Reference Mutability for DOT

Vlastimil Dort
Charles University, Prague, Czech Republic
dort@d3s.mff.cuni.cz

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

Abstract

Reference mutability is a type-based technique for controlling mutation that has been thoroughly studied in Java. We explore how reference mutability interacts with the features of Scala by adding it to the Dependent Object Types (DOT) calculus. Our extension shows how reference mutability can be encoded using existing Scala features such as path-dependent, intersection, and union types. We prove type soundness and the immutability guarantee provided by our calculus.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Object oriented languages

Keywords and phrases Reference Mutability, Read-only References, DOT Calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.18

Funding This work was partially supported by the Czech Science Foundation project 18-17403S, partially supported by the Mobility Fund of Charles University, and partially supported by the Charles University institutional funding project SVV 260588. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The Scala programming language integrates functional and object-oriented programming, making available many of the benefits of both paradigms. One important benefit of purely functional programming is referential transparency, which makes reasoning about program behaviour easier, both for human programmers and for automated optimizers and verifiers. Nevertheless, Scala does not provide any verifiable way to specify which parts of a program are purely functional. Purity is an absence of all side effects; in this paper, we focus on mutation of objects in the heap as one specific but important side effect.

Reference mutability, also called reference immutability [19, 10], has been studied especially in Java as a way to control mutation. References to objects are classified as either read-write or read-only, and writes to fields through a read-only reference are forbidden. This applies transitively: when a reference is read from the field of an object through a read-only reference, the newly-read reference is made read-only as well. As a result, if all parameters of a function are read-only (and if there are no accesses to global variables), the function must be pure in the sense that it cannot modify any state in the heap that existed before it was called, although it does have the ability to allocate and mutate new objects.

As in related work [19, 20], we distinguish reference mutability from the stronger guarantee of object immutability: a reference mutability system like ours ensures that an object is not mutated through a read-only reference, but it is still possible for the object to change if it is aliased by other, read-write references. For the same reason, we avoid calling references *mutable* or *immutable*, since it is not the reference that can be mutated, but the object that it refers to, and even the referent of a read-only reference is not necessarily immutable. By the same reasoning, it would be more accurate to speak of the *read-only-ness* of a reference rather than of its mutability, but we use the latter term because the former is awkward.

Our goal is to bring read-only references to Scala. An empirical study [8] showed that about 35 to 70 percent of classes in large Scala codebases are either deeply or shallowly immutable. One challenge is the complexity of Scala and its type system relative to Java, and the interaction of reference mutability with Scala language features. In particular, Scala programs frequently use nested functions that have access to variables in outer scopes. A second challenge is that for maintainability and ease of adoption, we seek a system that integrates well with Scala's existing type system. Reference mutability implementations for Java add entirely new type systems on top of Java's type system. Since Scala's type system already provides powerful and expressive features, it ought to be possible to use those features to implement at least parts of a reference mutability system, and we explore the feasibility of such an approach. By reusing existing features as much as possible, we aim for an implementation that would require few changes to an existing Scala compiler so that it could be easily maintained as the compiler evolves.

When we began this project, we explored designs by prototyping them in the Dotty compiler for Scala 3. The subtle conceptual errors that we encountered revealed the need for a more principled approach. Our contribution in this paper is a formalization of our design as an extension of the Dependent Object Types (DOT) calculus [2, 16, 14], a core calculus modelling the essence of Scala. We prove type soundness and the immutability property guaranteed by our extended DOT calculus, which we call roDOT. roDOT can serve as a foundation for a correct implementation of reference mutability for the full Scala language.

The rest of the paper is organized as follows. In Section 2, we present a baseline DOT calculus that we will extend with reference mutability. We overview our approach in Section 3: we identify the requirements needed from a type system to implement a useful reference mutability system, discuss how the features of DOT can partially satisfy the requirements, and introduce the changes that we make to DOT to fulfill the requirements. In Section 4, we present roDOT, the formal DOT calculus extended with reference mutability. In Section 5, we define properties of roDOT, namely type soundness and the immutability guarantee, and discuss their proofs. We survey related work in Section 6 and conclude in Section 7.

2 Baseline DOT

In this section, we present a baseline DOT calculus that we will later extend with reference mutability. There are multiple variants of DOT calculi in the literature. Our baseline is close to kDOT [12], which in turn is based Wadlerfest DOT [2], with the following main differences. In Wadlerfest DOT, objects are immutable. The semantics operates on plain program terms, with objects in the heap represented by let-binding terms. In kDOT, objects have mutable fields, which can be re-assigned to hold a reference to another object on the heap. The semantics of kDOT operates on configurations with an explicit heap, a program term being reduced, and a stack of continuations. We adopt these features of kDOT in our baseline DOT. In addition, kDOT defines constructors to model the gradual initialization of the fields of each object, like in Scala. We omit constructors from our baseline DOT because they are not necessary to model reference mutability. Thus, our baseline DOT is a variant of kDOT without constructors.

The syntax of the baseline DOT is in Figure 1. As in WadlerFest DOT and kDOT, the literals are objects and lambda abstractions. An object has a self parameter s (modelling the `this` keyword in Scala) and a sequence d of member definitions. Object members are either fields, which store a term that is reduced after each read of the field, or type members. As in kDOT, a literal can only appear in a let-binding of the form $\mathsf{let}\, z = l \,\mathsf{in}\, t$, which ensures that every literal can be referred to by some variable z.

$x ::=$	**Variable**
$\mid z$	local
$\mid s$	self
$\mid y$	location
$d ::=$	**Definition**
$\mid \{a = t\}$	field
$\mid \{A = T\}$	type
$\mid d_1 \wedge d_2$	aggregate
$\Gamma ::=$	**Context**
$\mid$	empty
$\mid \Gamma, x : T$	binding

$l ::=$	**Literal**
$\mid \nu(s : T)d$	object
$\mid \lambda(z : T)t$	lambda
$t ::=$	**Term**
$\mid \mathsf{v}x$	var
$\mid \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2$	let
$\mid \mathsf{let}\ z = l\ \mathsf{in}\ t$	let-lit
$\mid x_1.a := x_2$	write
$\mid x.a$	read
$\mid x_1 x_2$	apply

$T ::=$	**Type**
$\mid \top$	top
$\mid \bot$	bottom
$\mid \forall(z : T_1)T_2$	function
$\mid \mu(s : T)$	recursive
$\mid \{a : T_1..T_2\}$	field decl
$\mid \{A : T_1..T_2\}$	type decl
$\mid x.A$	projection
$\mid T_1 \wedge T_2$	intersection

Figure 1 Baseline DOT syntax.

$\Sigma ::=$	**Heap**
$\mid \cdot$	empty heap
$\mid \Sigma, y \to l$	heap object
$\sigma ::=$	**Stack**
$\mid \cdot$	empty stack
$\mid \mathsf{let}\ z = \square\ \mathsf{in}\ t :: \sigma$	let frame
$\mathrm{F} ::=$	**Inert context**
$\mid$	empty
$\mid \mathrm{F}, y : S$	binding

$c ::=$	**Configuration**
$\mid \langle t; \sigma; \Sigma \rangle$	
$Q ::=$	**Member type**
$\mid \{a : T..T\}$	tight field decl
$\mid \{A : T..T\}$	tight type decl
$R ::=$	**Record type**
$\mid Q$	member
$\mid R_1 \wedge R_2$	intersection
$S ::=$	**Inert type**
$\mid \forall(z : T_1)T_2$	function
$\mid \mu(s : R)$	object

Figure 2 Baseline DOT runtime syntax.

In the baseline DOT, we distinguish between local variables z that are bound by let terms and lambdas, self variables s that are bound by object literals, and heap locations y that represent addresses in the runtime heap. Heap locations cannot appear in the surface syntax; they are created only in runtime configurations in the operational semantics.

A program is expressed as a term, which, if correctly typed, reduces to a location of a single item on the heap. The meanings of the terms are standard. We use the notation $\mathsf{v}x$ to make it explicit when a variable is used as a term. A let term evaluates one term and substitutes the result into another term. Terms in DOT are in A-normal form (ANF), in that subterms of a term are generally variables, not arbitrary terms. Thus, every non-trivial term must be evaluated and assigned to a variable in a let binding before it can be used in a later term. A write term changes the value of a field of an object on the heap. A read term returns the last value written to a field, or the term value given to the field when the object was created. An apply term applies a lambda by substituting the argument into its body.

Types must be explicitly specified in lambda abstractions and object literals. Lambdas have a function type, which allows them to be applied in an apply term. Objects have a recursive type containing an intersection of field or type declaration types corresponding to the definitions forming the object. The recursive type allows the declarations to refer to other members of the object. An intersection type is a common subtype of two types. As in kDOT, a field declaration type specifies two types for a field, a setter and a getter type. The getter type is given to a read term that reads the field, while the setter is the type a variable must have so that it can be written to the field. A type declaration type specifies the lower and upper bounds for a type member, which can be referred to by type selection. The top type is a supertype of every type; the bottom type is a subtype of every type.

Evaluation of a program is described as reduction of an initial configuration, which consists of a term, an empty stack, and empty heap. The heap binds locations y to literals l. The heap is modified by creating a new object using the let-lit term, or by changing the value of a field using the write term. In a final configuration, the stack is empty and the term is in normal form – a location of a single item on the heap.

In a typed configuration, *heap correspondence* ensures that for each location in the context, the heap contains a function or an object of the specified type. For objects it means that if $\Gamma(y) = \mu(s : R)$, then $\Sigma(y) = \nu(s : R)d$, where $\Gamma \vdash d : [y/s]R$. The type R is syntactically the same in Γ and Σ.

Type soundness ensures that evaluation of a typed term either progresses indefinitely or reaches a final configuration. A key ingredient of the type soundness proof is the definition of inert typing contexts. A concrete object in the heap holds a specific term in each field and a specific type in each type member, so it is possible to type such an object with a type in which all member types are tight: each type declaration type $\{A : T..T\}$ has equal upper and lower bounds T and each field declaration type $\{a : T..T\}$ has equal getter and setter types T. Such types with tight bounds are called inert, and many theorems about DOT calculi hold only in typing contexts containing only inert types [14]. A progress theorem proves that if a configuration can be typed in a typing context that gives an inert type for each object in the heap, then it is in a normal form or steps to another configuration. A preservation theorem proves that the resulting configuration after the step can still be given the same type in an inert context that corresponds to the possibly updated heap.

3 Overview

In this section, we examine how the baseline DOT system can be extended to support read-only references and overview the path to roDOT.

3.1 Requirements

A type system for reference mutability should satisfy the following requirements:

Keeping expressiveness. The type system should admit programs that do not use the new features similarly to the baseline DOT.

Mutability constraints. Additionally, the type system should provide a way to distinguish between read-write and read-only references. Read-write references should be convertible to read-only references, but not the other way. In a reference mutability type system, the distinction can be achieved by making each type read-write or read-only, with read-write a subtype of read-only.

Integration with type system features. The extensions should use existing features of the DOT type system where possible, and not interfere with them.

Type soundness. The extensions should not make the type system unsound - as in kDOT, each typed program should reduce to an answer or run indefinitely.

Guarantee of immutability. The type system should guarantee that only read-write references are used to mutate objects. This guarantee should be transitive: starting from a read-only reference, the system should prevent mutation of any other objects reached by any sequence of field reads. To achieve this, if a read term reads from a field of an object through a read-only reference, the result of the read should be given a read-only type, even if the field contains a read-write reference. This change in the type of the reference is called viewpoint adaptation [10].

Mutability polymorphism. Previous work [19, 10] demonstrated the importance of methods that are polymorphic in the mutability of the receiver. Consider a getter method that reads a field of an object. When called on a read-only reference, the method can obtain only a read-only reference from the field due to viewpoint adaptation, and thus its return type must be read-only. But, when the same method is called on a read-write receiver, it can read a read-write reference from the field, and its return type should reflect this.

