: `10.1145/543552.512531`.

[Gos+21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith and Gavin Bierman, eds. *The Java Language Specification*. Java SE 17. 2021. URL: `https://docs.oracle.com/javase/specs/jls/se17/html/index.html` (visited on 07/29/2022).

[Hog91] John Hogg. “Islands: Aliasing Protection in Object-Oriented Languages”. In: *ACM SIGPLAN Notices* 26.11 (Nov. 1991), pp. 271–285. ISSN: 0362-1340. DOI: `10.1145/118014.117975`.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce and Philip Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: `10.1145/503502.503505`.

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers and Derek Dreyer. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: *Proceedings of the ACM on Programming Languages*. Vol. 2. Association for Computing Machinery, Dec. 2017. DOI: `10.1145/3158154`.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. Covers Rust 1.59 or later. No Starch Press, 2019. ISBN: 9781718500440. URL: `https://doc.rust-lang.org/stable/book/`.

[Lan+21] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich and Werner Dietl. “Scalability and Precision by Combining Expressive Type Systems and Deductive Verification”. In: *Proceedings of the ACM on Programming Languages*. Vol. 5. 143. Association for Computing Machinery, Oct. 2021. DOI: 10.1145/3485520.

[Lea+13] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmermann and Werner Dietl, eds. *Java Modeling Languagee (JML) Reference Manual*. 1st ed. 2013. URL: https://www.cs.ucf.edu/~leavens/JML/jmlrefman (visited on 07/27/2022).

[Min96] Naftaly H. Minsky. “Towards alias-free pointers”. In: *ECOOP – Object-Oriented Programming. 10th European Conference, Proceedings*. Ed. by Pierre Cointe. Springer Berlin Heidelberg, 1996, pp. 189–209. DOI: 10.1007/BFb0053062.

[NVP98] James Noble, Jan Vitek and John Potter. “Flexible alias protection”. In: *ECOOP – Object-Oriented Programming. 12th European Conference, Proceedings*. Ed. by Eric Jul. Springer Berlin Heidelberg, 1998, pp. 158–185. DOI: 10.1007/BFb0054091.

[Öst+08] Johan Östlund, Tobias Wrigstad, Dave Clarke and Beatrice Åkerblom. “Ownership, Uniqueness, and Immutability”. In: *Objects, Components, Models and Patterns. 46th International Conference, TOOLS EUROPE 2008, Proceedings*. Springer Berlin Heidelberg, 2008, pp. 178–197. DOI: 10.1007/978-3-540-69824-1_11.

[Pap+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins and Michael D. Ernst. “Practical Pluggable Types for Java”. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 201–212. ISBN: 9781605580500. DOI: 10.1145/1390630.1390656.

[Pot+13] Alex Potanin, Johan Östlund, Yoav Zibin and Michael D. Ernst. “Immutability”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 233–269. DOI: 10.1007/978-3-642-36946-9_9.

[Tom+20] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi and Naoki Kobayashi. “ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs”. In: *Programming Languages and Systems. 29th European Symposium on Programming (ESOP), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*. Springer International Publishing, 2020, pp. 684–714. DOI: 10.1007/978-3-030-44914-8_25.

[ZB08] Yang Zhao and John Boyland. “A Fundamental Permission Interpretation for Ownership Types”. In: *2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*. June 2008. DOI: 10.1109/tase.2008.45.