3.2 Example

As an example, we will encode an object with a field a and getter and setter methods m_g and m_s for that field. In Scala, such an object would be created by instantiating a class:

```
class C {
  var a: T = _
  def m_s(z: T): Unit = {a = z}
  def m_g: T = a
}
```

In the baseline DOT, such an object can be created by a let statement:
$\mathsf{let}\, z = \nu(s : T_o)\{a = x\} \wedge \{m_s = \mathsf{let}\, z_1 = \lambda(z : T)s.a := z \,\mathsf{in}\, z_1\} \wedge \{m_g = s.a\} \,\mathsf{in}\, t$, where $T_o \triangleq \mu(s : \{a : T\} \wedge \{m_s : \forall(z : T)\top\} \wedge \{m_g : T\})$ and x is an initial value of type T.

The m_s method mutates the contents, so a reference mutability type system should prevent calling it on a read-only reference. The m_g method should be polymorphic in the mutability of the receiver. After we present the roDOT calculus, we will show how this example can be encoded in it in Section 4.5.

3.3 Changes to the Calculus

We now summarize the changes to the baseline DOT to support read-only references. We motivate each change briefly here, but defer a detailed justification to Section 4.

3.3.1 Mutability Types

First, we must decide how to distinguish read-write and read-only types. A straightforward way is to define some special marker type M to designate read-write references. Any type can be made read-write by intersecting it with M. This satisfies our requirement that a read-write type should be a subtype of the read-only version of the same type.

We can test whether a type is read-write by testing whether it is a subtype of M. For a reference x, we define an operation $\Gamma \vdash \textbf{isrw}\ x$ as a typing judgment $\Gamma \vdash x : M$. To make a read-write version of an existing type T, we define an operation $\mathsf{rw}(T) \equiv T \wedge M$.

3.3.2 Dependent Mutability

Second, we need some mechanism to implement mutability polymorphism. We will use a dependent mutability type, defined to have the same mutability as some specified reference.

We can achieve this with a careful choice of the read-write marker type: we reserve a type member name M for mutability, and choose $M \equiv \{\mathsf{M} : \bot..\bot\}$. This choice makes it possible for a type to depend on the mutability of a reference x using the type selection $x.\mathsf{M}$. The type $\{\mathsf{M} : \bot..x.\mathsf{M}\}$ is read-write (in the sense of being a subtype of $\{\mathsf{M} : \bot..\bot\}$) if (and in an inert context only if) the reference x is read-write (has type $\{\mathsf{M} : \bot..\bot\}$).

3.3.3 Viewpoint Adaptation

Third, the type system requires an operation $\Gamma \vdash x \triangleright T \rightarrow T'$ that performs the viewpoint adaptation described above. Given a reference x and a type T of a field, it returns T if x is a read-write reference, but a read-only version of T if x is a read-only reference. This operation could also be composed from two simpler operations: making a read-only version of T and combining the mutability of T and x.

While a read-write version of a given type can be made with a simple intersection, the opposite of making a read-only version cannot be done using any of the type operations in the baseline DOT. If it is already the case that $T <: \{\mathsf{M} : \bot..\bot\}$, none of the operations removes this subtyping relationship while otherwise keeping T unchanged. Therefore we need a new relation **ro** that makes a read-only version of a type. We will define $\Gamma \vdash T\,\mathbf{ro}\,T'$ by recursion on the syntax of T, so that T' is a supertype of T, but not a subtype of $\{\mathsf{M} : \bot..\bot\}$.

We also need to combine the mutabilities of x and T. The mutability of x can be expressed using the type selection $x.\mathsf{M}$, but to determine the mutability of T, we need to define another relation **mu** similar to **ro**. In $\Gamma \vdash T\,\mathbf{mu}\,T'$, the type T' is a lower bound for the special type member M given by T. We will define the **ro** and **mu** relations in detail in Section 4.2.

Using these new type operators, we can define viewpoint adaptation as $\Gamma \vdash x \triangleright T \rightarrow T_{\mathrm{r}} \wedge \{\mathsf{M} : \bot..T_{\mathrm{m}} \vee x.\mathsf{M}\}$, where $\Gamma \vdash T\,\mathbf{ro}\,T_{\mathrm{r}}$ and $\Gamma \vdash T\,\mathbf{mu}\,T_{\mathrm{m}}$. Notice that the upper bound $T_{\mathrm{m}} \vee x.\mathsf{M}$ is a union type. To implement viewpoint adaptation, we therefore need to extend DOT with union types. Union types are a feature of Scala 3 and were studied in some variants of DOT [16, 3], but not in kDOT, from which our baseline DOT is derived.

Note that even with union types added, **ro** cannot be implemented by a simple union $T \vee \mathit{notM}$ of T with some fixed read-only type notM, because the set of members of such a union type is the intersection of the members of T and notM, so the union type would not have all the members of T.

3.3.4 Recursive Types

If we were to allow the self type T in a recursive type $\mu(s : T)$ to be read-write, an object of such a type would be inherently mutable, i.e., viewpoint adaption would not be able to create a read-only reference to it. This is because the read-write type $\mu(s : T \wedge \{\mathsf{M} : \bot..\bot\})$ is not a subtype of the read-only variant $\mu(s : T)$, for there is no subtyping between recursive types in DOT.

This means that the mutability of an object must be expressed *outside* the recursive type as $\mu(s : T) \wedge \{\mathsf{M} : \bot..\bot\}$, as opposed to inside as as $\mu(s : T \wedge \{\mathsf{M} : \bot..\bot\})$.

We also require the self type T to not refer to $s.\mathsf{M}$, the mutability of the self variable. Otherwise, the mutability could be stored in a type member such as $\{A : s.\mathsf{M}..\bot\}$, from which we could infer $s.\mathsf{M} <: \bot$, which would again make the object inherently mutable.

3.3.5 Methods

In a type of a mutability-polymorphic method, such as the getter m_{g} from the example, we want to specify its return type to be dependent on the mutability of the receiver. When the method is called on a read-write receiver reference, the type of the return value will become read-write as well, because of the mutability-dependent return type.

In the baseline DOT, a method is encoded as a function value stored in a field of an object. Given that the declaration of a field is typed with a self-variable s in scope, it would seem natural to use $s.\mathsf{M}$ for defining methods with return types polymorphic in the mutability of the receiver. For example, $\{m : \forall(z : \top)(\{a : \top\} \wedge \{\mathsf{M} : \bot..s.\mathsf{M}\})\}$ would be a method that returns a read-write referenece to an object with field a when called on a read-write receiver, but returns a read-only reference to the same object when called on a read-only receiver.

The problem with this encoding is that the dependent mutability $s.\mathsf{M}$ in the return type would refer to the mutability of the *object* that the method is contained in, not to the mutability of the *reference* to that object through which the method is called. Distinguishing these two concepts requires a rather complicated example, which we will present in Section 4.1.

To distinguish these concepts in roDOT, we introduce a new kind of variable r to represent the receiver reference and write it as an explicit additional parameter of each method. The type given to this parameter decides its mutability. In polymorphic methods, we type it as read-only, so that the method can be called on either a read-write or a read-only receiver.

Furthermore, the baseline DOT splits the typing of a method invocation into two steps: the first step reads the function value from the field and the second step applies the function to an argument. This two-step process separates the receiver reference, which is present only in the first step, from the function application in the second step. Thus, the mutability of the result can no longer polymorphically depend on the mutability of the receiver reference.

To overcome this problem, we need to unify method selection and method invocation into one step, so that the type of the method invocation can depend on the type of the receiver. We extend the baseline DOT with an explicit method construct. A method is called in a single step using a term of the form $x_1.m\, x_2$, which selects the method from the receiver x_1 and applies it to an argument x_2. A method type then has the form $\{m(z : T_1, r : T_3) : T_2\}$.

Visibility

If a method captures a variable from its surrounding environment, it can write to the object that the variable refers to even if it is called on a read-only receiver and with a read-only argument. To prevent this, we hide variables other than the receiver and method parameter in the typing context when typing the body of a method, so the method cannot capture them. Despite this restriction, it is still possible to encode a method that captures variables as follows: the captured variables are copied into fields of the containing object. This workaround ensures that viewpoint adaptation is applied when the captured variable is read out of the field of the object.

3.3.6 Reference Variables

In the baseline DOT, a reference value is a location y, the unique identifier of some item in the heap. To state and prove roDOT immutability guarantee, we need to distinguish read-only and read-write references to the same object in a runtime configuration. We therefore extend the calculus with references w. Two references w, w' may designate the same location y, but can have different mutabilities in the typing context Γ. Runtime configurations are extended with an environment ρ that maps each reference w to the location y that it designates.

In summary, the baseline DOT distinguishes three kinds of variables: local variables and method parameters z, recursive self variables s, and heap locations y. To these three, roDOT adds receiver variables r and reference variables w.

4 Type System

In this section, we present the formal definition of roDOT.

4.1 Syntax

The syntax of roDOT is defined in Figure 3. Changes from the baseline DOT are highlighted using shading.

Terms are formed by the same syntax as in the baseline DOT, except that the function application syntax is replaced by a method call syntax and lambda literals are replaced by method definitions with the ordinary parameter z and receiver parameter r. Since the calculus no longer needs lambda literals, all literals are objects, so we inline them into the let-lit term. Furthermore, the values of fields are variables x rather than arbitrary terms t. Terms were needed in the baseline DOT to allow fields to hold lambdas to encode methods.

Variables x are classified as either abstract variables u, which are bound in terms and definitions in the initial program, or global variables v, which are generated during reduction and are used in the heap and the runtime environment. They are not expected to be used in the initial term. Abstract variables are either general local variables z, object self variables s, or method call receivers r. Global variables are either heap locations y or references to heap locations w. A typing context Γ can give a type to variables of any kind.

To support viewpoint adaptation, we add union types. They are dual to intersection types and make it possible to define a distributive subtyping rule for intersections of type member types (ST-TypAnd in Figure 4), which makes it possible to combine multiple mutability declarations into one. A new type N is a read-only version of $\bot$.

x ::=	**Variable**
| u	abstract
| v	global
u ::=	**Abstract**
| z	local
| s	self
| r	receiver
v ::=	**Global**
| y	location
| w	reference
t ::=	**Term**
| vx	var
| let $z = t_1$ in t_2	let
| let $z = \nu(s : T)d$ in t	let-lit
| $x_1.a := x_2$	write
| $x.a$	read
| $x_1.m\, x_2$	call
B ::=	**Type name**
| A	type member
| M	mutability

Γ ::=	**Context**
|	empty
| $\Gamma, x : T$	binding
| $\Gamma, !$	hide
d ::=	**Definition**
| $\{a = x\}$	field
| $\{m(z, r) = t\}$	method
| $\{A(r) = T\}$	type
| $d_1 \wedge d_2$	aggregate
T ::=	**Type**
| $\top$	top
| $\bot$	bottom
| $\mu(s : T)$	recursive
| $\{a : T_1..T_2\}$	field decl
| $\{m(z : T_1, r : T_3) : T_2\}$	method decl
| $\{B(r) : T_1..T_2\}$	type decl
| $x_1.B(x_2)$	projection
| $T_1 \wedge T_2$	intersection
| $T_1 \vee T_2$	union
| N	read-only $\bot$

Figure 3 Syntax.

4.1.1 Methods

Method declarations bind the parameter variable z and the receiver variable r. In the corresponding definition, we omit the types, because they are not needed. Each method has only one parameter other than the receiver. Multiple values can be passed to a method by wrapping them in an object and passing a reference to the object as the argument.

To motivate the dedicated syntax for methods, consider an object that contains a method a that returns a reference with the same mutability $s.\mathsf{M}$ as the receiver that it is called on. In the syntax of the baseline DOT, this object could have the type $T \triangleq \mu(s : \{a : \forall(z : T')\{\mathsf{M} : \bot..s.\mathsf{M}\}\})$. Suppose x_1 is a read-write reference to such an object; it has type $T \wedge \{\mathsf{M} : \bot..\bot\}$. Now suppose x_1 is copied to another reference x_2 that is read-only. The reference can be made read-only in various ways: one way is to store x_1 into a field of some other object y, and then read it back into x_2 through a read-only reference to y, so viewpoint adaptation will make the type of x_2 read-only. But, even though x_2 is read-only (i.e., $x_2.\mathsf{M}$ is not a subtype of $\bot$), x_2 still has the same field types copied from x_1. In particular, x_1 has the type $T \wedge \{\mathsf{M} : \bot..\bot\}$, the type T, the type $\{a : \forall(z : T')\{\mathsf{M} : \bot..x_1.\mathsf{M}\}\}$ (by VT-RecE), and the type $\{a : \forall(z : T')\{\mathsf{M} : \bot..\bot\}\}$ (by ST-SelU). Even though x_2 is read-only, it still also has the latter field types. Thus, the expression $x_2.a$ can be typed as a function with a read-write return type, even though the receiver x_2 is read-only.

If we introduced methods, but without the receiver variable r, the type of the object would be $T \triangleq \mu(s : \{m(z : T') : \{\mathsf{M} : \bot..s.\mathsf{M}\}\})$. Suppose again that x_1 is a read-write reference to the object, which is copied to x_2 and made read-only. As before, the VT-RecE rule can be applied to the type of x_1 before it is made read-only, so x_1 has the type $T \wedge \{\mathsf{M} : \bot..\bot\}$, the type T, the type $\{m(z : T') : \{\mathsf{M} : \bot..x_1.\mathsf{M}\}\}$, and the type $\{m(z : T') : \{\mathsf{M} : \bot..\bot\}\}$. Even though x_2 is a read-only reference, it still also has the latter method types, and thus the method can return a read-write reference even when called on the read-only receiver x_2.

To avoid these problems, we prohibit references to the mutability $s.\mathsf{M}$ of the self object reference s in all definitions; the return type can only refer to the mutability of r, the receiver reference on which the method will be called.

One may wonder whether we could have achieved the same thing without changing the baseline DOT syntax by encoding a method with receiver r and parameter z using currying as $\{a = \lambda(r : T_3)\lambda(z : T_1)t\}$. The method would then be called on receiver r with argument x as $(r.a\ r)\ x$, i.e., the receiver r would have to be repeated. The problem with this encoding is that in the type of an object, we have to write the types of the object's methods, and those method types contain the type for the receiver, which should be the type of the object itself. Thus, the type of an object would have to recursively include itself, and thus the structure of the type would be infinite. On the other hand, with the explicit syntax for method types ($\{m(z : T_1, r : T_3) : T_2\}$), we can resolve this issue in the typing rule for method definitions: when we add the receiver r to the typing context for typing the method body, we add it not just with the specified type T_3, but with an intersection of T_3 with the self type of the object containing the method. This removes the need to recursively repeat that self type inside T_3.

4.1.2 Type Members

Type members can have either an ordinary name A, or the special mutability member name M, which cannot be defined in an object literal. We write B in places where both A and M can be used. In the formal syntax, all type members have a receiver parameter r with no type specified: the general form is $\{B(r) : T_1..T_2\}$. In a type selection, an argument for this parameter must be provided, and is substituted into the bounds T_1 and T_2. To reduce clutter, we omit writing the receiver parameter when it is not used in the bounds.

The M is a special member used as the mutability marker. A type $\{\mathsf{M} : \bot..T\}$ is read-write if $T <: \bot$ and read-only otherwise. Only the upper bound of M is significant for mutability because the lower bound is always $\bot$. A dual encoding would be equally possible, in which the lower bound would be significant and the upper bound would always be $\top$, so a read-write type would be expressed by $\{\mathsf{M} : \top..\top\}$.

The receiver parameter r allows making a generic method type, where the mutability of the result is determined by a type member of the containing object. For example, the mutability of the return type of the method in $\mu(s : \{m(z : \top, r : \top) : s.A(r)\})$ depends on A. It is read-write if the object defines $\{A(r) = \{\mathsf{M} : \bot..\bot\}\}$, read-only if the object defines $\{A(r) = \{\mathsf{M} : \bot..\top\}\}$, and polymorphic if the object defines $\{A(r) = \{\mathsf{M} : \bot..r.\mathsf{M}\}\}$.

4.2 Typing

The typing and subtyping rules are shown in Figures 4 to 6. The rules are obtained from the baseline DOT by applying the syntactic changes and by adding additional rules. As in the baseline DOT, typing rules for variables are separated from typing rules for terms.

The typing rules apply both to determine which initial terms are valid programs and to give types to intermediate terms during reduction in order to prove type safety. When used for this second purpose, the terms may contain type selections on reference variables such as $w.A$. For two references w and w' to the same location y, types $w.A$, $w'.A$ and $y.A$ are considered equivalent. In order to achieve this, subtyping and typing statements have the environment ρ on the left-hand side. The environment is passed around in all the typing rules, but only used in ST-Eq, which states that if two types only differ in references, then they are subtypes. It has no effect on typing of initial program terms, because those do not contain references w and are typed with empty ρ.

4.2.1 Subtyping Rules

The ST-Refl and ST-Trans rules ensure that subtyping is a preorder, and the ST-Top and ST-Bot rules establish $\top$ and $\bot$ as the maximum and minimum elements. The ST-Or* and ST-And* rules define the usual properties of unions and intersections. ST-Dist makes them distribute and ST-TypAnd allows merging bounds of type members in intersections.

The ST-Typ, ST-Met and ST-Fld rules allow subtyping between declaration types. In ST-Met, the parameter is in the typing context for subtyping of the receiver types, and both the parameter and receiver are in the context for subtyping of the result types. In ST-Typ, the r parameters do not have bounds and are not added to the typing context for subtyping. It is important for the proofs that in the tight-subtyping variant of this rule, the typing contexts remain inert.

The ST-Met rule is a counterpart of a subtyping rule for function types in DOT, which plays a major role in DOT being conjectured to be undecidable [9]. Correspondingly, in roDOT, this rule makes it difficult to test whether a particular type is read-write or read-only.

The ST-Sel* rules give bounds to type selections. They substitute the provided argument for the receiver parameter.

The ST-N-M rule makes $\bot$ the greatest lower bound of N and $\{\mathsf{M} : \bot..\bot\}$, expressing that nothing can be both read-write and read-only. The other ST-N-* rules establish N as a lower bound of read-only types. As in the baseline DOT, there is no subtyping between different recursive types.

$$\frac{}{\Gamma;\rho \vdash T <: T}\,(\text{ST-Refl})$$

$$\frac{\Gamma;\rho \vdash T_1 <: T_2 \quad \Gamma;\rho \vdash T_2 <: T_3}{\Gamma;\rho \vdash T_1 <: T_3}\,(\text{ST-Trans})$$

$$\frac{}{\Gamma;\rho \vdash T <: \top}\,(\text{ST-Top})$$

$$\frac{}{\Gamma;\rho \vdash \bot <: T}\,(\text{ST-Bot})$$

$$\frac{\rho \vdash T_1 \approx T_2}{\Gamma;\rho \vdash T_1 <: T_2}\,(\text{ST-Eq})$$

$$\frac{}{\Gamma;\rho \vdash T_1 <: T_1 \vee T_2}\,(\text{ST-Or1})$$

$$\frac{}{\Gamma;\rho \vdash T_2 <: T_1 \vee T_2}\,(\text{ST-Or2})$$

$$\frac{\Gamma;\rho \vdash T_1 <: T_3 \quad \Gamma;\rho \vdash T_2 <: T_3}{\Gamma;\rho \vdash T_1 \vee T_2 <: T_3}\,(\text{ST-Or})$$

$$\frac{\Gamma;\rho \vdash T_1 <: T_2 \quad \Gamma;\rho \vdash T_1 <: T_3}{\Gamma;\rho \vdash T_1 <: T_2 \wedge T_3}\,(\text{ST-And})$$

$$\frac{}{\Gamma;\rho \vdash T_1 \wedge T_2 <: T_1}\,(\text{ST-And1})$$

$$\frac{}{\Gamma;\rho \vdash T_1 \wedge T_2 <: T_2}\,(\text{ST-And2})$$

$$\frac{\Gamma;\rho \vdash x : \{B(r) : T_1..T_2\}}{\Gamma;\rho \vdash [x_2/r]T_1 <: x.B(x_2)}\,(\text{ST-SelL})$$

$$\frac{\Gamma;\rho \vdash x : \{B(r) : T_1..T_2\}}{\Gamma;\rho \vdash x.B(x_2) <: [x_2/r]T_2}\,(\text{ST-SelU})$$

$$\frac{\Gamma;\rho \vdash T_3 <: T_1 \quad \Gamma;\rho \vdash T_2 <: T_4}{\Gamma;\rho \vdash \{B(r) : T_1..T_2\} <: \{B(r) : T_3..T_4\}}\,(\text{ST-Typ})$$

$$\frac{\Gamma;\rho \vdash T_3 <: T_1 \quad \Gamma;\rho \vdash T_2 <: T_4}{\Gamma;\rho \vdash \{a : T_1..T_2\} <: \{a : T_3..T_4\}}\,(\text{ST-Fld})$$

$$\frac{}{\Gamma;\rho \vdash \mathsf{N} \wedge \{\mathsf{M}(r) : \bot..\bot\} <: \bot}\,(\text{ST-N-M})$$

$$\frac{}{\Gamma;\rho \vdash \mathsf{N} <: \mu(s : T)}\,(\text{ST-N-Rec})$$

$$\frac{}{\Gamma;\rho \vdash \mathsf{N} <: \{a : T_1..T_2\}}\,(\text{ST-N-Fld})$$

$$\frac{}{\Gamma;\rho \vdash \mathsf{N} <: \{A(r) : T_1..T_2\}}\,(\text{ST-N-Typ})$$

$$\frac{\Gamma;\rho \vdash T_3 <: T_1 \quad \Gamma, z : T_3;\rho \vdash T_6 <: T_5 \quad \Gamma, z : T_3, r : T_6;\rho \vdash T_2 <: T_4}{\Gamma;\rho \vdash \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}}\,(\text{ST-Met})$$

$$\frac{}{\Gamma;\rho \vdash \mathsf{N} <: \{m(z : T_1, r : T_3) : T_2\}}\,(\text{ST-N-Met})$$

$$\frac{}{\Gamma;\rho \vdash \{B(r) : T_1..T_2\} \wedge \{B(r) : T_3..T_4\} <: \{B(r) : T_1 \vee T_3..T_2 \wedge T_4\}}\,(\text{ST-TypAnd})$$

$$\frac{}{\Gamma;\rho \vdash T_1 \wedge (T_2 \vee T_3) <: (T_1 \wedge T_2) \vee (T_1 \wedge T_3)}\,(\text{ST-Dist})$$

Figure 4 Subtyping.

4.2.2 Variable Typing Rules

The VT-Var rule gives variables the type assigned by the typing context, VT-Sub adds subsumption and VT-AndI gives variables intersection types.

The typing rules VT-RecE and VT-RecI allow opening and closing recursive types. Both rules require that the inner type T is independent of the mutability $s.\mathsf{M}$ of s, written $T\,\mathbf{indep}\,s$. The introduction rule additionally requires the inner type to be read-only by requiring that the **ro** operation does not change the original type.

To ensure that the type selection $x.\mathsf{M}$ is valid for every variable x, we add the axiom VT-MutTop, which gives every variable the type $\{\mathsf{M} : \bot..\top\}$.

4.2.3 Term Typing Rules

Typing of terms requires that all variables occurring free in a term, but not as a part of a type (we call them t-free), are visible, as discussed in Section 3.3. For each such occurrence, there is a premise $\Gamma\ \textbf{vis}\ x$ in the corresponding rule. By Vis-Var, only the variables after the ! are visible for term typing. Variables in the context before the ! can still be used for type selection. This separation makes use of our explicit notation for a variable used as a term, written $\mathsf{v}x$, and typed using only the part of the context after the !. A plain variable x that appears in a type selection $x.A$ is typed using the full typing context.

The TT-Var rule gives types given by variable typing to visible variables. TT-Sub adds subsumption for term types. TT-Let types let terms as in the baseline DOT.

$$\frac{! \notin \Gamma_2}{\Gamma_1, x : T, \Gamma_2\ \textbf{vis}\ x}\ \text{(Vis-Var)}$$

$$\frac{\Gamma = \Gamma_1, x : T, \Gamma_2}{\Gamma;\rho \vdash x : T}\ \text{(VT-Var)}$$

$$\frac{\Gamma;\rho \vdash x : T_1 \quad \Gamma;\rho \vdash T_1 <: T_2}{\Gamma;\rho \vdash x : T_2}\ \text{(VT-Sub)}$$

$$\frac{\Gamma;\rho \vdash x : T_1 \quad \Gamma;\rho \vdash x : T_2}{\Gamma;\rho \vdash x : T_1 \wedge T_2}\ \text{(VT-AndI)}$$

$$\frac{\Gamma;\rho \vdash x : T \quad \Gamma\ \textbf{vis}\ x}{\Gamma;\rho \vdash \mathsf{v}x : T}\ \text{(TT-Var)}$$

$$\frac{\Gamma;\rho \vdash t : T_1 \quad \Gamma;\rho \vdash T_1 <: T_2}{\Gamma;\rho \vdash t : T_2}\ \text{(TT-Sub)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash t_1 : T_1 \quad z \notin \mathrm{fv}\ T_2 \\ \Gamma, z : T_1;\rho \vdash t_2 : T_2\end{array}}{\Gamma;\rho \vdash \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2 : T_2}\ \text{(TT-Let)}$$

$$\frac{\begin{array}{c}\Gamma, s : T_1;\rho \vdash d : T_1 \quad z \notin \mathrm{fv}\ T_2 \quad T_1\ \textbf{indep}\ s \\ \Gamma, z : \mu(s : T_1) \wedge \{\mathsf{M}(r) : \bot..\bot\};\rho \vdash t : T_2\end{array}}{\Gamma;\rho \vdash \mathsf{let}\ z = \nu(s : T_1)d\ \mathsf{in}\ t : T_2}\ \text{(TT-New)}$$

$$\frac{\begin{array}{cc}\Gamma;\rho \vdash x : \{a : T_1..T_2\} & \Gamma\ \textbf{vis}\ x \\ \Gamma;\rho \vdash T_2\ \textbf{ro}\ T_3 & \Gamma;\rho \vdash T_2\ \textbf{mu}(r)\ T_4\end{array}}{\Gamma;\rho \vdash x.a : T_3 \wedge \{\mathsf{M}(r) : \bot..(T_4 \vee x.\mathsf{M}(r))\}}\ \text{(TT-Read)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : \mu(s : T) \\ T\ \textbf{indep}\ s\end{array}}{\Gamma;\rho \vdash x : [x/s]T}\ \text{(VT-RecE)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : [x/s]T \\ T\ \textbf{indep}\ s \\ \Gamma;\rho \vdash [x/s]T\ \textbf{ro}\ [x/s]T\end{array}}{\Gamma;\rho \vdash x : \mu(s : T)}\ \text{(VT-RecI)}$$

$$\frac{\Gamma;\rho \vdash x : T}{\Gamma;\rho \vdash x : \{\mathsf{M}(r_0) : \bot..\top\}}\ \text{(VT-MutTop)}$$

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\Gamma;\rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\}} \\ \Gamma;\rho \vdash x_2 : T_1 & \Gamma\ \textbf{vis}\ x_2 \\ \Gamma;\rho \vdash x_1 : [x_2/z]T_3 & \Gamma\ \textbf{vis}\ x_1\end{array}}{\Gamma;\rho \vdash x_1.m\,x_2 : [x_1/r][x_2/z]T_2}\ \text{(TT-Apply)}$$

$$\frac{\begin{array}{cc}\Gamma;\rho \vdash x_1 : T_1 & \Gamma\ \textbf{vis}\ x_1 \\ \Gamma;\rho \vdash x : \{a : T_1..T_2\} & \Gamma\ \textbf{vis}\ x \\ \multicolumn{2}{c}{\Gamma;\rho \vdash x : \{\mathsf{M}(r) : \bot..\bot\}}\end{array}}{\Gamma;\rho \vdash x.a := x_1 : T_2}\ \text{(TT-Write)}$$

Figure 5 Typing.

$$\frac{}{\Gamma, s : T_4;\rho \vdash \{A(r) = T\} : \{A(r) : T..T\}}\text{(DT-Typ)}$$

$$\frac{}{\Gamma, s : T_4;\rho \vdash \{A(r) = T\} : \{A(r) : \bot..T\}}\text{(DT-TypB)}$$

$$\frac{\Gamma, s : T_4;\rho \vdash x : T \quad \Gamma, s : T_4 \textbf{ vis } x}{\Gamma, s : T_4;\rho \vdash \{a = x\} : \{a : T..T\}}\text{(DT-Fld)}$$

$$\frac{\begin{array}{c}\Gamma, s : T_4;\rho \vdash d_1 : T_1 \quad \Gamma, s : T_4;\rho \vdash d_2 : T_2 \\ d_1 \text{ and } d_2 \text{ have distinct member names}\end{array}}{\Gamma, s : T_4;\rho \vdash d_1 \wedge d_2 : T_1 \wedge T_2}\text{(DT-And)}$$

$$\frac{\begin{array}{c}z \notin \text{fv } T_1 \cup \text{fv } T_4, r \notin \text{fv } T_3 \cup \text{fv } T_1 \cup \text{fv } T_4 \\ \Gamma, s : T_4, !, z : T_1, r : T_4 \wedge [r/s]T_4 \wedge T_3;\rho \vdash t : T_2\end{array}}{\Gamma, s : T_4;\rho \vdash \{m(z, r) = t\} : \{m(z : T_1, r : T_3) : T_2\}}\text{(DT-Met)}$$

Figure 6 Definition typing.

The TT-New rule should give a read-write type to every constructed object. Therefore, the type of z in the context for typing t in $\mathsf{let}\, z = \nu(s : T_1)\,\mathsf{in}\, t$ is changed to $\mathsf{rw}(\mu(s : T_1))$. The type T_1 written for s must correspond to the definitions and cannot refer to $s.\mathsf{M}$. Because objects are given a recursive type, this corresponds to the requirement that recursive types must always be read-only.

The TT-Apply rule for method calls substitutes both the parameter and the receiver into the result type. The type declared for r restricts the type of the receiver. The typing rule for method application checks that both the receiver and the argument have the expected type. For example, if the receiver parameter type is declared to have a read-write type, the method can be called only on read-write references.

In TT-Write, we add a premise to ensure that the reference whose field we are mutating is read-write: $x : \{\mathsf{M} : \bot..\bot\}$.

Finally, in TT-Read, we need to viewpoint-adapt the type T_2 of the field with the mutability of the reference x through which we are reading the field. For $x : \{a : T_1..T_2\}$, we change the type of $x.a$ from T_2 to T_5, and add a premise $\Gamma \vdash x \triangleright T_2 \rightarrow T_5$.

Viewpoint Adaptation

In Section 3.3, we described how viewpoint adaptation can be expressed in terms of two new type relations $\Gamma \vdash T\, \textbf{ro}\, T_\mathrm{r}$ and $\Gamma \vdash T\, \textbf{mu}\, T_\mathrm{m}$. The relations are defined together in Figure 7. The operation $\Gamma \vdash T\, \textbf{ro}\, T_\mathrm{r}$ means that T_r is a supertype of T that is definitely read-only. The **ro** relation extracts the parts of a type other than mutability. Thus, the relation maps the mutability type member type $\{\mathsf{M} : T..T'\}$ to $\top$. The relation is the identity on the $\top$ type, field and method declarations, and type declarations other than M. Because we want $T <: T_\mathrm{r}$ whenever $\Gamma \vdash T\, \textbf{ro}\, T_\mathrm{r}$, the relation must also be the identity on recursive object types, because they do not participate in any subtyping relationships other than reflexivity and subtyping with $\bot$ and $\top$. To enforce this, the typing rule for recursion introduction needs to ensure that the self type T is read-only.

$$\frac{}{\begin{array}{c}\Gamma;\rho \vdash \top\ \mathbf{ro}\ \top \\ \Gamma;\rho \vdash \top\ \mathbf{mu}(r)\ \top\end{array}}\text{(TS-Top)}$$

$$\frac{}{\begin{array}{c}\Gamma;\rho \vdash \bot\ \mathbf{ro}\ \mathsf{N} \\ \Gamma;\rho \vdash \bot\ \mathbf{mu}(r)\ \bot\end{array}}\text{(TS-Bot)}$$

$$\frac{T = \{A(r) : T_1..T_2\}}{\begin{array}{c}\Gamma;\rho \vdash T\ \mathbf{ro}\ T \\ \Gamma;\rho \vdash T\ \mathbf{mu}(r_0)\ \top\end{array}}\text{(TS-Typ)}$$

$$\frac{T = \{m(z : T_1, r : T_3) : T_2\}}{\begin{array}{c}\Gamma;\rho \vdash T\ \mathbf{ro}\ T \\ \Gamma;\rho \vdash T\ \mathbf{mu}(r_0)\ \top\end{array}}\text{(TS-Met)}$$

$$\frac{T = \{a : T_1..T_2\}}{\begin{array}{c}\Gamma;\rho \vdash T\ \mathbf{ro}\ T \\ \Gamma;\rho \vdash T\ \mathbf{mu}(r)\ \top\end{array}}\text{(TS-Fld)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : \{B(r) : T_1..T_2\} \\ \Gamma;\rho \vdash [x_2/r]T_2\ \mathbf{ro}\ T_3 \\ \Gamma;\rho \vdash [x_2/r]T_2\ \mathbf{mu}(r_0)\ T_4\end{array}}{\begin{array}{c}\Gamma;\rho \vdash x.B(x_2)\ \mathbf{ro}\ T_3 \\ \Gamma;\rho \vdash x.B(x_2)\ \mathbf{mu}(r_0)\ T_4\end{array}}\text{(TS-Sel)}$$

$$\frac{}{\begin{array}{c}\Gamma;\rho \vdash \{\mathsf{M}(r) : T_1..T_2\}\ \mathbf{ro}\ \top \\ \Gamma;\rho \vdash \{\mathsf{M}(r) : T_1..T_2\}\ \mathbf{mu}(r)\ T_2\end{array}}\text{(TS-M)}$$

$$\frac{T = \mu(s : T_1)}{\begin{array}{c}\Gamma;\rho \vdash T\ \mathbf{ro}\ T \\ \Gamma;\rho \vdash T\ \mathbf{mu}(r)\ \top\end{array}}\text{(TS-Rec)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1\ \mathbf{ro}\ T_2 \\ \Gamma;\rho \vdash T_3\ \mathbf{ro}\ T_4\end{array}}{\Gamma;\rho \vdash T_1 \wedge T_3\ \mathbf{ro}\ T_2 \wedge T_4}\text{(TS-AndR)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1\ \mathbf{mu}(r)\ T_2 \\ \Gamma;\rho \vdash T_3\ \mathbf{mu}(r)\ T_4\end{array}}{\Gamma;\rho \vdash T_1 \wedge T_3\ \mathbf{mu}(r)\ T_2 \wedge T_4}\text{(TS-AndM)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1\ \mathbf{ro}\ T_2 \\ \Gamma;\rho \vdash T_3\ \mathbf{ro}\ T_4\end{array}}{\Gamma;\rho \vdash T_1 \vee T_3\ \mathbf{ro}\ T_2 \vee T_4}\text{(TS-OrR)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1\ \mathbf{mu}(r)\ T_2 \\ \Gamma;\rho \vdash T_3\ \mathbf{mu}(r)\ T_4\end{array}}{\Gamma;\rho \vdash T_1 \vee T_3\ \mathbf{mu}(r)\ T_2 \vee T_4}\text{(TS-OrM)}$$

Figure 7 Splitting relations.

On intersection and union types, the **ro** relation is defined recursively on the two parts of the type. For a type selection $x.B(x_2)$, **ro** is applied recursively to the upper bound of B in the type of x. For the bottom type $\bot$, **ro** cannot simply return $\bot$ itself because $\bot$ is read-write since $\bot <: \{\mathsf{M} : \bot..\bot\}$. We make N a subtype only of types that are definitely known to be read-only, including declaration types other than M and all recursive types.

The **mu** relation is defined to return T_2 for $\{\mathsf{M}(r) : \bot..T_2\}$, to recurse on intersection and union types and into the upper bound of a type selection, and to return $\top$ for all other (read-only) types. Because in TS-M, the type T_2 may refer to the receiver r, the **mu** relation is parameterized by a variable that binds to this receiver. This variable is used in the declaration of M in the viewpoint-adapted type in TT-Read.

4.2.4 Definition Typing

Definition typing, shown in Figure 6 (DT-*), is only used in the context of the TT-New rule, where the self reference s is the last variable in the typing context. Singling out this variable from the rest of the typing context is important for the DT-Met rule, in order to give r a type derived from the type of the object.

Typing of field definitions DT-Fld allows using s as the value of the field. It requires the value of the field to be visible, and gives the field a type with tight bounds. Typing type members allows tight bounds, but we also allow fixing just the upper bound and leaving the lower bound to be $\bot$. This allows declarations of ordinary type members to be similar to the declarations of the mutability member M, which always have $\bot$ as the lower bound.

In DT-Met, z is given the parameter type specified in the method declaration, but the type for r is formed by intersecting the declared type with the type T_4 given to s in TT-New. The rule looks up the type of s in the context and gives the same type to r. Additionally, a version of T_4 with s replaced by r is added to the intersection, allowing deriving the recursive type $\mu(s : T_4)$ for r.

Variables other than the parameter and the receiver are hidden from the context and not allowed to be used as a value in the method. That is achieved in DT-Met by splitting the typing context for the method body into two parts separated with an ! symbol.

4.3 Runtime Configuration

The syntax of runtime configurations is shown in Figure 8. A machine configuration c consists of a focus of execution t, stack s, runtime environment ρ and heap Σ. Each frame of the stack is a let term with a hole $\Box$ into which the reduced focus of execution will be substituted. The runtime environment ρ is a new part of a configuration, which maps references w to the locations y to which they refer.

Because the only items in the heap are objects, we omit the header $\nu(s : R)$ and store only the definition d, which is an intersection of field, method and type member definitions. The values of fields of heap objects are restricted to only locations y by heap correspondence. Since each object in the heap is at a known location y, we substitute this location y for any occurrences of the self variable s in the member definitions.

Valid configurations are given a type under an inert context F. The rules for typing configurations are given in Figure 9. Stack typing assigns to each stack a pair of types, an input type T_1 and an output type T_3, indicating that if the focus of execution reduces to a value of type T_1, then the entire stack will reduce to a value of type T_3. The environment ρ must correspond to the typing context, meaning that each reference w corresponding to a location y under ρ must appear after y in F and have the same type except for mutability. The heap must correspond to F, which requires that for every location y in F, an object has to exist on the heap, and the object must have the correct type with y substituted for s. Finally, to type a configuration, the CT-Corr rule checks environment correspondence and heap correspondence, then types the focus of execution t and the stack σ, checks that the type of the focus of execution matches the input type of the stack, and finally gives the output type of the stack to the entire configuration.

$\Sigma ::=$	**Heap**
$\mid \cdot$	empty heap
$\mid \Sigma, y \to d$	heap object
$\sigma ::=$	**Stack**
$\mid \cdot$	empty stack
$\mid \mathsf{let}\ z = \Box\ \mathsf{in}\ t :: \sigma$	let frame
$\rho ::=$	**Environment**
$\mid \cdot$	empty environment
$\mid \rho, w \to y$	assignment
$c ::=$	**Configuration**
$\mid \langle t; \sigma; \rho; \Sigma \rangle$	
$\Gamma_h ::=$	**Heap Context**
$\mid \Gamma, y/s : R$	

$Q ::=$	**Member type**
$\mid \{a : T..T\}$	tight field
$\mid \{m(z : T_1, r : T_3) : T_2\}$	method
$\mid \{A(r) : T..T\}$	tight type
$\mid \{A(r) : \bot..T\}$	upper-bounded type
$R ::=$	**Record type**
$\mid Q$	member
$\mid R_1 \wedge R_2$	intersection
$S ::=$	**Inert type**
$\mid \mu(s : R) \wedge \{\mathsf{M}(r) : \bot..T\}$	object
$\mathrm{F} ::=$	**Inert context**
$\mid$	empty
$\mid \mathrm{F}, y : S$	binding

Figure 8 Runtime.

$$\frac{\mathrm{F};\rho \vdash T_1 <: T_2}{\mathrm{F};\rho \vdash \cdot : T_1, T_2}\,(\text{CT-EmptyS}) \qquad \frac{\begin{array}{c}\mathrm{F}, z : T_1;\rho \vdash t : T_2 \\ z \notin \text{fv } T_2 \quad \mathrm{F};\rho \vdash \sigma : T_2, T_3\end{array}}{\mathrm{F};\rho \vdash \mathsf{let}\ z = \Box\ \mathsf{in}\ t :: \sigma : T_1, T_3}\,(\text{CT-LetS})$$

$$\frac{}{\mathrm{F};\rho \vdash \sim \cdot}\,(\text{CT-EmptyH})$$

$$\frac{\mathrm{F}_1;\rho \vdash \mathrm{F}_2 \sim \Sigma}{\mathrm{F}_1;\rho \vdash \mathrm{F}_2, w : T \sim \Sigma}\,(\text{CT-RefH})$$

$$\frac{}{\Gamma \sim \cdot}\,(\text{CT-EmptyE})$$

$$\frac{\mathrm{F}_1, y/s : R;\rho \vdash d : [y/s]R \quad \mathrm{F}_1;\rho \vdash \mathrm{F}_2 \sim \Sigma \quad R\ \textbf{indep}\ s}{\mathrm{F}_1;\rho \vdash \mathrm{F}_2, y : \mu(s : R) \wedge \{\mathsf{M}(r) : \bot..\bot\} \sim \Sigma, y \to d}\,(\text{CT-ObjH})$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1, w : \mu(s : R) \wedge \{\mathsf{M}(r) : \bot..T\}, \Gamma_2 \quad \Gamma_1 \sim \rho \\ \Gamma_1 = \Gamma_3, y : \mu(s : R) \wedge \{\mathsf{M}(r) : \bot..\bot\}, \Gamma_4\end{array}}{\Gamma \sim \rho, w \to y}\,(\text{CT-RefE})$$

$$\frac{\begin{array}{c}\mathrm{F};\rho \vdash \mathrm{F} \sim \Sigma \\ \text{all fields in } \Sigma \text{ are locations}\end{array}}{\mathrm{F};\rho \sim \Sigma}\,(\text{CT-CorrH}) \qquad \frac{\begin{array}{c}\mathrm{F} \sim \rho \qquad \mathrm{F};\rho \sim \Sigma \\ \mathrm{F};\rho \vdash t : T_1 \quad \mathrm{F};\rho \vdash \sigma : T_1, T_2 \\ \text{no locations in } t \text{ and } \sigma\end{array}}{\mathrm{F} \vdash \langle t; \sigma; \rho; \Sigma\rangle : T_2}\,(\text{CT-Corr})$$

$$\frac{\begin{array}{c}z \notin \text{fv } T_1 \cup \text{fv } T_4, r \notin \text{fv } T_3 \cup \text{fv } T_1 \cup \text{fv } T_4 \\ \Gamma, !, z : T_1, r : [y/s]T_4 \wedge [r/s]T_4 \wedge T_3;\rho \vdash t : T_2\end{array}}{\Gamma, y/s : T_4;\rho \vdash \{m(z, r) = t\} : \{m(z : T_1, r : T_3) : T_2\}}\,(\text{HT-Met})$$

Figure 9 Configuration typing and correspondence.

4.3.1 Environment

When a new object is created, both a fresh location y and a fresh reference w are created, with the same read-write type. The location y is put on the heap, the reference w is put into the focus of execution, and w is connected to y by the environment ρ. When writing a reference w to a field, the corresponding location y is stored on the heap. Its mutability is determined by the type of the field. When reading the value of field a from a reference w_1, a new reference w_2 is created for the location y_2 stored in the field of the object stored at location $y_1 = \rho(w_1)$ on the heap. The new reference w_2 is given the type of y_2 with the mutability changed by viewpoint adaptation to be an upper bound of the mutability of the field and of the reference w_1.

4.3.2 Heap Correspondence

The heap correspondence relation checks that the type of each location y in the typing context corresponds to the object stored at y in the heap.

The type of y in the context is the read-write version of the type specified when creating the object. That is, when a literal $\nu(s : T)d$ leads to creating y on the heap, then $\Sigma(y) = [y/s]d$ and $\mathrm{F}(y) = \mu(s : T) \wedge \{\mathsf{M} : \bot..\bot\}$.

To check that the definition of the object corresponds to its type, we define a modified definition typing. The baseline DOT uses the same rules for typing object literals in let statements and typing heap items in heap correspondence. In roDOT, to preserve typing after the substitution, the type of r in the context for method bodies must be changed to use y instead of s in T_4. Because of that, we have a set of definition typing rules for heap items HT-*, similar to the set of definition typing rules DT-* in Figure 6. They give types to definitions on the heap in a *heap context* with the special syntax $\Gamma, y/s : T_4$. We show only the HT-Met rule which differs from DT-Met in that it removes s from the typing context and substitutes y for s in T_4. The reason for this is so that in the HT-Met rule shown in Figure 6, r can be given the types $[y/s]T_4$ and $[r/s]T_4$. Other HT-* rules not shown here are similar to the DT-* rules, except that HT-Fld does not put s into the context for typing x.

4.4 Reduction

Reduction is defined in Figure 10. There is a reduction step for each kind of term, which produces the next configuration. We change the reduction from the baseline DOT to use reference variables w to represent references in runtime configurations, rather than directly using the locations y. Rules that access the heap have additional premises that relate the references with the corresponding locations in the environment. If the term is a single variable and the stack is empty, then no step can be taken and the evaluation ends in a final configuration. The R-LetPush and R-LetLoc rules work with the stack in the same way as in the baseline DOT. The R-Write rule overwrites the value of a field on the heap. In a R-Read step, a new reference to an object is created in the focus for a location that was stored in a field. The R-Apply rule is changed to apply a method of an object instead of a function value. It substitutes both the parameter and the receiver into the method body and proceeds to reduce it. In a R-LetNew step, the heap is extended with a new object y and the environment is extended with a new reference w with the same read-write type. The definition of the object on the heap is constructed from the provided object literal by replacing all references by corresponding locations and replacing the self variable by y.

$$\frac{y_1 \to \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma \quad w_1 \to y_1 \in \rho_1 \quad \rho_2 = \rho_1, w_2 \to y_2 \quad (w_2\ \textbf{fresh})}{\langle w_1.a; \sigma; \rho_1; \Sigma\rangle \longmapsto \langle \mathsf{v}w_2; \sigma; \rho_2; \Sigma\rangle}\ \text{(R-Read)}$$

$$\frac{\begin{array}{ll} w_1 \to y_1 \in \rho & y_1 \to \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma_1 \\ w_3 \to y_3 \in \rho & \Sigma_2 = \Sigma_1[y_1 \to \ldots_1 \{a = y_3\} \ldots_2] \end{array}}{\langle w_1.a := w_3; \sigma; \rho; \Sigma_1\rangle \longmapsto \langle \mathsf{v}w_3; \sigma; \rho; \Sigma_2\rangle}\ \text{(R-Write)}$$

$$\frac{w_1 \to y_1 \in \rho \quad y_1 \to \ldots_1 \{m(z, r) = t\} \ldots_2 \in \Sigma}{\langle w_1.m\, w_2; \sigma; \rho; \Sigma\rangle \longmapsto \langle [w_1/r][w_2/z]t; \sigma; \rho; \Sigma\rangle}\ \text{(R-Apply)}$$

$$\frac{\rho_2 = \rho_1, w \to y \quad \Sigma_2 = \Sigma_1, y \to [y/s][\rho_1]d \quad (y, w\ \textbf{fresh})}{\langle \mathsf{let}\ z = \nu(s : T)d\ \mathsf{in}\ t; \sigma; \rho_1; \Sigma_1\rangle \longmapsto \langle [w/z]t; \sigma; \rho_2; \Sigma_2\rangle}\ \text{(R-LetNew)}$$

$$\frac{}{\langle \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2; \sigma; \rho; \Sigma\rangle \longmapsto \langle t_1; \mathsf{let}\ z = \square\ \mathsf{in}\ t_2 :: \sigma; \rho; \Sigma\rangle}\ \text{(R-LetPush)}$$

$$\frac{}{\langle \mathsf{v}w; \mathsf{let}\ z = \square\ \mathsf{in}\ t :: \sigma; \rho; \Sigma\rangle \longmapsto \langle [w/z]t; \sigma; \rho; \Sigma\rangle}\ \text{(R-LetLoc)}$$

Figure 10 Reduction.

$$\frac{\mathrm{F} \vdash \langle t; \sigma; \rho; \Sigma\rangle \ \mathbf{mreach}\ y_1 \quad y_1 \to \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma \quad \mathrm{F};\rho \vdash y_1 : \{a : \bot..\{\mathsf{M}(r) : \bot..\bot\}\}}{\mathrm{F} \vdash \langle t; \sigma; \rho; \Sigma\rangle \ \mathbf{mreach}\ y_2}\text{(Rea-Fld)}$$

$$\frac{t \ \mathbf{tfree}\ w \lor \sigma \ \mathbf{tfree}\ w \quad w \to y \in \rho \quad \mathrm{F};\rho \vdash w : \{\mathsf{M}(r) : \bot..\bot\}}{\mathrm{F} \vdash \langle t; \sigma; \rho; \Sigma\rangle \ \mathbf{mreach}\ y}\text{(Rea-Term)}$$

Figure 11 Mutable reachability.

4.5 Example

In roDOT, we can rewrite the example from Section 3.2 with the intended mutability types.

Assume that T is a read-only type in the sense that $\Gamma \vdash T \, \mathbf{ro} \, T$. Then we can let the field have a read-write type by adding the mutability marker. By using the mutability marker as the type of r in m_s, we can express that the setter can only be called on read-write references. By adding $\{\mathsf{M}(r_0) : \bot..r.\mathsf{M}\}$ to the result type of m_g, we ensure that the getter only returns a read-write reference if called on a read-write reference. The type of the object will be $T_{o1} \triangleq \mu(s : \{a : T \land \{\mathsf{M} : \bot..\bot\}\} \land \{m_\mathrm{s}(z : T \land \{\mathsf{M} : \bot..\bot\}, r : \{\mathsf{M} : \bot..\bot\}) : \top\} \land \{m_\mathrm{g}(z : \top, r : \top) : T \land \{\mathsf{M}(r_0) : \bot..r.\mathsf{M}(r_0)\}\})$. Given an initial value x of type $T \land \{\mathsf{M} : \bot..\bot\}$, it can be instantiated in $\mathsf{let}\, z = \nu(s : T_{o1})\{a = x\} \land \{m_\mathrm{s}(r, z) = r.a := z\} \land \{m_\mathrm{g}(r, z) = r.a\} \, \mathsf{in}\, t$.

To allow storing read-only values in the a field as well, we can parameterize the mutability of the field using a type member A. The type of the object will then be: $T_{o2} \triangleq \mu(s : \{A : \bot..\top\} \land \{a : T \land \{\mathsf{M}(r_0) : \bot..s.A(r_0)\}\} \land \{m_\mathrm{s}(z : T \land \{\mathsf{M}(r_0) : \bot..s.A(r_0)\}, r : \{\mathsf{M} : \bot..\bot\}) : \top\} \land \{m_\mathrm{g}(z : \top, r : \top) : T \land \{\mathsf{M}(r_0) : \bot..r.\mathsf{M}(r_0) \lor s.A(r)\}\})$.

5 Properties and their Proofs

In this section we state and prove type soundness of roDOT and the immutability guarantee.

5.1 Immutability Guarantee

We define the immutability guarantee that roDOT provides as follows: an object on the heap Σ in a configuration $c = \langle t; \sigma; \rho; \Sigma\rangle$ typed under F can be modified in an execution starting from c only if it is *mutably reachable*, i.e., reachable from the configuration using only read-write references. Mutable reachability is formally defined in Figure 11. By the Rea-Term rule, y is mutably reachable if a read-write reference to it occurs in the focus of execution t or the stack σ in a position other than in type selection (the reference is t-free in t or σ). By the Rea-Fld rule, y_2 is mutably reachable if its location is stored in the field of some mutably reachable object y_1 and this field has read-write type. The immutability guarantee does not say anything about objects which are yet to be created; these may be modified. We will prove that if an object y on the heap Σ in the initial configuration c is not mutably reachable, then it will not appear on the left-hand side of a write term in any execution starting from c.

5.2 Proofs

Figure 12 summarizes the overall structure of groups of properties that we have proved about the type system and their major dependencies. The full statements of all lemmata and their proofs are in a supplementary technical report [6]. We discuss the most significant properties in the rest of this section; they are identified by numbers in Figure 12.

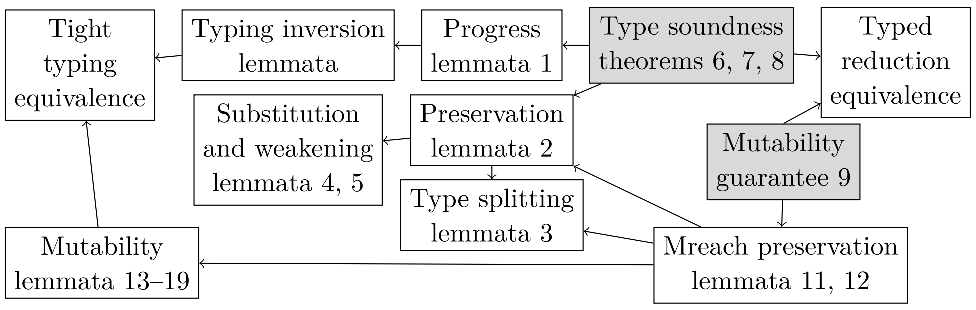

Figure 12 Overview of properties and dependencies within proofs of the main theorems.

$$\frac{\begin{array}{c} t_0 = w_1.a \quad \mathrm{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \quad \mathrm{F};\rho_1 \vdash w_1 : \{a : T_4..T_3\} \quad w_1 \to y_1 \in \rho_1 \\ y_1 \to \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma_1 \qquad\qquad \mathrm{F};\rho_1 \vdash T_3 \ \mathbf{mu}(r)\ T_7 \\ \mathrm{F} = \mathrm{F}_3, y_2 : \mu(s_1 : R_1) \wedge \{\mathsf{M}(r_0) : \bot..\bot\}, \mathrm{F}_4 \\ T_2 = \mu(s_1 : R_1) \wedge \{\mathsf{M}(r) : \bot..(T_7 \vee w_1.\mathsf{M}(r))\} \qquad \rho_2 = \rho_1, w_2 \to y_2 \end{array}}{\mathrm{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \mathrm{F}, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1 \rangle} \text{(TR-Read)}$$

$$\frac{\begin{array}{c} t_0 = \mathsf{let}\ z = \nu(s : R)d\ \mathsf{in}\ t \quad \mathrm{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \quad \rho_2 = \rho_1, w_1 \to y_1 \\ T = \mu(s : R) \wedge \{\mathsf{M}(r_0) : \bot..\bot\} \qquad \Sigma_2 = \Sigma_1, y_1 \to [y_1/s][\rho_1]d \end{array}}{\mathrm{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \mathrm{F}, y_1 : T, w_1 : T \vdash \langle [w_1/z]t; \sigma_1; \rho_2; \Sigma_2 \rangle} \text{(TR-LetNew)}$$

Figure 13 Typed reduction.

5.2.1 Proof of Type Soundness

Type soundness ensures that evaluation of a typed term does not get stuck in a non-final configuration where no reduction rule can be applied. Similarly to kDOT, it is shown using two properties of reduction: Progress means that unless a typed configuration is final, a step can be taken. Preservation means that the step retains the type of the configuration.

We state the properties differently than kDOT. Typing a configuration requires a typing context, which after taking a step such as creating a new object, might have to be extended to give a type to the newly created location. The usual reduction rules do not specify how the typing context should change. For the proofs, we define a typed variant of reduction. It transforms configurations in the same way as the rules in Figure 10. Additionally, it requires the configuration to have a type, and also produces a typing context for the next configuration. This makes type preservation easier to state because the typing context is fixed, and makes it possible to state a similar preservation property for proving the immutability guarantee.

The typed reduction rules for the two interesting cases are shown in Figure 13. In TR-Read, a type for a new reference variable is constructed. This type must be precise enough so that the resulting term keeps the expected type, but at the same time, it must be read-only if either the field or the reference to the containing object was read-only. We construct the type by taking the recursive part of the heap type of the object and changing the mutability. The new mutability is an upper bound of the mutability of the containing object, expressed by $w_1.\mathsf{M}$, and the mutability of the field type given by $\mathbf{mu}$.

In TR-LetNew, both the new location y_1 and the reference w_1 are given a read-write type based on the object literal. The other typed reduction rules (not shown) are straightforward because the typing context does not change.

We state progress and preservation for each of these 6 typed reduction rules, which each handle one syntactic form of a term. Progress states that if a configuration with such a term in the focus of execution has a type, then a typed rule can be applied. Preservation states that the context produced by the rule gives the new configuration the same type. Lemmata 1 and 2 are examples of progress and preservation for the TR-Read rule.

▶ **Lemma 1** (PgRead). *If* $\mathrm{F} \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1\rangle : T_0$*, then there exist* w_2*,* T_2 *and* ρ_2*, such that* $\mathrm{F} \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1\rangle : T_0 \longmapsto \mathrm{F}, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1\rangle$*.*

▶ **Lemma 2** (TPRead). *If* $\mathrm{F} \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1\rangle : T_0 \longmapsto \mathrm{F}, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1\rangle$*, then* $\mathrm{F}, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1\rangle : T_0$*.*

Note that these lemmata also imply progress and preservation of the untyped reduction rules. For progress, since the premises of each typed reduction rule contain all the premises of the corresponding untyped reduction rule, if progress ensures that some typed reduction rule applies to a configuration, then the corresponding untyped reduction rule also applies. Preservation for untyped reduction rules requires that there exist some extended typing context in which the next configuration has the same type, and the typed reduction rules explicitly provide this context.

The proofs of these lemmata follow the recipe from [12] and [14]. We define precise, tight and invertible variants of typing for variables, and a tight variant of subtyping. Invertible typing together with heap correspondence ensures that objects used in Read, Write or Apply terms have the member needed for progress, and preservation. In an inert context, typing and invertible typing are equivalent.

Notable differences from the baseline DOT are in the TR-Read and TR-LetNew cases.

In Lemma 2, it must be shown that the new reference variable w_2 has the expected viewpoint-adapted type as defined by the TT-Read typing rule. This type T_2 is formed by an intersection of a read-only part and a mutability member declaration. To show that the reference has the read-only part of the type, we use Lemma 3 , which states that a reference w corresponding to a location y has the read-only version of the type of y.

▶ **Lemma 3** (RefT). *If* $\mathrm{F};\rho \vdash y : T_1$*, and* $\mathrm{F};\rho \vdash T_1$ **ro** T_2*, and* $\mathrm{F} \sim \rho$*, and* $w \rightarrow y \in \rho$*, then* $\mathrm{F};\rho \vdash w : T_2$*.*

Showing the same for the mutability part of the type is easy, because it is formed in the same way as in the TT-Read term typing rule.

The rule also changes the runtime environment ρ. Correspondence of ρ with the typing context is ensured because w is given the same type as y except for mutability.

In the TR-LetNew rule, it must be shown that heap correspondence is preserved. That is, the object on the heap has the type given to the new location y added to F. First, we show that definitions keep their type if references are replaced by locations, by Lemma 4. Then, we use a variant of a substitution Lemma 5 to show that if the definitions d of the object had type T_1 given by the DT-* definition typing rules, then under substitution of the location y for the self variable s, the definition will get the type by the HT-* typing rules. Preservation of ρ correspondence is ensured by giving w_1 the same type as y_1.

▶ **Lemma 4** (DeD). *If* $\mathrm{F}, s : T_2;\rho \vdash d : T_1$*, and* $\mathrm{F} \sim \rho$*, then* $\mathrm{F}, s : T_2;\rho \vdash [\rho]d : T_1$*.*

▶ **Lemma 5** (SubD). *If* $\Gamma, s : T_3;\rho \vdash d : T_1$*, and* $s \notin \Gamma$ *and* Γ **vis** y *and* $\Gamma;\rho \vdash y : [y/s]T_3$*, then* $\Gamma, y/s : T_3;\rho \vdash [y/s]d : [y/s]T_1$*.*

Finally, weakening lemmata state that adding variables to the typing context preserves typing derivations.

With progress and preservation lemmata proven for individual cases, we can state a common progress and preservation Theorem 6.

▶ **Theorem 6** (TPP). *If* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T$*, then either* $\langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle = \langle \mathsf{v}w_1; \cdot; \rho_1; \Sigma_1\rangle$*, or there exist* t_2*,* σ_2*,* Σ_2*,* ρ_2*,* F_2*, such that* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T \longmapsto \mathrm{F}_1, \mathrm{F}_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$*, and* $\mathrm{F}_1, \mathrm{F}_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle : T$*.*

By induction on the number of steps, type soundness of typed reduction follows – Theorem 7. Because typed reduction affects typed configurations in the same way as untyped reduction, we can easily show the final type soundness for untyped reduction Theorem 8.

▶ **Theorem 7** (TyS). *If* $\vdash t_0 : T$*, then either* $\exists w, j, \Sigma, \rho, \mathrm{F}\colon \vdash \langle t_0; \cdot; \cdot; \cdot\rangle : T \longmapsto^j \mathrm{F} \vdash \langle \mathsf{v}w; \cdot; \rho; \Sigma\rangle$ *or* $\forall j\colon \exists t_j, \sigma_j, \Sigma_j, \rho_j, \mathrm{F}_j\colon \vdash \langle t_0; \cdot; \cdot; \cdot\rangle : T \longmapsto^j \mathrm{F}_j \vdash \langle t_j; \sigma_j; \rho_j; \Sigma_j\rangle$*.*

▶ **Theorem 8** (S). *If* $\vdash t_0 : T$*, then either* $\exists w, j, \Sigma, \rho\colon \langle t_0; \cdot; \cdot; \cdot\rangle \longmapsto^j \langle \mathsf{v}w; \cdot; \rho; \Sigma\rangle$ *or* $\forall j\colon \exists t_j, \sigma_j, \Sigma_j, \rho_j\colon \langle t_0; \cdot; \cdot; \cdot\rangle \longmapsto^j \langle t_j; \sigma_j; \rho_j; \Sigma_j\rangle$*.*

5.2.2 Proof of the Immutability Guarantee

A new essential property of the type system is the immutability guarantee, expressed by Theorem 9. It says that if an object exists in a typed configuration c_1, then either it is mutably reachable by **mreach** (and therefore can change), or it stays the same after any number of execution steps (never changes).

▶ **Theorem 9** (IG). *If* $y \to d \in \Sigma_1$*, and* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T$*, and* $\langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle \longmapsto^k \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$*, then either* $y \to d \in \Sigma_2$ *or* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle$ **mreach** y*.*

For the proof, we again make use of the typed reduction rules from Figure 13. We show two properties of **mreach**: First, Lemma 10 states that if an object is mutated in a reduction step, then it was mutably reachable before the step. Second, Theorem 11 states that **mreach** is preserved by typed reduction, that is, an existing object that is not **mreach** will never become **mreach** in the future. This has to be shown for each of the reduction rules.

▶ **Lemma 10** (MMR). *If* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T \longmapsto \mathrm{F}_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$*, and* $y \to d \in \Sigma_1$*, then either* $y \to d \in \Sigma_2$ *or* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle$ **mreach** y*.*

▶ **Theorem 11** (MP). *If* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T \longmapsto \mathrm{F}_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$*,* $y \to d \in \Sigma_1$*, and* $\mathrm{F}_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$ **mreach** y*, then* $\mathrm{F}_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle$ **mreach** y*.*

The proof of Lemma 10 is straightforward from the reduction and typing rules, because only the TR-Write rule modifies existing objects on the heap and the typing rule for write terms requires the object reference to have a read-write type.

The rest of this section is about proving Theorem 11. For each of the 6 reduction rules, we look at the changes in the configuration and typing context that affect the **mreach** relation.

In the TR-LetPush and TR-LetLoc rules, the only difference in the configuration relevant to **mreach** is that t-free object references are moved between the focus and the stack. No new references are introduced, and the heap, environment and context do not change. These cases are handled by Lemma 12, which states that under these conditions, no object becomes mutably reachable.

▶ **Lemma 12** (MPres). *If* $\mathrm{F} \vdash \langle t_2; \sigma_2; \rho; \Sigma\rangle$ **mreach** y*, and* $\forall x\colon (t_2$ **tfree** $x \vee \sigma_2$ **tfree** $x) \Rightarrow (t_1$ **tfree** $x \vee \sigma_1$ **tfree** $x)$*, then* $\mathrm{F} \vdash \langle t_1; \sigma_1; \rho; \Sigma\rangle$ **mreach** y*.*

For TR-Apply, Lemma 12 also applies, because the HT-Met rule ensures that variables other than the receiver and the argument are not visible, and therefore cannot be t-free in the body of the method.

In TR-Write, a location may be stored on the heap as a new value of a read-write field. The typing of write terms ensures that if the field is read-write, then the reference w_3 that provided the value was read-write, so the location y_3 was mutably reachable from the focus of execution.

A TR-LetNew step creates a new mutably reachable object. New objects are not covered by the immutability guarantee, but the new object may contain read-write fields referring to existing objects. Similarly to the Write case, the typing of field definitions in the object literal ensures that if the fields have read-write type, then the references in the literal must have been read-write.

The TR-LetNew rule also adds a new location and reference to the typing context. The preservation of **mreach** depends on an essential property of how mutability is defined: *existing* read-only references and fields cannot be made read-write by creating new objects and references. Lemma 13 states that if a variable v_1 is read-write in an inert context F with a new location y_2 added, then it was already read-write in the context F without y_2. (In other words, it keeps its mutability if the last location y_2 is removed from the context.) We state Lemma 14 for mutability of fields and similar Lemmata 15 and 16 for adding a new reference w_2 to F and ρ.

▶ **Lemma 13** (StnMLoc). *If* $v_1 \neq y_2$*, and* $\mathrm{F}_2 = \mathrm{F}_1, y_2 : T$*, and* $\mathrm{F}_2;\rho \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$*, then* $\mathrm{F}_1;\rho \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$.

▶ **Lemma 14** (StnMFLoc). *If* $y_1 \neq y_2$*, and* $\mathrm{F}_2 = \mathrm{F}_1, y_2 : T$*, and* $\mathrm{F}_2;\rho \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$*, then* $\mathrm{F}_1;\rho \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$.

▶ **Lemma 15** (StnMRef). *If* $v_1 \neq w_2$*, and* $\mathrm{F}_2 = \mathrm{F}_1, w_2 : T$*, and* $\rho_2 = \rho_1, w_2 \to y_2$*, and* $\mathrm{F}_2 \sim \rho_2$*, and* $\mathrm{F}_2;\rho_2 \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$*, then* $\mathrm{F}_1;\rho_1 \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$.

▶ **Lemma 16** (StnMFRef). *If* $y_1 \neq w_2$*, and* $\mathrm{F}_2 = \mathrm{F}_1, w_2 : T$*, and* $\rho_2 = \rho_1, w_2 \to y_2$*, and* $\mathrm{F}_2 \sim \rho_2$*, and* $\mathrm{F}_2;\rho_2 \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$*, then* $\mathrm{F}_1;\rho_1 \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$.

The case of adding a new *reference* to an existing location has a quite straightforward proof: in the typing derivation that gives a read-write type in the new context, we can replace the new reference by the corresponding location.

The case of adding a new *location* y_2 is more complicated, because the location has a type that may not be present anywhere else in the context. We can use the infrastructure of invertible typing to show that the mutability of a reference w_1, location y_1, or its field a can be derived from the type specified for w_1 or y_1 in F by tight subtyping. That is sufficient to prove Lemma 13, but in Lemma 14, we still need to show that the upper bound given to $y_1.a$ in the typing context is a tight subtype of $\{\mathsf{M} : \bot..\bot\}$. It is not possible to show that for subtyping of arbitrary types, even if both types do not reference the variable y_2. In particular, this is caused by subtyping of method types, where subtyping of the result type may be in a typing context that is not inert. Fortunately, we need this property only for the special case when the right-hand side of the subtyping is the simple type $\{\mathsf{M} : \bot..\bot\}$, as stated by Lemma 17. The premise $T_2\,\mathbf{nosel}\,y_2$ means that T_2 does not contain type selections involving y_2. The # sign indicates the use of tight subtyping.

▶ **Lemma 17** (StnSub). *If* $\mathrm{F}_2 = \mathrm{F}_1, y_2 : T_1$*, and* $\mathrm{F}_2;\rho \vdash_\# T_2 <: \{\mathsf{M} : \bot..\bot\}$*, and* $T_2\,\mathbf{nosel}\,y_2$*, then* $\mathrm{F}_1;\rho \vdash T_2 <: \{\mathsf{M} : \bot..\bot\}$.

Proof sketch of Lemma 17. We want to show that the subtyping derivation never needs to use y_2, that is, that we can derive the same subtyping without invoking the ST$_\#$-SelL or ST$_\#$-SelU rules on selections from y_2. There are 3 ways in which the original derivation may involve y_2: using the ST$_\#$-SelL or ST$_\#$-SelU selection rules, using subtyping of method types, or using rules such as ST-Top or ST-And1, where one of the types may be chosen freely. The proof proceeds in 3 steps, where in each step, we eliminate one of these issues by simplifying the types on both sides of the subtyping, and showing that the simplified types are related by a restricted version of subtyping.

For the first step, we define a reduction relation $\longmapsto^{\mathrm{s}}$. Because in an inert context, bounds are either tight or have a lower bound of $\bot$, using the ST$_\#$-Sel* rules does not add anything that could not be derived by other subtyping rules. Therefore, we can get rid of unnecessary uses of ST$_\#$-Sel* in the subtyping derivation by replacing type selections by their bounds. The reduction has two variants, $\longmapsto^{\mathrm{s}}_{\oplus}$ and $\longmapsto^{\mathrm{s}}_{\ominus}$, that replace a type selection by its upper (respectively lower) bound. The restricted version of subtyping allows using the selection rules only in the direction from the selection to the bound, not vice versa.

In the second step, we show that subtyping of method types is not needed using a reduction relation $\longmapsto^{\mathrm{m}}$ that replaces all method types by $\top$ or $\bot$. The restricted version of subtyping does not have the ST-Met rule.

In the last step, we show that y_2 never has to appear in the types involved in the derivation of the subtyping taken from left to right. The reduction $\longmapsto^{\mathrm{e}}$ replaces the remaining selections on y_2 by $\top$ or $\bot$.

In each of these steps, the type on the left-hand side, starting with T_2, is simplified to a supertype, and the $\{\mathsf{M} : \bot..\bot\}$ on the right hand side is not affected by the simplification. Therefore after all the steps, we get $\mathrm{F}_1;\rho \vdash T_2 <: \{\mathsf{M} : \bot..\bot\}$. ◀

Finally, in the TR-Read rule, a new reference is added to the context. The effect of adding the reference to the typing context is handled by Lemmata 15 and 16. A final piece in the proof of **mreach** preservation is mutability of the new reference w_2 created in a TR-Read step. It is created for a location that was stored in a field and is put into the focus of execution. We must ensure that the reference is read-write only if the field was read-write. Because the mutability of w_2 is a union of the field mutability and the mutability of the source reference w_1, we first show that if w_2 is read-write, then both the field and the source are read-write. For the field, we use Lemma 18 to show that the field has type $\{a : \bot..\{\mathsf{M} : \bot..T_7\}\}$, and then by Lemma 16, it must have had the same type before.

▶ **Lemma 18** (MUSub). *If* $\Gamma;\rho \vdash T_1\ \mathbf{mu}(r)\ T_2$*, then* $\Gamma;\rho \vdash \top <: T_2$ *or* $\Gamma;\rho \vdash T_1 <: \{\mathsf{M}(r) : \bot..T_2\}$.

For the object, we need Lemma 19 to show that if the upper bound of $w_1.\mathsf{M}$ in the new context is $\bot$, then w_1 was a read-write reference in the old context F.

▶ **Lemma 19** (WMu). *If* $w_1 \neq w_2$*, and* $\mathrm{F}, w_2 : T_2;\rho_2 \vdash_\# w_1.\mathsf{M}(r) <: \bot$*, and* $\rho_2 = \rho_1, w_2 \to y_2$*, and* $\mathrm{F}, w_2 : T_2 \sim \rho_2$*, then* $\mathrm{F};\rho_1 \vdash w_1 : \{\mathsf{M}(r_2) : \bot..\bot\}$.

We use the $\longmapsto^{\mathrm{s}}_{\oplus}$ relation from the proof of Lemma 17 above to show this. It simplifies $w_1.\mathsf{M}(r)$ to its bound: $\mathrm{F}, w_2 : T_2 \vdash w_1.\mathsf{M}(r) \longmapsto^{\mathrm{s}}_{\oplus} T_1$. Then, by further simplifying both sides of the subtyping, we find a type which is a supertype of T_1 and subtype of $\bot$ in F.

6 Related Work

6.1 Reference Mutability

Scala is influenced by Java, which has seen several extensions for reference mutability.

Javari [19] extends the Java syntax with reference mutability qualifiers. An unqualified reference type `T` is by default a read-write reference, while `readonly T` is a read-only reference. Javari comes with both a formal system based on Featherweight Java [11], and an implementation in the Checker Framework [13], using type annotations. The type system provides a transitive immutability guarantee, but allows opting out of that by declaring fields as always assignable, even through read-only references, or as always read-write, meaning viewpoint adaptation does not apply to them. (Non-transitive immutability could be achieved in the framework of our type system by removing the viewpoint adaptation in the typing rule for read terms and changing the definition of mutable reachability.) Qualifiers can be applied to any type in the program. Qualifier polymorphism is limited to the `romaybe` qualifier, which acts as a variable qualifier which can be instantiated at use locations by `mutable` or `readonly` – all `romaybe` qualifiers in a method declaration by the same qualifier. This allows Javari to express the first example T_{o1} from Section 4.5:

```
class C {
  T a;
  void m_set(T z) {a = z;}
  romaybe T m_get() romaybe {return a;}
}
```

The field is by default this-mutable and the `m_set` method is by default mutable with a mutable parameter. Javari allows mutability to be part of a type argument, so we could make the field `a` mutability-polymorphic like in the second example T_{o2}, but we would not be able to express the full example because Javari has no way to combine the mutability of the field with the mutability of the receiver of `m_get`.

The type system of ReIm [10] is simpler than that of Javari to enable fast and scalable inference of qualifiers. It has only 3 qualifiers, `mutable`, `readonly` and `polyread`. The polyread qualifier expresses simple qualifier polymorphism and viewpoint adaptation, similar to `romaybe` in Javari. Fields are either `readonly` or `polyread`; always-read-write fields are not supported. Usage of qualifiers is limited – they can be applied to any type, but not to type arguments of generic types. The first example T_{o1} can be expressed in ReIm as follows:

```
class C {
  polyread T a;
  void m_set(mutable C this, mutable T z) {a = z;}
  polyread T m_get(polyread C this) {return a;}
}
```

The second example T_{o2} cannot be expressed due to the lack of qualifiers on type arguments.

Immutable Generic Java (IGJ) [20] encodes mutability qualifiers in Java generics. It defines the first parameter of a class or interface to specify its mutability: the type `T<Mutable>` is read-write and the type `T<ReadOnly>` is read-only. This approach agrees with our desire to use features of the underlying type system to specify reference mutability. IGJ does not have viewpoint adaptation. Transitivity has to be opted into by declaring fields with a “this-mutable” type, using the mutability parameter of the containing class. As in Javari, fields may be declared always assignable. IGJ also supports object immutability by distinguishing `ReadOnly` references and `Immutable` references. The latter guarantee that the object will not be modified through any reference. Our first example T_{o1} can be expressed in IGJ as follows by explicitly specifying viewpoint adaptation in the return type of the getter method:

```
class C<I extends ReadOnly> {
  T<Mutable> a;
  @Mutable void m_set(T<Mutable> z) {a = z;}
  @ReadOnly T<I> m_get() {return a;}
}
```

The second example T_{o2} cannot be fully expressed for the same reason as in Javari.

Glacier [5] has a system based on class immutability. It has only two qualifiers that apply to classes. An `@Immutable` class must only have immutable subclasses and all fields must have immutable types. All other classes are `@MaybeMutable`. Class types other than the top class `Object` cannot be qualified when used and always have the mutability declared by the class.

The type systems above were implemented in the Checker Framework. This framework expresses type qualifiers using Java annotations, so that the Java syntax does not have to be modified. Qualifiers that apply to the receiver of a method are written by annotating the explicit `this` parameter. We achieve the same result in our approach using the explicit receiver parameter to a method. Explicit `this` parameters are not supported in Scala.

The type systems above share the limitations of Java generics and of Java; in particular, they do not support type intersections and unions.

The reference mutability system for the C# language [7] is the most flexible. As in the systems above, a type is composed of a qualifier and a normal type. In this type system, a generic class can be parameterized by both normal types and type qualifiers, but separately, by declaring a second qualifier parameter list after the type parameter list. Therefore, a class may have any number of qualifier parameters, which can be used to individually specify mutability of fields, method parameters and result types, or be passed as qualifier arguments to the types used at those places. Qualifiers can be combined by the special type operator $\rightsquigarrow$, which viewpoint adapts the second qualifier by the first one. This makes it possible to express a class similar to our second example T_{o2} from Section 4.5 as follows:

```
class C<PT> {
  PT T a;
  void Ms(PT T z) writable {a = z;}
  PC⇝PT T Mg<_><PC>() PC {return a;}
}
```

Other supported features include object immutablity and uniqueness in a multi-threaded context for safe parallelism.

6.2 DOT

In traditional DOT calculi, such as WadlerFest DOT, objects are immutable and their fields cannot be changed. A heap is not needed because object literals can be used directly as values in terms.

Mutable WadlerFest DOT [15] introduced mutability by means of mutable cells, which are allocated to hold a single variable and can be reassigned to other variables. In this scheme, an equivalent of a mutable field can be achieved by having a field which contains a mutable cell. Although the field itself cannot be reassigned, writing a value to this cell and reading the value of the cell behaves as writing and reading a value from a mutable field.

A move towards a more direct definition of mutable fields has been made in kDOT [12], where all objects are stored on the heap and referred to by object locations, and field assignments through such locations directly modify the object on the heap. This approach is closer to how object mutation works in Scala. With slight modifications, we used it as the baseline for our type system.

6.3 Programming Languages with Reference Mutability

Some programming languages have reference mutability as a part of their type system.

The const-qualified pointers and methods in the C++ programming language disallow mutation of the object pointed to [18]. However, `const` is not transitive, so it does not correspond to our definition of immutability. There is no concept of a viewpoint-adapted field. Qualifier polymorphism is not supported, but can be achieved using templates, or by directly duplicating the definitions in source code.

The D language has transitive `const` and `immutable` type qualifiers, which express reference and object immutability [1]. Like in the Java-based systems, D does not have intersection, union and dependent types, and objects have class types. Qualifier polymorphism is limited to templates combined with D's advanced support for support of metaprogramming and compile-time evaluation.

The Pony programming language defines reference capabilities, which qualify the type of every reference [4, 17]. The system is defined in the context of multiple simultaneously running actors, and by allowing only one actor to have a read-write reference or multiple actors to have read-only references to an object, it ensures that no race conditions can occur when modifying an object. The qualifiers not only specify whether the reference can be used to mutate an object, but also limit which other references to the same object may exist within the same or a different actor. Qualifiers corresponding to read-write and read-only references as used in this paper would be `ref` and `box`. Pony also has viewpoint adaptation applied to types of fields, and it can be also used from source code by writing arrow types, which allow viewpoint adapting a type by a type parameter, `this`, or `box`.

In Rust, mutability is tied to ownership. There can only be one mutable reference to an object. Read-only references are transitive. Although a reference can be qualified by lifetime parameters, its mutability is fixed, so there is no mutability polymorphism.

7 Conclusion

We have extended the DOT calculus with support for reference mutability. Our calculus, roDOT, supports a transitive immutability guarantee using viewpoint adaptation. Unlike most existing reference mutability systems that have been proposed for Java, which are expressed as a separate type system in parallel to the Java type system, our system is encoded using existing features of the DOT type system. Specifically, the mutability of an object is encoded using a type member, which makes it possible to refer to it in other types using a path-dependent type. An important and necessary enhancement to DOT is a separate notion of a reference as opposed to just a heap location, which makes it possible to distinguish mutable and read-only references to the same location. We have proven type soundness of roDOT, as well as the immutability guarantee that it provides. The calculus can serve as a formal foundation for future work on a principled implementation of reference mutability in a Scala compiler.

References

1 Andrei Alexandrescu. *The D programming language*. Addison-Wesley, Upper Saddle River, N.J, 2009.

2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. `doi:10.1007/978-3-319-30936-1_14`.

3 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 233–249. ACM, 2014. `doi:10.1145/2660193.2660216`.

4 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 1–12. ACM, 2015. `doi:10.1145/2824815.2824816`.

5 Michael J. Coblenz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Glacier: transitive class immutability for java. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 496–506. IEEE / ACM, 2017. `doi:10.1109/ICSE.2017.52`.

6 Vlastimil Dort and Ondřej Lhoták. Reference mutability for DOT - roDOT definitions and proofs. Technical Report D3S-TR-2020-01, Dep. of Distributed and Dependable Systems, Charles University, 2020.

7 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 21–40, 2012. `doi:10.1145/2384616.2384619`.

8 Philipp Haller and Ludvig Axelsson. Quantifying and explaining immutability in scala. In *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, pages 21–27, 2017. `doi:10.4204/EPTCS.246.5`.

9 Jason Z. S. Hu and Ondřej Lhoták. Undecidability of D<: and its decidable fragments. *PACMPL*, 4(POPL):9:1–9:30, 2020. `doi:10.1145/3371077`.

10 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 879–896, 2012. `doi:10.1145/2384616.2384680`.

11 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. `doi:10.1145/503502.503505`.

12 Ifaz Kabir and Ondřej Lhoták. κDOT: scaling DOT with mutation and constructors. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*, pages 40–50, 2018. `doi:10.1145/3241653.3241659`.

13 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 201–212. ACM, 2008. `doi:10.1145/1390630.1390656`.

14 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *PACMPL*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870.

15 Marianna Rapoport and Ondřej Lhoták. Mutable WadlerFest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*, pages 7:1–7:6, 2017. doi:10.1145/3103111.3104036.

16 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 624–641. ACM, 2016. doi:10.1145/2983990.2984008.

17 George Steed. A principled design of capabilities in Pony. https://www.ponylang.io/media/papers/a_prinicipled_design_of_capabilities_in_pony.pdf.

18 Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Upper Saddle River, NJ, 2013.

19 Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 211–230, 2005. doi:10.1145/1094811.1094828.

20 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 75–84, 2007. doi:10.1145/1287624.1287637.